Developing Embedded Software Using Compile-Time Virtualisation

Ian Gray

This thesis is submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science
University of York

September 2010

Abstract

The architectures of embedded systems are becoming increasingly non-standard and application-specific. They frequently contain multiple heterogenous processing cores, non-uniform memory, complex interconnect or custom hardware elements such as DSP and SIMD cores. However, programming languages have traditionally assumed a single processor architecture with a uniform logical address space and abstract away from hardware and implementation details. As a result, the programmer is prevented from making efficient use of unique hardware features by the abstraction models of the programming language and must use abstraction-breaking techniques such as libraries, OS calls, or inline low-level coding.

This thesis describes Compile-Time Virtualisation (CTV), a virtualisation-based technique for assisting the mapping of general-purpose software onto complex hardware. CTV introduces the Virtual Platform, an idealised view of the underlying hardware that presents a simplified programming model. The Virtual Platform ensures that general-purpose code will execute correctly regardless of the complexity of the actual platform. In order to effectively exploit application-specific architectures, CTV allows the programmer to influence the mappings implemented by the Virtual Platform, for example to target specific processors or hardware elements. CTV differs from existing run-time virtualisation systems in that its virtualisation layer only exists at compile-time, resulting in a system which displays minimal run-time overheads.

An implementation of CTV called Anvil is developed which is evaluated alongside the general CTV approach. Experiments and simulations demonstrate that CTV-based systems can be used to efficiently target a wide range of complex systems.

Contents

List of Figures

Acknowledgements

First and foremost I would like to thank my supervisor Neil Audsley for his advice and support, not just over the course of this thesis but throughout my entire academic life at York. From accepting me as an undergraduate through to the completion of my thesis, his help and guidance has been invaluable.

I also extend my gratitude to all of the members of the Real-Time Systems Group at York for providing an enjoyable and closely-knit working environment. Particularly, my assessor Andy Wellings, and my colleagues Jack Whitham and Nick Lay who were always around to answer my many questions and whose work on the RTS Virtual Lab was essential for the completion of my experiments.

Finally, I would like to thank my family, friends, and my girlfriend Erin, without whom this would not have been possible.

Declaration

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Philosophy of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references.

Some work in this thesis is based on research by the author that has previously been accepted for publication in international conferences. [92, 93, 94]

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ....(candidate)

Date ....

Chapter 1

Introduction

Embedded systems are application-specific computer systems that are deployed as part of a larger device or system. They are transparently integrated into their host systems so that the user is not explicitly aware of their presence. Unlike general-purpose computers they frequently perform only a small set of dedicated tasks that does not change, although with the development of user-programmable smartphones and similar devices even this distinction is becoming increasingly blurred. Consumer electronics is one of the largest markets for such devices, and the International Technology Roadmap for Semiconductors [118] shows their complexity (for example the number of integrated processors in a design) is increasing rapidly. The automotive industry makes extensive use of embedded systems to create engine management or system diagnostic modules, and they are also frequently deployed in high-integrity systems such as aeroplanes, factories, medical devices and power stations. In general, almost all digital devices contain some kind of embedded computer system.

Embedded systems are becoming ubiquitous. In the past, the substantial cost associated with digital technology meant that computer systems would only be used in situations that actively required computer control and would be otherwise impossible if attempted by a human operator. Precision engineering is one such example, where a human would be incapable of the fine motor skills required and so computer-controlled actuators are used. However as technology progressed and the cost of microprocessors decreased, embedded systems were integrated into more and more devices where computer control was not strictly essential but instead simply added new features or improved existing ones. As silicon technology advanced further, processors have become so inexpensive that they are now commonly used to actually reduce the cost of a system. Often a large number of integrated circuits can be replaced by a single embedded processor with no loss in computational power, thereby reducing the overall build cost of the system. Indeed, the pervasiveness of embedded systems is so great that of the nearly 8.3 billion microprocessing units shipped in the year 2000, 8.14 billion (98%) were used in embedded applications [217].

This trend is allowing technology to move towards the idea of *pervasive computing* [195], where a large number of small, inexpensive embedded devices are deployed throughout all facets of everyday life. These devices are frequently networked, either to each other or via the internet. For example, lighting, temperature and other controls in the rooms of a house might be networked with biometric sensors worn by inhabitants to allow the internal environment to be automatically monitored and adjusted. The main concept of pervasive computing states that the user does not need to consciously interact with the system, it instead integrates itself into everyday life. This was described by Mark Weiser as follows:

> ”The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.” [236].

Embedded systems present a unique set of design challenges [160, 104]. The computational requirements placed on embedded systems are constantly increasing, but their physical size severely limits the amount of processing power and memory that they have access to. Furthermore, embedded applications are often battery-powered so they must minimise the amount of energy they consume. As a result, the architectures of embedded systems are frequently more complex and application-specific than that of a general-purpose system.

1.1 Characterising embedded architectures

Limits on single-core clock speed mean that the designer cannot simply keep using faster and faster processors, they must instead leverage parallelism and include multiple processing elements in their designs. Due to the general trends expressed by Moore’s Law, transistor counts of hardware architectures double approximately every 18 months. This has allowed hardware designers to integrate an increasing number of cooperating elements into a single design. This has become known as the System-on-Chip (SoC) paradigm [83] and is a divergence from the common single-processor, single-memory space assumed by most programming languages.

The application-specific nature of embedded systems means that the designer will use different kinds of processing element to perform the various subtasks that are required by the system. Control logic will be implemented on a simple processor core with shallow pipeline, limited throughput and no floating-point unit, whereas encoding or decoding tasks might be assigned to dedicated DSP cores.

However, it is not just processing units that are non-standard. Memory access times have been increasing much slower than processor clock speeds [241] - a phenomenon known as the ‘memory gap’ - making memory access the bottleneck in most general-purpose systems. The memory gap is illustrated in figure 1.1. Caches can alleviate this to some extent, but they consume a large amount of power and silicon area and they make it very difficult to reason about the worst-case execution time of the system. As embedded systems interact with the outside world they are frequently classified as real-time systems [31], and so the designer must be able to perform this analysis to prove that their system will meet its timing constraints. Due to all this, the memory layout of embedded systems is usually as non-standard and application-specific as the processing devices that it is built around, making use of a range of different memory technologies and topologies.

Finally, because embedded systems have a predetermined function, it is common for some parts of that function to be implemented using dedicated hardware rather than software. Such cores are frequently orders of magnitude faster than an equivalent software implementation and consume less power, at the expense of greater silicon area and manufacturing cost.

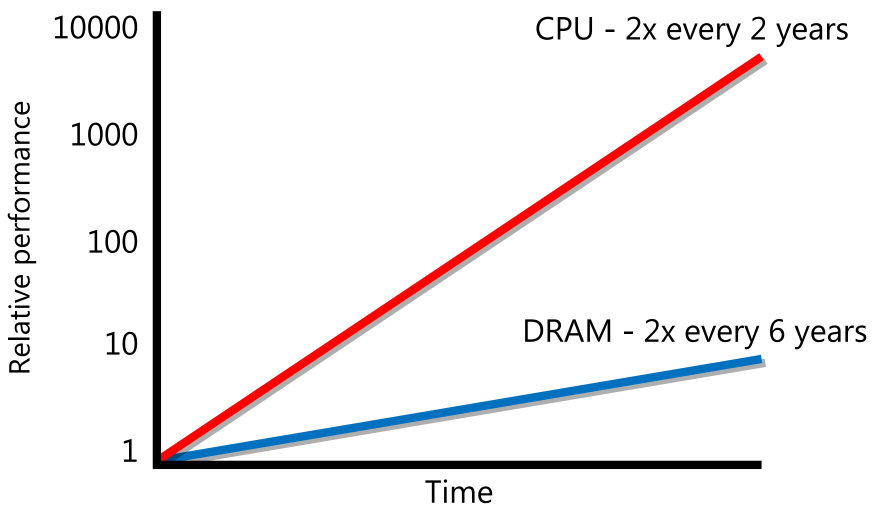

Figure 1.1: The ‘memory gap’ - memory access times have not scaled at the same rate as processor performance. Equivalent graphs show similar gaps between application complexity vs programmability and power consumption vs battery capacity.

These factors have led to an explosion in the variety of on-chip architectures being deployed. A major trend is towards more specialised, application-specific designs that may contain:

- Multiple heterogenous processing elements [87, 35].
- On-chip networks and buses, possibly spanning clock domains [59, 139, 240, 268]
- Non-standard memory hierarchies with shared memory and the integration of new memory technologies [76, 11, 22]
- Unique features of the implementation fabric such as DSP cores [210, 243], SIMD units [122, 190], or other custom hardware [98, 81]

Such architectures are non-regular and are heavily biased towards efficient execution of a specific task. Modern smartphone architectures are good examples of this, as they are optimised for low-power usage and contain a wide range of unique I/O devices. A generalised smartphone architecture is shown in figure 1.2. Modern examples have at their centre a System-on-Chip device such as the Texas Instruments OMAP [58] which comprises a main processor, embedded memory, and hardware graphics acceleration. This performs the majority of application processing, runs the phone’s operating system, and renders graphics. Outside the central processor there is a collection of external devices connected over a diverse selection of on-chip buses, each optimised for their expected traffic patterns. Modern smartphones are multicore devices because they have a dedicated baseband processor which receives high-level commands from the main processor and implements the low-level communications over the phone’s radios. Some of the deviations from general-purpose design that allow low power operation are:

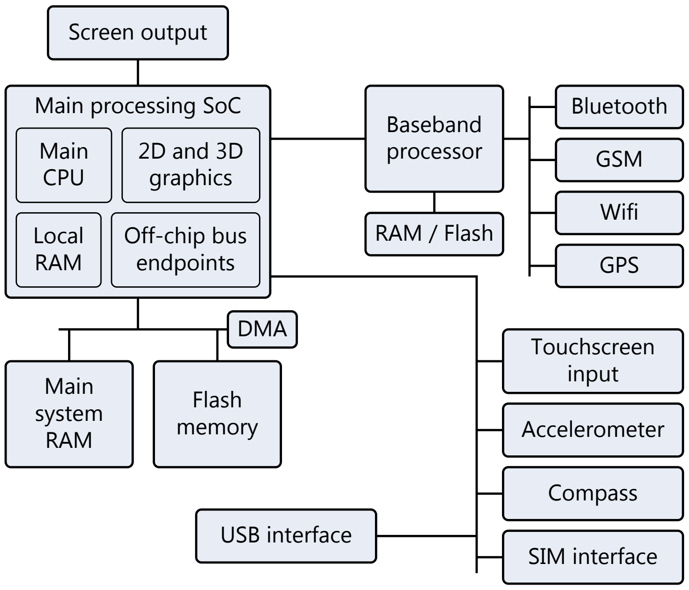

Figure 1.2: An example modern smartphone architecture.

- They use processors with low clock speeds and implement complex calculations directly in dedicated logic. Dedicated logic uses much less power than a powerful processor, but takes up space and increases hardware complexity.
- They use processors with custom instruction sets tailored towards the target application.
- All devices in the system use aggressive power scaling and sleep modes. Devices turn off as much as possible, at the expense of throughput.
- Modular design allows for entire sections of the architecture (such as the Bluetooth stack for example) to be deactivated when not in use.

All of these features require some amount of developer attention in order to be used effectively. Developing for such diverse platforms requires that the programming model used is amenable to architectural variety and can effectively exploit the hardware to its full capacity. The following section discusses the architectural support provided by the most common programming languages and highlights the areas in which this support is insufficient.

1.2 Programming embedded architectures

Programming complex embedded architectures is a considerable challenge. The first embedded systems were relatively simple and performed only a small number of fixed operations. They still contained examples of the non-standard features described above, but because of the small amount of functionality that they had to implement the majority of systems were programmed in assembler with program sizes of only a few thousand lines. Manual assembly programming is error-prone and requires a high degree of programmer skill, but it also allows the greatest amount of expressive power over the operation of the processor and the system as a whole. The programmer can directly control features such as memory access, register allocation and the use of custom hardware without being limited by the expressive power of a high-level language or compiler.

Modern embedded systems are now considerably larger and many are programmed with millions of lines of code. This increase in complexity makes the exclusive use of assembly programming infeasible and forces the adoption of high-level languages. However, these languages evolved for use on general-purpose computers and so do not provide abstraction models that allow the programmer to reason about architectural concepts. In general, the implementation architecture of languages (such as C or Ada) is assumed to be a single processor with a standard instruction set, no external hardware elements and a single, contiguous, logical address space. If multiple processors are present then a symmetric multiprocessor architecture with operating system support is assumed. Deviations from this require the programmer to break the abstraction models of the language, using techniques such as in-line assembly programming, code annotations, custom compilers, language extensions and custom linker directives. These techniques reduce programmer comprehension and limit traceability and debugging.

As there is no sign that computation requirements for embedded systems will stop increasing, it is expected that the kinds of architectures being developed for use in such systems will continue to get more complex, and so this problem will worsen over time.

1.3 Architecture support in modern languages

Language support for complex modern architectures is poor in common programming languages due to the manner in which they have developed over the last 40 years. The first high-level languages like C [129] and Fortran [212] were developed to reduce the complexity inherent in assembly programming. For example, C was developed to assist in the porting of the UNIX operating system, which was originally developed in assembly language [129].

Much of the complexity that is managed by high-level languages comes from architecture-specific features that the programmer does not want to deal with, such as manually assigning registers to program variables or management of the stack and heap. Due to the observation that the vast majority of computer systems in use were single-processor systems with a single contiguous block of memory, such an architecture was a reasonable assumption for the languages of the time to use. Fortran, which was targeted at supercomputers of the 60s and 70s hides all architectural details, including parallelism, from the programmer.

These languages made programming much easier by making assumptions based on a standard target architecture and hiding all hardware concerns. Unfortunately, the legacy of these choices has meant that modern programming languages can hide what are now important architectural details from the programmer.

This section examines the ways in which three major languages, Ada [15], C/C++ [72], and Java [9] can express and exploit the modern architectural features identified previously in section 1.1. These three languages are considered because C and C++ are by far the most common high-level languages used in current embedded systems, Ada is a rigourously-defined language developed to be suitable for use in safety-critical embedded systems, and the Real-Time Specification for Java [91] is an attempt to make the popular Java language suitable for such systems also.

1.3.1 Parallelism

In general, modern languages provide reasonable support for parallelism. Ada and Java both include the concept of concurrency as a native part of the language and so therefore their run-time systems both include schedulers that the programmer can control. Also in both Ada and the Real-Time Specification for Java (RTSJ) [91] the programmer can control task/thread priorities and scheduling schemes that allow a high level of control over the units of concurrency in their program and the ways in which they interact. C/C++ provide no language-level support for concurrency at all, but it is expected that they will be used under a POSIX-compliant operating system which is guaranteed to provide a similar level of concurrency control though system calls. The POSIX threading library is called pthreads [115] and provides dynamically-created threads and synchronisation using mutexes and condition variables. The pthreads interface is simple and does not provide high-level thread control, meaning that it can be challenging to write and debug large multithreaded programs to ensure liveness and the absence of deadlocks.

All of these languages carry the underlying assumption that the target architecture is comprised solely of relatively equivalent processors with a single shared block of memory. In Ada 95 it is not possible to directly express which processor should be assigned which task, demonstrating that the language assumes that it is not of critical importance. Newer revisions of the language are likely to address this [237]. Using the thread affinities of the RTSJ it is possible to specify that threads should only execute on certain processors, however the default is for threads to be able to migrate between all processors and affinities simply narrow this. This places a requirement on the language's run-time system to implement thread migration, and requires a system-wide scheduler. In systems with separate memory spaces migration is usually very difficult because data needs to be kept physically close to the thread that uses it in order to maintain performance. Affinities do not currently allow the programmer to express this. Also, if the processors in the system use different instruction set architectures (ISAs) then migration is not possible without recompilation or the use of multi-architecture binaries (such as Java bytecode). Finally, with the introduction of DSP cores and vector processors the programmer has to manually split their code between the varying processing devices and separately compile each program to make use of the different capabilities of these cores.

Another option available to programmers is to make use of an autoparallelising compiler, described in section 2.1.1. Autoparallelising compilers automatically extract the parallelisation inherent in the input code, at the penalty that they often cannot extract as high a degree of parallelism as would be possible though manually rewriting the code. When using an autoparallelising language or compiler, because the programmer does not manually define the parallel units (threads, processes, etc.), they cannot map them to the processors of the target architecture. This leads to the same problems already discussed in this section.

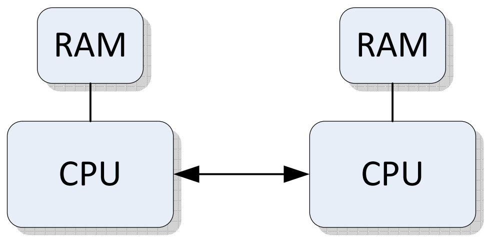

Figure 1.3: An architecture with two separate memory spaces can be difficult to target in standard programming languages.

1.3.2 Memory

In C/C++ all memory is presumed to be equivalent. That is, that the memory is all the same type (no scratchpads [11, 207], locked caches [231], etc.) and that access time is unrelated to address. C's arrays and pointer arithmetic relies upon the system appearing to have a single contiguous block of main memory starting at address 0, but in practice this limitation can be ameliorated by run-time address translation in the kernel and MMU. However, as there is no concept of different blocks of memory in C, the programmer cannot express that different blocks have different properties, such as access latency or average contention. The family of I/O instructions on x86 processors (`IN`, `OUT`, etc.) can only be used through the use of inline assembly because the programmer cannot express to the compiler which data accesses are I/O and which are memory. Related to this, variables are placed in memory by the linker, outside of the programmer's control. The use of thread affinities can position computation, but only if a compatible kernel is used and it can not position data.

This situation is improved in Ada, which still has most of the problems of C but allows the use of representation specifications to give the programmer a higher level of control over the placement of program data. This mechanism was included for the programming of low-level device drivers however, and it is unwieldy use on general-purpose memory for standard program variables. Representation specifications allow the programmer to specify the absolute address that data should be stored at, which can be combined with a very low-level knowledge of the target architecture to place data in specific memory spaces throughout the design. However both languages still assume a single consistent address space across all memory in the system. This is reasonable for a modern SMP system [96] (which these languages tend to target), but in embedded systems this is rarely the case. A system with two processors, each processor with its own memory (figure 1.3) exposes the weakness in this approach. An address map is a feature of a processor $\rightarrow$ memory connection rather than a program, but since the days of C and Fortran they have been assumed to be equivalent.

Furthermore, it is not reasonable for the programmer to have to manually place all the variables in their system as this becomes too onerous once a system grows large and complex. Linkers would be able to allocate addresses effectively throughout a complex memory space, but they are restricted by the assumptions made by source languages. The programmer is not given the ability to influence the linker to place memory items in specific parts of the system.

1.3.3 Unique hardware elements and custom hardware

Support for unique hardware elements is limited in most modern languages. Due to the fact that the abstractions of these languages were developed under the assumption that the target architecture would be fixed and uniform, the programmer has to work outside the language to exploit non-standard hardware elements. Techniques such as Ada's representation specifications or the use of hand-written linker scripts are used to fix program variables to specific addresses and thereby allow the manipulation of memory-mapped devices. Equally, access to these hardware features require manual function calls in all of the major languages. The hardware feature can never be used as a first-class part of the language. For example, if an architecture contains a vector co-processor it should be automatically used by the language for calculating the result of vector operations. However this is not the case and the programmer is forced to manually construct driver functions and pass operands to the co-processor specifically. The end result is broadly the same, but the final code is much less readable and is not portable to an architecture that contains, for example, two such co-processors.

1.4 Thesis Aims

This work aims to explore the many reasons why developing software for modern embedded systems is problematic and proposes a possible solution. Modern embedded systems will be characterised and from this, a new abstraction technique for embedded development will be presented that fulfills the following goals:

- **Heterogeneous architectures:** The main goal of this work is to present a system that can aid development of software for modern embedded architectures. The system should also be able to target the kind of future embedded architectures that can be forecast from current trends. These architectures are likely to contain dozens, perhaps hundreds or thousands, of heterogeneous processing devices that are served by a highly-irregular memory hierarchy consisting of many different memory technologies. Communication between processing devices will use a variety of media, such as shared buses and on-chip networks. Many existing solutions for programming multicore architectures display scalability problems and are only suitable for use over a small number of processor cores.
- **Minimal run-time overheads:** Due to the intended target domain of embedded and real-time systems, it is undesirable to introduce a heavyweight middleware layer, as has been done with systems such as CORBA [183]. This work aims to minimise run-time overheads as much as possible because this allows designers to create system architectures with reduced clock speeds and lower architectural complexity. Systems with lower clock speeds can consume much less power [142], which is of particular importance for devices that are battery-powered (such as mobile phones or sensor networks). Even for mains-powered devices, lower power consumption results in looser demands for cooling and greater manufacturing tolerances.
- **Predictability:** Due to the fact that many embedded systems are also real-time systems, it is critical that the proposed solution does not undermine execution time analysis or traceability.

- **Compatibility with existing languages and toolchains:** The proposed solution should remain as compatible as possible with the languages and tools that are already used in the embedded industry. Formal verification of compilers is a very expensive and time-consuming process, so companies are often unwilling to discard this effort simply to adopt a new programming language. Also, adoption of a drastically-different language reduces the productivity of software engineers whilst they learn the new tools. This work proposes a system that is language-agnostic, allowing it to make use of existing languages but to also benefit from the development of more-suitable embedded systems languages in the future.

1.5 Hypothesis

Abstraction and virtualisation have been shown to be useful techniques for hiding implementation complexity and providing a high-level programming model to aid software development. However, they introduce significant overheads and the programmer cannot influence the mapping of their application when targeting non-standard architectures. This thesis contends that moving the virtualisation layer from run-time to compile-time will allow the programmer more control over the implementation of the system, resulting in support for a much wider range of target architectures, the exploitation of unique hardware features, and lower run-time overheads.

1.6 Thesis Structure

The structure of this thesis is as follows. Chapter 2 looks at related work and analyses the current problems encountered in embedded systems development. Chapter 3 then introduces Compile-Time Virtualisation (CTV) as a potential solution to these problems. Chapter 4 discusses Anvil, an implementation of CTV that is designed to show its applicability to real-world systems. Chapter 5 evaluates CTV and Anvil against previous work and the thesis hypothesis, and chapter 6 concludes with a summary of findings.

Chapter 2

Literature

This chapter will give an overview of relevant research that has attempted to describe, analyse, or efficiently exploit heterogeneous embedded architectures. The chapter will particularly focus on modern embedded systems by considering the elements that such architectures commonly contain, what challenges they pose, and the ranges of solutions that have been proposed to overcome these challenges. The chapter is split into three main approaches to the problem of targeting non-standard architectures:

- **Describe the architecture from a software viewpoint.** If the programmer can include hardware considerations at the same abstraction level as their software then it is possible to give the compiler and toolchain more mapping information. Section 2.1 looks at new languages or language extensions that all attempt to do this by bringing architectural information up to the source-code level. The section describes the common features of embedded architectures, and then looks at the way that each of these features can be represented in software.
- **Hide the architectural complexities.** Abstraction is an essential feature of software development that is designed to help the programmer to manage the complexity of large systems. Abstraction can be provided by the programming language or operating system (as discussed in section 2.1), but a branch of research attempts to completely hide architectural details from the programmer through the insertion of a run-time virtualisation layer that handles software to hardware mapping. Section 2.3 considers virtualisation and the way that this can be done.
- **Generate the hardware along with the software.** If the target hardware is generated from a software description then it is guaranteed that it will be compatible with the semantics of the program. There are two main approaches to this - section 2.4 discusses hardware description languages and high-level synthesis systems and section 2.4.4 discusses hardware/software co-design.

The chapter also discusses the implementation fabrics that are commonly used to build embedded systems in section 2.5 because the unique properties of different implementation choices can affect the overall efficiency of the final design. Particular attention is paid to

FPGA-based systems because they afford the designer some interesting new options that are not possible with standard fixed hardware.

2.1 Architecture-oriented languages

As a result of the insufficient architectural support provided by mainstream languages (described in section 1.3) a number of new research languages have been developed. The following sections describe languages that have been designed to provide better support for concurrency (section 2.1.1), non-uniform memory architectures (section 2.1.3), and new computation paradigms such as data streaming applications (section 2.1.5) and data-path computation (section 2.1.6).

2.1.1 Parallelism

Making use of architectural parallelism is one of the main areas that modern languages are concentrating on. The clock speeds of single cores have reached a plateau, and without a major redesign it is unlikely we will see the kind of increases experienced from the late 80s up until around the year 2000 [215]. Therefore, to keep increasing in computational throughput architectures must include an increasing number of processor cores. At the time of writing, the largest supercomputer in the world contained 224,162 cores [202]. Clearly the challenge for programmers and language designers therefore is to create a development environment in which this massive amount of parallelism can be reasonably exploited.

As detailed in section 1.3, modern language support for parallelism is reasonable but lacking in some areas. All of the major languages rely on explicit parallelism - the programmer must manually identify the parallel tasks in their code an describe them as such. This is acceptable for low numbers of cores, but becomes much harder as the numbers of tasks and processor cores increases. Programmers have to contend with issues of deadlock, interference, scheduling, data passing and functional correctness; few of which are treated explicitly in existing languages. Two approaches have developed to manage this problem. First, a new class of parallel programming languages has emerged to try to allow the programmer to more easily express issues of parallel execution. Second, languages that attempt to automatically extract parallelism have been created. These two alternatives are described below.

UPC

Unified Parallel C (UPC) [34] is an extension to C designed for supercomputing environments. UPC allows the programmer to express a Single Instruction Multiple Data (SIMD) style of execution. A UPC program typically only has a single thread of execution per processor core, with parallelism coming from the ability to express massively parallel loop or array-based operations. UPC does not provide a programming model that is as high-level as that of Chapel, because it requires the programmer to explicitly use barrier and lock-based synchronisation control. Figure 2.1 shows a vector addition operation in UPC. No assumptions are made about

```
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];
void main() {
   int i;
   upc_forall(i=0; i<N; i++; i)
      v1plusv2[i]=v1[i]+v2[i];
}
```

Figure 2.1: UPC implementation of vector addition.

the data accessed in the `upc_forall` statement. If the data is not disjoint across parallel operations, the result is undefined.

UPC's main contribution is its strict definition of the memory model. The programmer's view of the system is that of a set of threads, each executing in a partition of a single global address space. This is known as the Partitioned Global Address Space (PGAS) model, described in section 2.1.4. UPC threads have affinity with a specific partition and also their own private memory space. As a result, UPC explicitly requires the implementation hardware to expose a global address space, which can be challenging to implement on some non-uniform memory architectures. Figure 2.1 shows an example of UPC.

Chapel

Chapel [38] is a parallel programming language developed by Cray with the stated goal of increasing the productivity of supercomputing platforms. Chapel is not an autoparallelising language, it instead aims to give the programmer abstractions that allow the expression of potentially parallel operations which can then be exploited by the compiler and runtime. Similarly, the programmer directs placement of data throughout the memory hierarchy and makes explicit use of synchronisation constraints.

Unlike UPC, Chapel programs contain a set of explicitly-defined threads of execution. Chapel allows the use of coarse-grained task-level parallelism (similar to Java threads or Ada tasks) that require explicit synchronisation, but also introduces constructs for fine-grained parallelism that use implicit synchronisation and data passing. Figure 2.2 shows the `forall` statement which expresses loop-level parallelism to the compiler, which can therefore automatically make use of available cores, pass input data appropriately, synchronise the worker threads, and collate output results, all without direct programmer input. Figure 2.3 shows a program that expresses that two processes may execute concurrently and again the compiler will automatically handle data flows and synchronisation. As with UPC, the language assumes that race conditions are not generated by the parallelisation. These two examples demonstrate that Chapel is a data-parallel language. The concept of data-parallel languages was developed in the late 80s to early 90s with the HPF [151], ZPL [146] and SISAL [77] languages.

As Chapel was developed to run on supercomputing architectures, it tends to assume a regular SMP or grid-based architecture. On this kind of system it can use the concepts of *locales* and *domains* to bind computation and data together onto specific nodes of the architecture. However, it can be challenging to make full use of entirely irregular architectures and it relies on the presence of a hardware-supported single shared memory space across the entire system.

```
var n: int = 1000;
var a, b, c: [1..n, 1..n] float;
forall ij in [1..n, 1..n]
   c(ij) = a(ij) + b(ij);
```

Figure 2.2: Chapel implementation of data-parallel matrix addition. (Fine-grained, data-parallel programming.)

```
computePivot(lo, hi, data);
cobegin {
   Quicksort(lo, pivot, data);
   Quicksort(pivot, hi, data);
}
```

Figure 2.3: Chapel implementation of task-parallel sort. (Coarse-grained, task-parallel programming.)

The memory programming model exposed by Chapel is broadly a PGAS model (section 2.1.4).

OpenMP

Unlike the programming languages discussed in the previous sections, OpenMP [39] is an API for multi-threaded shared memory parallelism that can be implemented on top of a given operating system with many different languages providing bindings to it. Currently the API is specified for C/C++ and Fortran and it has been implemented on many Linux and Windows-based platforms. OpenMP is designed for shared memory systems and cannot target distributed memory systems on its own. For such systems it is common for designers to use the combination of OpenMP and another API specification called MPI [97]. MPI (Message Passing Interface) is designed to provide an API for communication in a distributed system, and it is primarily used in cluster and high-performance computing contexts.

Due to its implementation as an API for languages like C and Fortran, OpenMP is limited in the amount of freedom that it has to express new execution models. As a result, it exclusively uses a ‘fork and join’ model of parallel execution, whereby a *master thread* is the only thread running in the system until it enters a parallel region, at which point a team of lightweight worker threads are created (or ‘forked’) to perform the work. At the end of the region, the main thread waits for all worker threads to complete before continuing. Using this model the programmer can express code that can be parallelised and the compiler will perform the necessary work. OpenMP can support similar constructs to Chapel’s `forall` and `cobegin` statements.

Other languages

A large number of languages have also developed to express similar constructs to UPC and Chapel. X10 [42] was developed by IBM as a derivative of Java to provide an object-oriented approach to parallel computing. It uses a similar partitioned global address space to UPC and attempts to prevent the locking problems that complicate parallel programming by introducing the concept of parent and child relationships for tasks. A child is not allowed to wait for a parent, easing analysis of code and helping to reduce deadlocks.

Intel Threading Building Blocks (TBB) [187] is a C++ template library which aims to simplify the low-level parallelisation primitives already provided by C++. TBB allows the definition of tasks which are automatically balanced across the cores of the system to perform load balancing and maximise cache hits. It offers similar constructs to those found in Chapel and UPC, but also provides many low-level constructs, such as mutexes and atomic fetch and add operations. TBB's memory model is similar to C++'s flat model with the addition of thread local storage. The use of MPI is recommended when using TBB on a distributed system with shared memory.

All of the other systems currently mentioned in this section require the programmer to manually identify the parallelism in their code. However, it is also possible to attempt to automatically derive this information from a single-threaded program. This can be done from standard languages using parallelising compilers (SUIF [244], Polaris [24]), or from specialised languages that have constructs to help the parallelising compiler, such as SieveC [63]. In general however, autoparallelising systems tend to perform rather poorly because they still rely on the structure of the input software. The programmer must carefully express their code in such a way that any potential parallelism is exposed. Code written purely for single-threaded execution tends to contain only a small amount of potential parallelism. For example, data dependencies that prevent parallel execution can often only be removed by complex restructuring of the control flow or the introduction of new data structures, both of which are outside the scope of most auto-parallelising systems. Also, the parallelisation step complicates debugging and verification because the code running on the target architecture is not the code that the programmer wrote. Finally, code that is automatically parallelised cannot be bound by the programmer to specific areas of the target architecture, meaning that its use is restricted to uniform cluster-based systems.

2.1.2 General-purpose programming on GPUs

Recently, a number of new languages have been developed to make use of graphics processing units (GPUs). Whether integrated as part of the motherboard chipset or as a dedicated expansion board, GPUs comprise a main processing core and a number of vector processing units and are designed to vastly accelerate the kinds of operations that are performed by 3D visualisations. The host computer issues commands to the main core which in turn uses its vector units to calculate the result (and store it as pixel data in a frame buffer typically). If the programmer can cast their algorithm in terms of the supported graphics operations they can be sent to the GPU for evaluation, and because GPUs typically have dozens of vector processing units and tightly-coupled dedicated RAM with very low latency the calculation will be performed in a fraction of the time that the host processor would require.

In order to make GPUs useful for general purpose computing, GPU vendors inserted programmable stages into their rendering pipelines, and extended the chip to support higher precision arithmetic than is typically required by graphics alone. Two important languages have developed to assist programmers in using these new features, CUDA [175] and OpenCL [169]. Both of these languages provide similar constructs to the rest of the parallelising languages discussed in section 2.1.1, but with an emphasis on loop-parallel operations that can

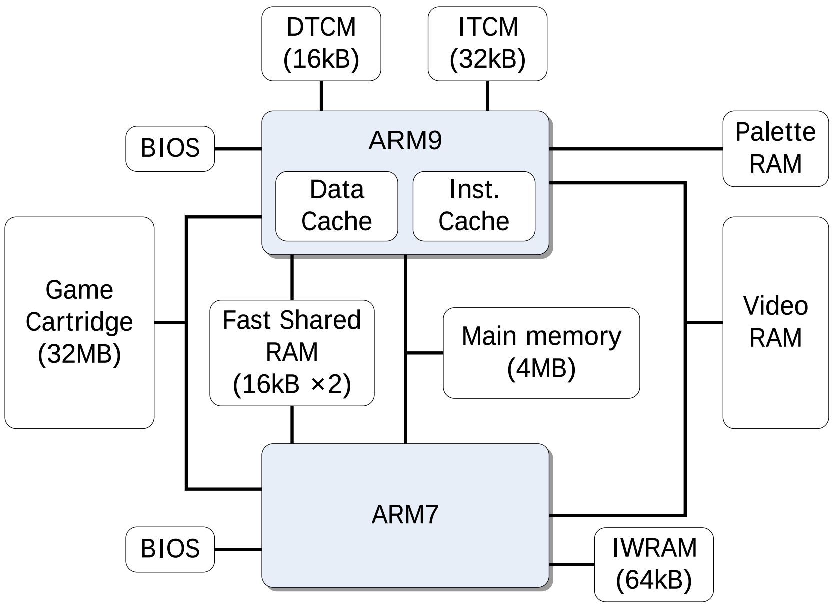

Figure 2.4: The memory layout of the Nintendo DS [111]. The dual-CPU system has 4MB of shared memory which is relatively slow, many banks of dedicated fast RAM that have much lower latency, and multiple caches.

be expressed as SIMD vector operations.

General-purpose programming on GPUs, or GPGPU as it has become known, gives regular computer users access to supercomputing-style programming environments, but the technique is not generally applicable for use on embedded architectures. GPGPU is aimed towards efficient utilisation of data streaming architectures with high degrees of data parallelism (SIMD systems) and does not support irregular embedded architectures.

2.1.3 Non-uniform memory architectures

The memory layout of modern embedded systems is becoming highly variable. As mentioned in chapter 1, memory access rates have not increased at the same rate as processor clock speeds leading to a disparity that has promoted the use of caches, and sometimes multi-layered caches. Also, as systems are moving to include greater amounts of parallelism (section 2.1.1) it is no longer efficient to require that all processor cores arbitrate for access to a single memory space. Architectures are including a mixture of shared and private memories with many different variations of DMA access and cache coherency algorithms. One such example is that of the Nintendo DS, a portable computer games console, which contains a highly non-standard memory system described in figure 2.4.

In general the following features are observed:

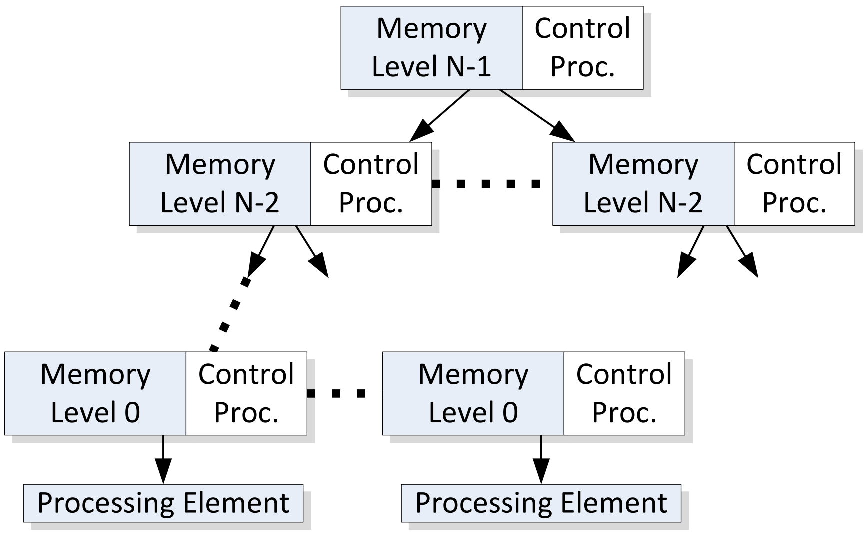

Figure 2.5: The Sequoia Abstract Machine Model

- **Levels:** Memory is frequently layered. Fast memory is kept close to the processor whilst large and slow memories are further away. This describes caches, but in the context of the System-on-Chip paradigm it also describes local memory and remote memory, where local memory is on the processor's memory bus and remote memory must be accessed through an on-chip network.
- **Layout:** Cores can have their own blocks of private memory and there can also be blocks of shared memory that are accessed by a number of cores. There may or may not be a single block of memory accessible to the entire system.
- **Caching:** Caches can be multi-layered and may use many different cache coherency mechanisms.
- **New memory technologies:** The system might employ novel memory techniques such as scratchpad memories or locked caches.

As has been discussed in sections 1.3 and 2.1.1, programming languages have developed assuming that system memory is a single contiguous block of memory, so when the architecture is no longer compatible with this assumption the programmer is forced to step outside the language to ensure their program operates correctly and uses the available hardware efficiently.

As a result, a number of programming languages have been developed with the intent of allowing the programmer to better exploit the memory hierarchy. Sequoia [76] is one such example. Sequoia is an extension to C that concentrates on the data transfers that take place throughout the execution of a program. It models the system memory as a tree of memories where higher memories in the tree are generally larger yet slower. Each memory module may have associated with it a control processor, which can operate on the data stored in that memory. This model is shown in figure 2.5.

Sequoia programs are composed from tasks. Tasks are abstract - they do not refer to physical

```
void task<inner> VectScale::Inner(in float A[N], in float x, out float Y[N])
{
   tunable T;
   unsigned int numBlks = (128+T-1)/T;

   mappar (unsigned int i = 0 : numBlks) {
      VectScale( A[i*T;T], x, Y[i*T;T] );
   }
}
```

Figure 2.6: Sequoia example of constant vector multiplication. The `mappar` statement assumes disjoint operations.

locations in the target architecture until a mapping phase that takes place during compilation. Communication between tasks is only possible by *task calling*, where a task at memory level i calls a subtask mapped to memory level $i-1$. Input arguments are then copied from level i to $i-1$ and when the task completes the output arguments are copied back. This explicit copying allows the compiler to make use of DMA engines and to maximise cache effectiveness. An example Sequoia program for adding two vectors is shown in figure 2.6. The main limitation of Sequoia is that it assumes a hierarchical memory layout in which processing elements pull data from 'far away' into caches or other tightly-coupled memories. Grid and cluster-based architectures may not fit well into this model. Also, when decomposing a problem into a Sequoia program, the programmer is still required to have a reasonable amount of architecture-specific knowledge. The resulting program can be mapped to other architectures but is likely to be quite inefficient without refactoring.

In addition to Sequoia, many of the languages presented in the previous section on addressing parallelism provide ways for the programmer to express memory features. UPC allows local and global arrays that have the concept of hierarchical memory, and the locales of Chapel, ZPL and X10 all convey similar features. These languages however, tend to assume a single flat memory space and concentrate on task-to-task transfers (horizontal) rather than the hierarchical (vertical) transfers that Sequoia considers.

Charm++ [126] is an object-oriented parallel programming language based on C++ primarily intended for use on regular supercomputing architectures. Charm++ uses explicit thread-level parallelism and a run-time system to pass remote method invocations between computation nodes. The mapping of *chares* (Charm++'s threading objects) to processing nodes is transparent to the programmer, which permits the run-time system to dynamically reallocate computation to support features such as dynamic load balancing, fault tolerance and application footprint scaling (how many computing nodes an application is assigned to). Charm++ is poorly suited for use in embedded architectures because its primary benefits are only apparent when used on a regular grid of largely homogenous processing cores. AMPI [108] is an implementation of MPI on top of the Charm++ run-time that augments the standard provisions of MPI with the dynamic capabilities of Charm++.

STAPL (Standard Template Adaptive Parallel Library) [186] is a C++ template library similar to STL which is designed to provide support for writing programs with shared or distributed memory. It provides a set of components similar to the components in the ISO C++ standard library. Programmers build applications using STAPL-derived data structures which are automatically distributed through shared memory by the STAPL run-time system. STAPL provides a uniform shared object view of the target memory architecture, throughout which STAPL-derived data structures are distributed. The physical distribution of STAPL objects can be assigned automatically or can be user-specified. A novel feature of STAPL is that it can present a flat shared memory architecture to the novice user or a partitioned global address space (section 2.1.4) to an advanced user depending on which API features they use.

2.1.4 Partitioned Global Address Space languages

A number of the programming languages already presented in this section fall into the category of Partitioned Global Address Space (PGAS) languages. PGAS is a memory model for parallel programming that represents one of the first major divergences from the uniform, flat memory model of early languages like C.

The primary difference between a flat model and the PGAS model is that a PGAS model includes the concept of *local* and *remote* memory accesses. In a flat model all memory accesses are treated as equal, and carry the implicit assumption that there are no significant differences in latency between addresses. In an SMP shared-memory system this assumption is reasonable because it is supported by the hardware. However in an embedded system with disparate memory spaces this is often not the case, as some accesses may require the navigation of numerous on-chip buses and so are much slower.

The PGAS model still assumes the presence of a single, contiguous global address space, but it allows the programmer to split that address space into a set of partitions. Threads (or the language's equivalent) and shared data items can then be assigned to reside within a single partition. All memory accesses by a thread to data that is in the same partition are termed local accesses. Accesses to data from a different partition are termed remote (or global) accesses. Local accesses are notionally cheaper than remote accesses, in terms of latency, throughput, power usage etc.

The PGAS model is sometimes informally subdivided into *synchronous PGAS* and *asynchronous PGAS* [194]. This distinction is not universally-applied and many authors simply use the term *PGAS* to refer to both forms. When the distinction is made, a synchronous PGAS model assumes that each partition runs on approximately similar hardware and so is tailored towards the execution of vector ope rations, highly-parallel programs, and the SIMD paradigm. This is the model commonly used in high-performance computing languages such as UPC. In contrast, the asynchronous model relaxes the homogeneity requirement and allows partitions to be much more disparate. For example, a traditional networked computing architecture where nodes are of different architectures, capabilities, and operating systems. Unlike the synchronous model which tends towards the execution of fine-grained parallelism, asynchronous partitions invoke coarsely-grained jobs on other partitions. The X10 language [42] is an example of this model.

The advantages of the PGAS model are that it provides more information about the memory hierarchy in a high-level way that does not overburden the programmer. The compiler and run-time system are still tasked with the complex task of implementing the communications and ensuring cache coherency, unless it is already provided by the architecture (such as with cc-NUMA). The programmer can use the memory partitions to give more guidance about the way in which their program should be implemented in an architecturally-neutral way. Placing a set of threads into the same partition informs the compiler and toolchain that they are tightly coupled and should be placed physically close together on the target architecture. Threads from different partitions are loosely-coupled and so may be mapped to separate regions of the architecture.

The main limitation of PGAS languages is the requirement for a single global address space (albeit a partitioned one) and that they tend to still assume perfect cache coherency across that address space. PGAS languages were motivated by high-performance computing architectures which are necessarily regular, so they are weaker at targeting highly non-uniform systems.

There are many examples of PGAS languages, many of which have already been described. Split-C [57], X10, Chapel [38], and Titanium [106] are some of the more well-known examples.

2.1.5 Data streaming architectures

When programming a conventional single processor system with a uniform global address space model, the programmer uses a ‘programmable state machine’ paradigm. This paradigm is served well by imperative programming languages like C, Java, Pascal, Fortran etc. However, hardware can take many different forms, some of which are easier to program when using a different computational paradigm.

Data streaming architectures are one such example. Broadly speaking, the computation performed by embedded architectures tends to be either control-driven or data-driven. Control-driven tasks involve reading the state of a small amount of input data (such as sensor values or user input) and then effecting some simple output as a response. The computation may be quite complex, but it is based on a small amount of input data and produces a relatively small amount of output data. A good example of this kind of system is a digital calculator. Input is very sparse and output is commonly only a single number, but still the calculations performed might be very complex. Processor-based systems can implement these designs well. Data-driven tasks, conversely, must cope with huge volumes of data that frequently produce similarly large volumes of output data. Video compression or real-time encryption are examples of this. The huge data rates of these problems means that processor-based designs are frequently incapable of performing fast enough, and custom architectures must be created. Data streaming architectures are designed as a directed graph where the nodes of the graph are *kernels* and the edges are data flows, as shown in figure 2.7. Kernels perform some fixed operation on their input data (such as a FFT, a linear congruence filter, etc.) and may be implemented using either simple combinatorial logic, specialised DSP cores, or with software running on full processor cores. Kernels therefore operate in parallel, and will have their own memory to avoid contention. Data flows are point-to-point to remove the need for bus arbitration and they make use of input and output buffers to improve performance and prevent pipeline stalls.

Standard imperative programming has trouble expressing these architectures because the programmer has to manually describe each kernel as a task or thread and the overall operation of the stream becomes obfuscated. Therefore, a class of languages known as *data streaming languages* has emerged to allow a more natural programming model. Streams-C [89], StreamC [162] and StreamIt [221] are all similar languages that bring the notion of a kernel into the language, but they also allow the programmer to connect kernels together in a high-level way and it is in this area where they provide the greatest contribution. Figure 2.8 shows

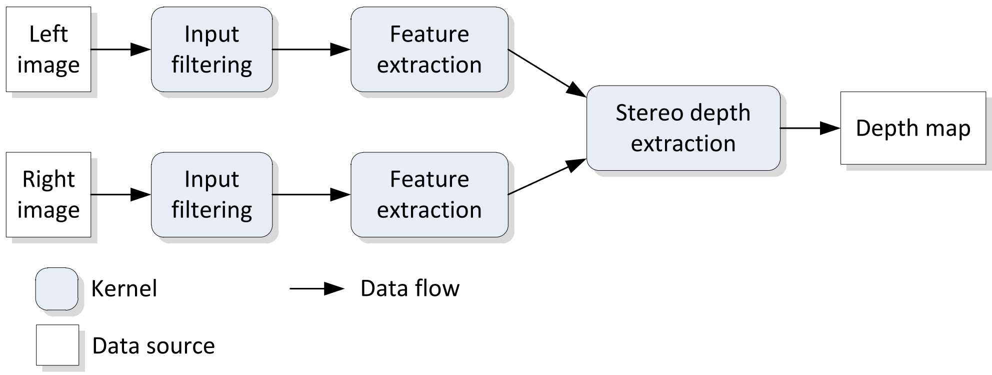

Figure 2.7: A data streaming architecture for performing stereo depth extraction.

an example of a Streams-C program which declares a single process with a number of input and output streams. Much more complex stream layouts are possible.

Standard imperative languages with notions of concurrency (like Java or Ada) allow parallel kernels to be defined, but they do not address the unique requirements of streaming architectures. Pipeline stages must be balanced to achieve a high throughput. As shown in figure 2.9, if stages are not of similar length then the system performance is degraded. The streaming languages above all provide compiler support to automatically split stages and high-level simulation or debugging support to assist the programmer in balancing their design. Also, the compilers of the above languages will generate flow control code which would have to be otherwise manually written by the programmer.

2.1.6 Datapath architectures

Datapath architectures are another implementation style commonly used in embedded systems. The evolution of processors from CISC to RISC to Very Long Instruction Word (VLIW) [78] shows a trend towards simpler (and therefore faster) hardware, but at the cost of greater code footprint and compiler complexity. In VLIW systems the compiler is given control over the run-time scheduling of the functional units of the processor. This scheduling information is encoded in instruction words that may be hundreds of bits long. This allows the hardware of the processor to become much simpler, as instruction decoding logic is reduced, hardware instruction scheduling units are not required, and techniques like out-of-order instruction execution require almost no additional hardware. Datapath architectures are the successors of VLIW processors, in that they compile applications for more general-purpose datapaths. VLIW concepts are used (instruction words are very long and complex compilers are required), but the target hardware units are no longer a fixed processor pipeline but a configurable set of units that can be adapted for different applications.

NISC [188] allows the designer to compile standard C code to *nanocodes* which are VLIW-style instruction words that control the operation of a generalised datapath to execute the programmer's input code. The compiler is passed an architecture description which describes the datapath layout. The programmer can add extra functional units at any time by amending the architecture description and recompiling, allowing for easy design-space exploration.

```
//Streams-C process declarations

/// PROCESS_FUN read_image_run
/// OUTPUT word_o
/// OUTPUT ImageDef_o

/// PROCESS_FUN controller_run
/// INPUT input_i
/// OUTPUT frame_o

/// PROCESS_FUN contrast_run
/// INPUT frame_i
/// OUTPUT remap_o

/// PROCESS_FUN remap_run
/// INPUT remap_i
/// OUTPUT output_o


//Streams-C process body declaration

/// PROCESS_FUN_BODY
SC_FLAG(tag);
SC_REG(frame_word, 32); //Declare stream registers
SC_REG(remap_word, 16);

int frame[256*256];

//Open the streams before the kernel starts
SC_STREAM_OPEN(frame_i);
SC_STREAM_OPEN(remap_0);

//While data is still present in the stream
while(SC_STREAM_EOS(frame_i) != SC_EOS) {

   //...Process the input data...

   //Data can be read into local memory like this
   SC_STREAM_READ(frame_i, frame_word, tag);
   frame[i] = SC_REG_GET_BITS_INT(frame_word, 0, 32);

   //When kernel is complete, close streams
   SC_STREAM_CLOSE(frame_i);
   SC_STREAM_CLOSE(remap_0);
}
/// PROCESS_FUN_END
```

Figure 2.8: Example of a process declaration in Streams-C.

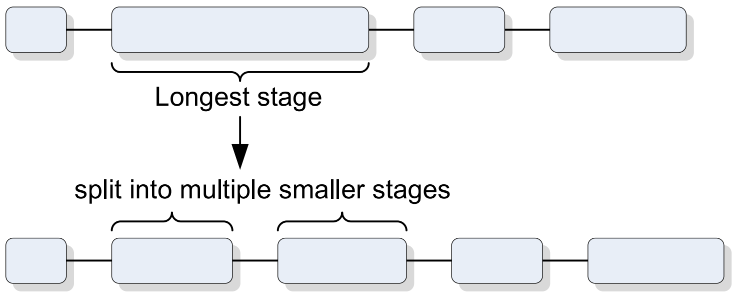

Figure 2.9: An example of pipeline balancing. The upper pipeline has a lower throughput because each stage must wait for the single long stage. Throughput is improved by splitting the long stage into smaller sections, at the cost of a small increase in latency and area.

The RICA [130] system targets a totally-connected grid of functional units of variable size. The functional units are taken from a standard VLIW processor and include arithmetic units, memory load and store units and standard register banks. At compile time, the compiler schedules multiple datapaths through this grid that start at either a RAM read or register bank, go through the required arithmetic units, and end at either a RAM store or register bank. These schedules are encoded into instructions and become the VLIW program of the RICA architecture. This is shown in figure 2.10. Because a new instruction is issued each clock cycle the result of this is that the RICA architecture behaves like a set of datapaths that can reconfigure on each clock cycle to perform the operations that are required by the original program.

RICA displays between 5 and 10 times the throughput of the ARM7 and the OpenRISC32 processors whilst consuming approximately the same amount of power. However, the large routing and interconnect requirements mean that it consumes 7.6 times more silicon area.

2.1.7 Architecture description languages

Systems that target changing, non-standard architectures must provide a facility to describe the target architecture. This might be done in a variety of ways, and one of the most common is the emerging area of Architecture Description Languages (ADLs).

ADLs [49] are a class of languages that are used to formally represent the architecture of a given system. They may be used to describe hardware implementation architectures, but most frequently ADLs are used describe the software architectures of large systems. The main use of ADLs are as a tool for cooperation between design teams. The ADL serves as a formal communication point when designing very large systems between the various hardware and software teams. The software teams write code to fit an architecture described by the ADL, whilst the hardware teams devise an implementation platform that will be able to efficiently execute the described system. ADLs also allow the analysis of languages at a very high level, allowing designers to reason about the performance and correctness of the system before development is complete.

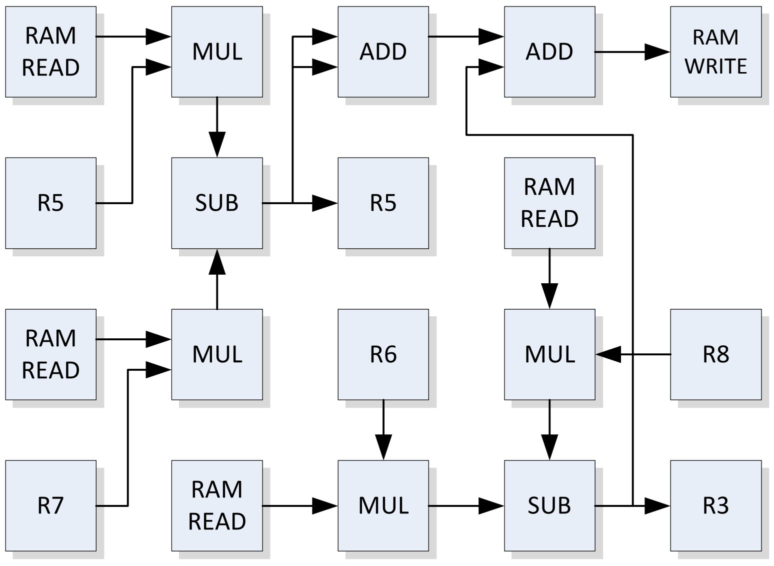

Figure 2.10: Example of a RICA grid during a single clock cycle. All data flows begin at either a RAM read or register, and end at a RAM write or register. The following clock cycle may have a totally different layout of data flows.

ADLs are currently a rather loose concept and there is no universal agreement on the abstractions that ADLs should have. Generally, *architecture* refers to the components that comprise the system, the behavioural specifications of those components, and the patterns and mechanisms that describe the interactions between them. In this way, they share many concepts with system design languages, that are discussed in section 2.4.3. Whilst there is no specific definition, it appears that an ADLs should display the majority of the following features [153, 198]:

- Abstraction of architectural components and their behaviour
- Representation of communications and interfaces, along with protocols and implementations
- Types and type checking
- Support for hierarchical refinement
- Ability to reason about time and causality
- Machine-readable - permitting the development of easy-to-use (and possibly graphical) analysis tools

ADLs appear to have fallen out of fashion recently and there are not many newer examples being developed. This is likely down to the dominance of UML [25] which often is used to perform a similar role and also that they have never been fully embraced by industry as a whole. Some examples of early ADLs are ACME [86], Rapide [153] and Wright [5]. Newer

ADLs appear to focus on highly dynamic architectures, such as π-ADL [176] which leverages the π-calculus and DAOP-ADL [182] which takes elements from aspect-oriented programming to describe dynamic run-time behaviour.

2.2 Other architecture-aware techniques

Section 2.1 talked about new languages or language extensions that were aimed at providing the programmer greater visibility of the target architecture. From this information, the programmer could either tailor their code to be more efficient, or control the implementation of their code to obtain a better mapping. There exist a number of techniques however that attempt to target future architectures through the addition of architectural information, but are not in themselves new languages. All of these lend weight to the argument that architectural information is an important consideration for the development of modern embedded systems.

2.2.1 OS services in dedicated hardware

The Multiprocessor System-On-Chip (MPSoC) paradigm is a common area of research because it can be particularly challenging to provide efficient Operating System support. On a normal symmetric multiprocessor (SMP) or multicore system, applications that need OS services can make a relatively efficient cross-core request. However, in an MPSoC there may be a huge number of cores which are all competing for OS services at any one time. Also, the use of POSIX services assumes that there exists a kernel that is compatible with the MPSoC's layout and the chosen processor type. Recent work [3] moves the core OS services out of a processor core and into dedicated hardware modules connected to the main system bus. The modules are:

- **Synchronization manager:** Synchronises threads, provides mutexes.
- **Conditional variables:** Provides POSIX condition variables.
- **Thread manager and thread scheduler:** Allows threads to fork and join with other threads, and manages the run queue.

The resulting architecture is shown in figure 2.11. The primary advantage of this approach is that there is not a single bottleneck for all OS services, it is now theoretically possible for one core to request a mutex lock at the same time as another core requesting a condition variable. Also, because the services are implemented in dedicated hardware, no processor time is lost to these services and they can be processed considerably faster than a general-purpose processor is capable of. The hardware modules allow a standard set of operating system services to be used in MPSoCs regardless of architecture, promoting coding standards and aiding software development.

The main disadvantage of the system, however, is that the modules must all sit on a single main system bus so the design as a whole experiences a single request bottleneck, although it

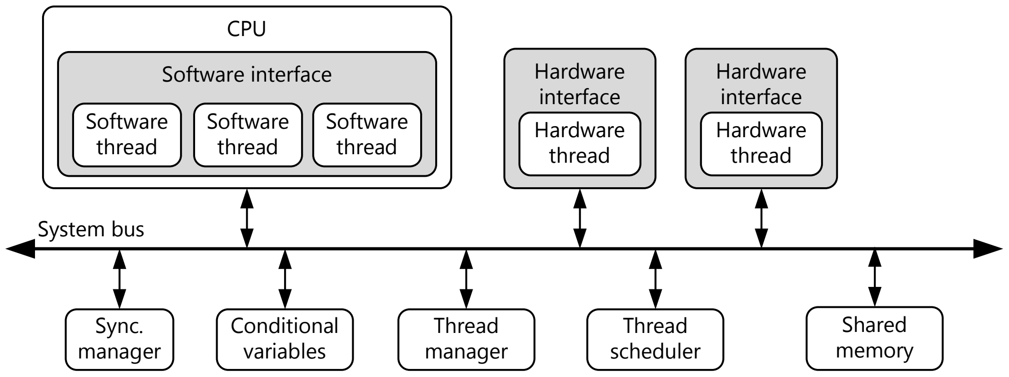

Figure 2.11: Moving operating system services into dedicated cores to improve efficiency and ensure they are always available.

has been ameliorated by the move to dedicated hardware. It is not clear whether such a technique can be extended to 'bridge' system buses and therefore allow truly parallel OS requests. Also, the size of the hardware modules is considerable and they can only be reasonably accommodated on the largest commercial FPGAs (see section 2.5.2) currently available.

2.2.2 Architecture-aware application mapping

A common theme in much recent work is that the standard approach of abstracting away from the target architecture is unsustainable. Providing the toolchain with a greater level of architectural information can greatly aid the process of mapping applications to non-regular systems, reducing inefficiency.

The system by Kim et. al. [134] explores the problem of mapping software applications to Coarse-Grained Reconfigurable Arrays (CGRAs) with NUMAs. The authors note that it is not enough to simply improve the computation speed - the local memory architecture must also be considered in order to achieve scalability. By providing the mapping system with extra information about the target CGRA, including the number of available memory banks, their location and available bandwidth, a more power efficient mapping can be achieved. The presented heuristic achieves up to 62% reduction in the energy-delay product.

2.3 Virtualisation and Virtual Platforms

Virtualisation is a general term for the abstraction of computing resources from their physical implementation. For standard desktop computing architectures the abstracted resources are frequently processing cores, memory spaces, communication resources, and I/O devices (network interfaces, keyboards etc.), but the term is much more general and can relate to almost any part of a given system. Virtualisation inserts a layer in the abstraction hierarchy of a system. The layer exposes a set of virtual resources that items at a higher abstraction levels can be implemented on top of. The mapping of the virtual resources to actual physical resources is hidden.

Virtualisation is used in a large variety of applications. It is most commonly understood to refer to software-based virtual machines which are discussed in section 2.3.1, but virtualisation also exists in the way operating systems are built (section 2.3.2) and is frequently used for verification and simulation of systems (section 2.3.4). Other techniques related to virtualisation are summarised in section 2.3.5.

2.3.1 Virtual machines

Virtual Machines (VMs) are the most well-known application of virtualisation and are commonly employed in all areas of computing, from high-end industry down to the home user. VMs became popular in the early 1970s as an alternative to system simulation [90]. Simulating a system involves the run-time interpretation of the entire ISA of a simulated processor as well as simulation of the memory and system buses. This results in execution times that are many orders of magnitude slower than native execution. Whilst this is required when simulating a new architecture or custom processor, it is frequently the case that the simulated system shares enough commonality with the host machine that some aspects could be executed natively, resulting in near real-time execution speed. The simulated machine becomes a virtual machine.

A VM is defined by Popek and Goldburg [184] as “an efficient, isolated duplicate of a real machine”. They also define a *Virtual Machine Monitor* (VMM), which is a piece of software running on the host machine that enables this virtualisation to take place. The VMM is defined to provide the following three features:

- **Equivalence:** The software executed by the VM should operate the same (expect in terms of execution speed) as if it were executing on a native machine. This requirement must still hold if the host is hosting multiple VMs and software of its own.
- **Resource control:** The VMM is in complete control over the virtual resources. In practical terms, this usually means running as a privileged process in the host operating system (hosted), or using a specialised virtualising operating system (native). Note that this requirement means that it is not possible to run a VMM inside a VM.
- **Efficiency:** Where possible the machine instructions of the virtual machine should execute natively on the host machine.

There are a large number of modern VMMs available, although they are more frequently known by the modern term *hypervisor*. There are two broad classifications of VMM that exist. *Native* VMMs are software systems that execute directly on the target hardware to provide virtualisation services to hosted operating systems. Pioneered with CP/CMS [56] in the late 1960s at IBM to provide time-sharing services on mainframe computers, modern examples include VMware ESX Server [232, 234], Xen [12, 48], and the L4 microkernel family [145]. *Hosted* VMMs are the second classification and are different to native VMMs because they run on a normal operating system and host VMs within its process. Most commercial products for end-users are of this type and include VMware Server [233], QEMU [21], Microsoft Virtual PC [53] and VirtualBox [213].

A normal hosted VM that is running an unmodified operating system (guest OS) will occasionally attempt to perform privileged operations that the host OS cannot allow. For example, paging and virtual memory operations must only be performed by the host OS and whenever a guest OS attempts to manipulate the memory management unit the operation must be *trapped* and replaced with a virtualisation-safe alternative. Similar problems are encountered when the guest OS attempts to access I/O devices (e.g. power management features). This trapping mechanism introduces overhead and requires dedicated hardware support from the processor and target hardware. An alternative to this called *paravirtualisation* was introduced with the Denali system [238] in which the VMM provides purely *virtual instructions* that do not have a direct counterpart in the target ISA. These instructions are conceptually similar to OS system calls that allow VMs to directly communicate with their VMM. These allow, for example, a hosted VM to sleep where previously the hosted OS would spin in a busy loop. Denali also sets aside dedicated memory areas for fast and efficient VM to VMM communication. The main disadvantage of paravirtualisation is that it requires the software of the hosted VM to be modified to include the virtual instructions as appropriate. This means the technique is not compatible with closed-source software unless a special version is produced by the software vendor.

2.3.2 Virtualisation in the OS

The previous section described one form of software-based virtualisation – the provision of an entire VM on top on an existing system. However, standard multiprogrammed operating systems also use a range of techniques that can be linked to the field of virtualisation. Essentially, they expose a VM to each currently executing process that displays a variety of desirable properties, rather than forcing developers to create hardware-specific versions of their applications. The following features are virtualised by modern OSs:

- To aid relocatable code and remove a dependency on memory layout, the memory space of a process starts at address 0 and is much larger than the available physical memory. The VM hides address translation, paging, and concerns regarding virtual memory.
- Shared resources (such as processor time, memory, I/O devices, OS services) are automatically shared between processes by the OS.
- Features such as coherent interprocess communications and atomic actions are provided.
- Hardware variations are hidden. For example, a 64-bit processor is exposed as 32-bit to a 32-bit process, yet a 64-bit process executing at the same time has full access to the 64-bit instruction set. The memory addresses of standard hardware features are homogenised though OS API calls.

2.3.3 Virtual machines for software languages

This section has so far discussed VMs as a way of sharing hardware between multiple software processes. However, it can equally be used as to aid the development of architecturally-neutral code that can be frequently reused and easily ported to new systems and architectures.

The most well-known example of this is the Java Virtual Machine (JVM) [147], which is at the heart of the Java language. The JVM provides a VM which shares the resources available to it (processor cores and memory spaces) amongst the threads of the system, but more importantly it also provides a single target for all Java developers regardless of the actual target hardware. If a standards-compliant JVM implementation exists for a given system, almost all existing Java code will run correctly on that system. Furthermore, it can be said that Java uses paravirtualisation-style techniques because Java code can call directly into the JVM, but it is arguably not a 'pure' VM because the language provides low-level system access through extension libraries which means that some code can be platform-specific.

A number of other very-high level languages make similar use of VM-based implementations. Smalltalk [189] is an object-oriented, reflective language that was created at Xerox PARC during the 1970s. It bears a number of conceptual similarities to Lisp in that it is almost entirely written in itself, and is organised using meta-level objects that represent most parts of the language, including the classes, methods, compiler, interpreter and even stack frames. Together, the Smalltalk runtime (an interpreter and just-in-time compiler) presents a VM model to the programmer's code. Smalltalk has one of the most complete sets of reflective facilities of any language in widespread use. For example, it is possible for an object to examine its own run-time stack or to refine the methods that it implements at run-time. Code blocks can be passed between objects and their contents examined and altered before being executed. Smalltalk classes frequently cooperate to construct other classes, and therefore implement a metaobject protocol [133, 132, 131, 157, 46]. Figure 2.12 shows an example of one of the ways in which Smalltalk implements introspection and intercession. The expressive power afforded to the programmer by Smalltalk necessitates its implementation using a virtual machine because it is not possible to build and modify the source code of a running program without virtualisation, interpretation, or complex just-in-time compilation.

2.3.4 Virtual platforms for verification

Virtualisation is commonly used to assist designers in the verification and validation of their systems. Modern MPSoCs may contain dozens of interacting modules and complex bus topologies that make the actual overall performance of the system almost impossible to determine simply through code inspection and offline analysis. Frequently with such systems, the performance of an individual core can be sufficiently estimated, but system bottlenecks appear as an emergent property of many cores and may be chaotic and transient in nature. Therefore, for all but the most safety-critical systems, a form of testing is preferred over analysis. Standard testing, where the system is implemented and its performance measured is problematic for two main reasons. Firstly, many systems are costly to implement because they require the creation of a custom ASIC (see section 2.5.1). FPGA technology (section 2.5.2) can help reduce this cost, but FPGAs are lower in speed than ASICs which might force a slightly different behaviour, limiting the value of the testing. Secondly, whilst the inputs to the test hardware can be controlled and the outputs observed, it it impossible to determine exactly

```
"Declare 4 local variables for use."
| x y className methodName |

"Set className and methodName to string values."
className := 'MyClass'.
methodName := 'aMethod'.

"Evaluate className as code, rather than a string."
"x is set to the result of the evaluation."
x := Compiler evaluate: className.

"Check that the evaluation succeeded by asking if x"
"is a subclass of the class Class. Class is the ancestor"
"of all classes in Smalltalk and a descendant of the"
"Object class, the root of the hierarchy."
(x isKindOf: Class) ifTrue:
[
   "The evaluation was successful so create a new instance"
   "of MyClass. Save it in y."
   y := x new.

   "Ensure that y will respond to the method 'aMethod'."
   (y respondsTo: methodName asSymbol) ifTrue:
   [
      "It will, so execute it."
      y perform: methodName asSymbol
   ]
]
```

Figure 2.12: An example of Smalltalk's reflective capabilities. Smalltalk comments are enclosed in double quotation marks ("").

what is happening inside the design. It is possible to observe that the design can sustain a given data throughput, but no insight is obtained as to why.

One solution to this problem is to simulate the design rather than implement it. The effect of the code is still observed rather than reasoned about, but complete visibility of the internal state of the system can be obtained. Of course, simulation suffers from its own problems, primarily the amount of computation time it takes, poor availability of cycle-accurate component models, and the enormous volume of data it can produce.

The CoWare Virtual Platform [54, 230] attempts to mitigate some of these problems through the use of virtualisation. CoWare creates a *virtual platform*, which is an executable simulation of the device hardware and the environment in which it is operating. This is different to standard simulation because, as with other modern virtualisation systems, parts of the target code can execute natively on the host processor. This means that execution speed can be orders of magnitude faster than pure simulation. The CoWare system allows the designer to inspect the state of the simulation, set breakpoints, watch memory and apply stimulus to the system, all with the intent of giving the designer a greater understanding of how their system is operating as a complete design. The main limitation of the CoWare system is that, due to its goal of native execution, only processors that have a broadly compatible ISA can be virtualised efficiently and the designer can only use processors for which there exists a CoWare model. Unique application-specific processors or components cannot be virtualised and the designer must instead rely on standard simulation.

2.3.5 Other virtualisation-based techniques

Processor virtualisation can be used to divide a physical processor's time among a set of virtual machines, as discussed in section 2.3.1. Whilst this section discussed such systems for use in servers and desktop computers, similar techniques can also be used in the embedded domain, where its main goal is to provide temporal isolation for parts of the system. For example, one aspect of a system might be timing-critical and so is developed and verified to a high standard. Virtualisation can then be used to ensure that non-timing-critical tasks in the rest of the system do not compromise the performance of the timing-critical code. This requires the virtualisation system to be *composable* [168]. The main advantage of such a system is that it requires only the timing-critical parts of the system to be analysed for correctness, the rest of the system can be implemented as normal and any erroneous behaviour will be contained. This assumes that the virtualisation layer and compilers are correct and trusted.

2.4 Generating hardware from software-style descriptions

The previous sections in this chapter have discussed the ways in which software language research has attempted to find better ways for the programmer to reason about the implementation environment of their code. This may concern the mapping of the software system to the hardware, as was primarily discussed in section 2.1, or the ways in which virtualisation can be leveraged to hide unimportant details and make this mapping simpler, as in section 2.3. An alternative approach to this mapping problem is to actually generate the hardware directly from software descriptions. This can range from low-level hardware description languages (section 2.4.1) through system design languages (section 2.4.3) to high-level synthesis systems (section 2.4.2) which take as input a program of similar abstraction level to that of normal software languages.

2.4.1 Hardware description languages

When the first embedded systems were being developed, the available fabrication technologies meant that only relatively small circuits could be built. Designs were presented in the form of a schematic diagram showing the circuit as a design composed of logic gates or transistors, which could then be fed into CAD tools to create silicon masks directly. Early microprocessors were complex, but it was still tractable for the designers to work with schematics to describe their entire design. Designing with schematics meant that the hardware designers had complete control over every flip-flop and logic gate in the system, which was important because even a slight sub-optimality was expensive.

The main problem with this approach was that the lack of abstraction meant schematics became very difficult to work with when design sizes began to increase. Since the development of the integrated circuit in the early 1960s and consequently the microprocessor in around 1970, the logic density and effective computational power of digital circuits has increased massively. The gate count of modern processors is measured in the hundreds of millions, making a complete schematic diagram unmanageably large. To remedy this, Hardware Description Languages (HDLs) were developed.

HDLs are a software-based technique for describing the arrangement of elements in a digital circuit. They describe the circuit's operation and organisation and can frequently also be used to simulate the described circuit and thereby verify the design. HDLs are a higher level of abstraction than schematics as they allow the designer to partition a design into a logical hierarchy of instantiated components, whereas a schematic is single-level and flat.

One of the first HDLs, ABEL [254], was devised in 1983 for targeting programmable logic devices but it was not until the late 1980s that modern HDLs were developed. Modern HDLs provide a higher level of abstraction than simple netlist formats like EDIF [69] because they allow for large amounts of circuitry to be designed very rapidly using source code that shares some abstractions with traditional programming languages. The language might include a notion of time or state, common arithmetic operators and program flow statements like loops and condition tests. A tool called a *synthesis engine* is then used to convert these statements into a netlist-based logic description for implementation. This process, known as *synthesis*, is analogous to compilation of software programs to relocatable object code. As shown in figure 2.13, these developments have led towards the unification of software and hardware design flows into what has become known as *hardware / software co-design* (section 2.4.4).

The two most common HDLs that are in use today are VHDL [43] and Verilog [64]. They both provide a rich expressive environment for designing digital circuits and are widely supported by industry-level toolsets. VHDL (VHSIC Hardware Description Language) was initially developed as a way of documenting the behaviour of ASICs, but has now grown into a tool for designing, simulating and synthesising them also. As a result only a subset of VHDL can be directly synthesised to hardware, the rest of the language can only be simulated. VHDL was designed to be similar in style to the Ada programming language [15] and so it shares a very similar syntax, is case insensitive and is strongly-typed. Verilog, in contrast, was designed to be

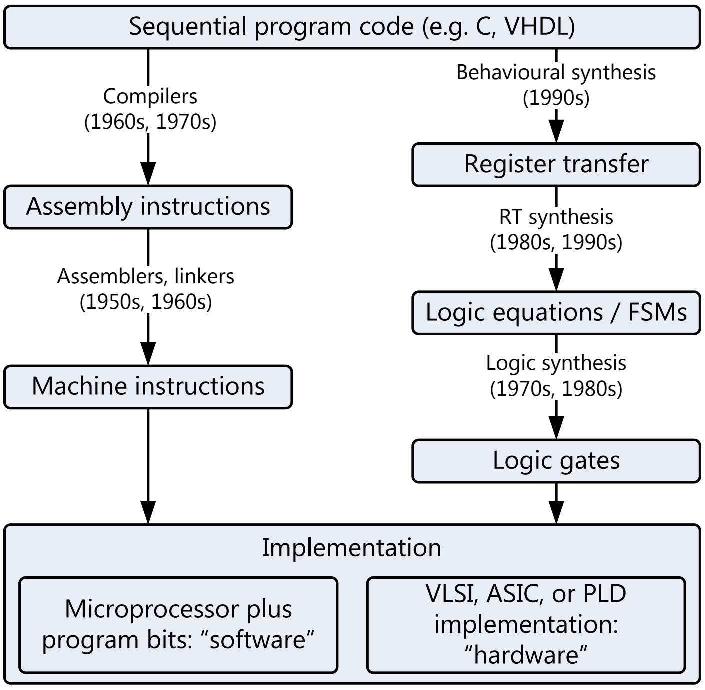

Figure 2.13: The co-design ladder: recent maturation of synthesis enables a unified view of hardware and software. Figure from [228]

familiar to C [129] programmers so it is case sensitive and uses a preprocessor. As with VHDL, only a subset of Verilog is synthesisable.

Both languages differ fundamentally from their procedural language counterparts, however, because Ada and C are both imperative languages whereas VHDL and Verilog are declarative. Whilst in a standard programming language two consecutive statements are executed one after the other, in a HDL they are generally executed in parallel because each statement describes a hardware element that is operational at all times. The designer must implement state machines and other techniques in order to achieve sequential execution.

The main problem with using HDLs is that of simulation. Once a design has been generated the designer needs to verify that it is functionally correct before it is fabricated. Fabrication is a very expensive process (see section 2.5.1) so it is essential that mistakes are not made in the final mask. Verification of HDL-based designs can be performed through formal analysis of the code [68, 224] but this is a highly-skilled technique that is both time-consuming and expensive. Frequently time-to-market constraints instead force the company to verify using functional simulation. Simulation is performed by compiling the HDL using a simulating compiler (for example ModelSim [164]) and then applying a series of stimuli to the input ports of the design and observing the effect on the output ports. Clearly this suffers from problems of test coverage, and because it is a computationally-expensive operation it is frequently very slow. FPGAs (section 2.5.2) have helped with this problem by allowing rapid prototyping and therefore reducing the overhead of simulation, but they introduce new problems of visibility. It is not possible to single-step through an FPGA-based design applying arbitrary inputs and to observe the internal state of the circuit as it is with a simulator.

2.4.2 High-level synthesis languages

Whilst HDLs greatly aid the design of hardware, newer design trends are beginning to strain current techniques. 50 million gate ASICs are commonplace today, yet the design and verification tools used to design these systems remain similar to those developed a decade ago. There is a disparity between the capability of modern fabrication technologies and the designer's ability to create and validate designs to make use of them [250]. Current HDLs force the designer to express the functionality of their system in terms of state machines and interacting parallel processes communicating only by bit vectors. Any higher-level software features like function calls or abstract data-types are non-synthesisable. For larger designs this makes development challenging and is a barrier to code maintainability and reuse.

As a result, there has been a great deal of research into the field of high-level language synthesis. The aim of such research is to increase the abstraction level of synthesisable HDL code, thereby making it simpler for the designer to create large systems without having to consider low-level implementation issues. Such high-level languages attempt to provide more expressive power than is afforded to the designer by standard HDLs. For example, this may include support for compound data types, procedures and functions, object-orientation, polymorphism, etc. A small amount of expressive code can be used to generate a large and complex circuit through the use of a verified compiler and synthesis tool. The high abstraction level allows the designer to concentrate on system-wide architectural concerns rather than gate-level concerns, potentially resulting in a better design. Also, when higher-level descriptions are used simulation becomes much more efficient because the programmer-provided model of the target circuit is smaller.

Synthesis techniques vary between languages but they all must perform the following main tasks [150]:

- **Resource allocation:** Determine the resources available to the synthesis system (adders, registers, reconfigurable area, buses, etc.)
- **Scheduling:** Determine the order in which each operation in the input design is executed, often this step can include register stages to balance pipelines and must balance the throughput of the design against resource usage.
- **Binding:** Map features of the input design to the resources of the implementation fabric. For example, variables will be mapped to registers and storage elements.
- **Control synthesis:** Generate the control logic that marshals the flow of data through the design so that it operates correctly.

Many of these stages also involve optimisation stages that are not shown. For example, before the synthesis runs an optimiser will commonly rewrite parts of the input code to make better use of the chosen implementation fabric, combine similar operations, or remove unnecessary ones.

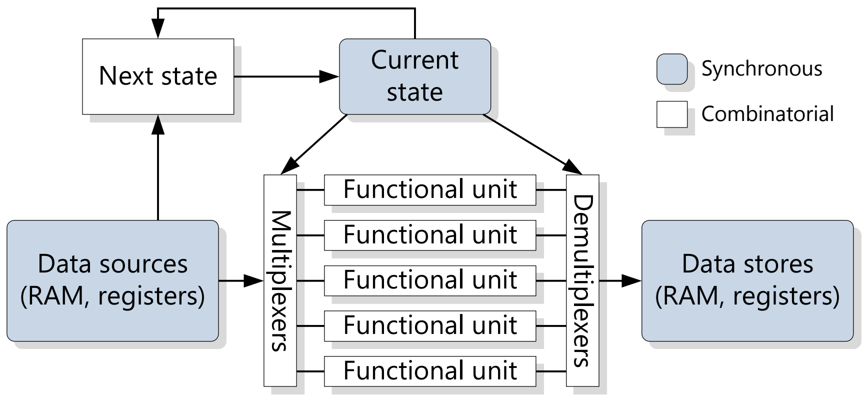

Figure 2.14: State machine-style high-level synthesis.

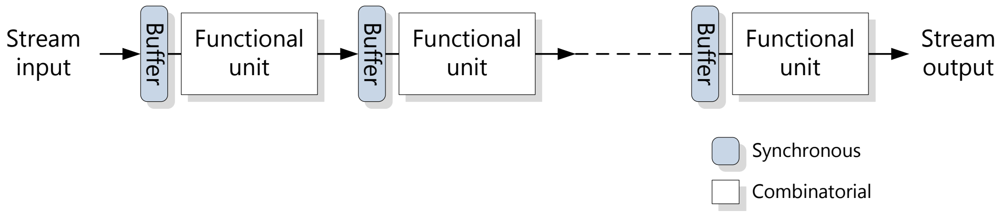

Figure 2.15: Datapath-style high-level synthesis.

There are two main implementation schemes that are employed by high-level synthesis systems, state machine and datapath. State machine-based synthesis builds a *one-hot* state machine, which is a synchronous circuit that exists in a given state and transitions to another state each clock cycle. The state transitioned to can be either dependent upon the passage of time (clock cycles), the value of internal registers, or the value of external inputs. The current state signal is used to schedule and control the flow of data through the rest of the circuit, which comprises standard functional units such as registers, adders, multiplexers etc. This implementation strategy is particularly good at describing complex control logic which can be encoded in the 'next state' circuitry, but it can suffer from low clock speeds because it tends to generate long combinatorial paths. This synthesis style is shown in figure 2.14.

In contrast, datapath-based synthesis [209] removes the state machine logic and instead chains together a series of functional units to form a pipeline where data enters at one end, flowing through each unit until it reaches the end. This style requires almost no control logic, but it is much less flexible and does not describe complex changes in control flow very well. More complex schemes employ a form of Very-Long Instruction Word (VLIW) [78] coding to schedule the operation of the functional units and therefore provide greater flexibility. Two such systems are discussed in depth in section 2.1.6 and many other examples exist [165, 101, 113]. Datapath synthesis tends to produce shorter combinatorial paths, and is excellent for generating circuits to process large volumes of data, although its inflexibility may force the otherwise unnecessary duplication of functional units. This style is illustrated in figure 2.15.

Kahn Process Networks (KPNs) [123] are frequently used as a computation model for datapath synthesis-based systems. KPNs model computation as a group of deterministic sequential processes that communicate with either other though FIFO channels. This very closely models hardware pipelines and data paths, so they have been used in systems such as Daedalus [223] and Sesame [74] to generate hardware descriptions from C code.

Handel-C

Handel-C [26] is a well-known high-level synthesis language developed by Celoxica as a successor to LOLA [80] for rapid prototyping of hardware designs to FPGAs and ASICs. It is implemented as a subset of ANSI C with a number of extensions that allow for parallelism and communication between parallel blocks. All state machine and control flow hardware is inferred and generated automatically by the Handel-C synthesiser. This means that an off-the-shelf solution that is written in ANSI C can often be synthesised directly to hardware, once wrapped in a hardware interface.

The process of synthesising Handel-C to hardware is very similar to the way in which normal C is compiled to object code. The compilation strategy is basically recursive descent, but rather than recursively generating machine code, the synthesiser generates small hardware blocks which are recursively connected together. The resulting circuit is expressed in the EDIF format which can then be optimised by any number of tools before the design is converted to an FPGA bitfile by vendor-specific toolchains.

The Handel-C design process is fundamentally different to the way in which a circuit is built up with a hardware description language such as VHDL. As mentioned previously, VHDL is a declarative language whereas Handel-C (like its parent language C) is imperative. As a result, to describe two events that occur sequentially can be difficult in VHDL as a one-hot state machine must be described that switches between the two events in the correct order.

```
process Do_The_Tasks(clk, current_state)
begin
   if (clk'event and clk = '1') then
      if (current_state = 1) then
         Do_Task_One;
      elsif (current_state = 2) then
         Do_Task_Two;
      end if;
   end if;
end process;

process Change_State(clk)
begin
   if (clk'event and clk = '1') then
      if(current_state = 1 and Task_One_Finished) then
         current_state <= 2;
   end if;
end process;
```

In Handel-C this is much simpler because state machines are automatically inferred by the synthesiser and the inherently sequential nature of C can be exploited by simply calling the two items one after the other.

```
void main(void)
{
   Do_Task_One;
   Do_Task_Two;
}
```

Like most standard programming languages, normal Handel-C designs can only include a single clock source. The reason for this is that Handel-C was designed to have a very predictable timing model and the provision of multiple clock sources would undermine this aim. It is possible to make use of multiple clock sources in a Handel-C project but this is achieved by creating multiple designs, each with its own `void main(void)` function, and then linking them together using the Handel-C IDE. The designs can then export and import communication channels to share data asynchronously. This approach is quite limited because the encapsulation of design units is performed outside of the language and it gives no scope for modular composition, meaning that designs cannot be nested. Also, only channels can be shared between designs; variables, signals, interfaces and functions cannot. The number of clock sources is limited to the number of clock nets available on the FPGA.

There are a number of problems with Handel-C that limit its potential for exclusive use in the embedded systems market. First, Handel-C has a very simple timing model that is intended to produce circuits with easily predictable performance characteristics. Essentially, the model states that "every assignment and communications statement takes one clock cycle, everything else has no cost" [235]. This means that all assignments will take the same amount of time and so it is easy to inadvertently reduce the maximum speed of the entire circuit by introducing a single statement with a large propagation delay. To avoid this, the programmer must split large sections of combinatorial logic into multiple stages. This splitting cannot be done by the compiler as it would violate the semantics of the program.

Related to this problem is the fact that even though Handel-C appears to be standard C, it is in fact targeted at a very different implementation. Consequentially, a programmer who forgets this and writes in the style that they would for normal C will end up with an extremely inefficient design. This can be seen when comparing loop termination constraints in the two languages. In C it is common to compose a loop as such: `for(x=0; x<10; x++)`. However it is more efficient in Handel-C to replace the less-than comparator with an inequality check: `for(x=0; x!=10; x++)`. The inequality check is a simpler circuit than the comparator that is required by the less-than operator and so it results in a smaller design. Finally, due to its reliance on standard C, Handel-C lacks real modular decomposition or encapsulation making it difficult to use when producing larger designs and limiting the language's potential for code reuse. As a result of all these problems, Handel-C is now positioned as a language for synthesising function accelerators and custom logic cores rather than entire systems.

Lava

Most high-level synthesis languages tend to be imperative, based on a language like C or Ada. Lava [23] is noteworthy because it is based on Haskell [121], meaning that it has functional semantics instead. Functional semantics are useful for hardware description because they carry strong compositional information. In a functional language each function is expressed as a composition of smaller functions, in exactly the same way that circuits are built from a composition of smaller circuits. This structure greatly facilitates code reuse and is perfect for describing certain types of circuit, primarily signal processing designs. As a result, Lava makes use of a data path implementation style.

Lava is primarily used as a platform for analysing hardware and formally verifying designs. System components (written in standard Haskell) can have many different *interpretations*. The default interpretation synthesises the function to a netlist, but more complex interpretations allow higher-level understanding, verification, simulation, and reasoning about non-functional properties. Also, because Lava uses Haskell as an underlying language, the system can handle symbolic data and expressions rather than purely concrete ones that would be required by a C-based language.

For example, the following code describes a half adder in Lava:

```
halfAdder (a, b) = (sum, carry)
  where
    sum = xor2 (a, b)
    carry = and2 (a, b)
```

This circuit has two input wires (`a` and `b`) and two output wires (`sum` and `carry`). This implementation uses two logic gates, an `and` gate and an `xor` gate.

Two half adders can then be combined into a full adder as follows:

```
fullAdder :: (Bit,(Bit,Bit)) -> (Bit,Bit)
fullAdder (carryIn, (a,b)) = (sum, carryOut)
  where
    (s1, c1) = halfAdder (a, b)
    (sum, c2) = halfAdder (carryIn,s1)
    carryOut = xor2 (c2, c1)
```

Lava essentially embeds VHDL-style code in low-order functions and builds a type system on top of this to ensure consistency. Higher-order functions can then succinctly describe regularity that would ordinarily involve repetitive or complex VHDL code. A good example of this is the way that a ripple carry adder can be constructed from full adders.

```
rippleCarryAdder (carryIn, (as,bs)) = (sum,carryOut)
  where
    (sum,carryOut) = row fullAdder (carryIn, zipp (as,bs))
```

In this description, `row` is a *connection pattern* that chains full adders together, one for each bit of the input vector, and `zipp` combines the output bits into a single output vector.

The main limitations of Lava come from its use of Haskell. Due to its functional nature it can be difficult to reason about precise timing properties of a circuit, and I/O is very difficult to handle in a clean way. As a result, creating large Lava-based systems that meet timing closure can be challenging.

Other high-level synthesis languages

Aside from the languages already presented, there are a huge number of languages that can be targeted at hardware implementations rather than software ones.

The York Hardware Ada Compiler (YHAC) [235] has shown success in retargeting the Ada language to generate hardware designs in the form of EDIF netlists. (Ada is a very large language so only the Ravenscar subset [30] is implemented.) YHAC translates sequential Ada into hardware by generating one-hot state machines for each procedure. Unlike Handel-C, these state machines allow for individual operations to take multiple clock cycles, thereby reducing propagation delay in the final design and maximising potential clock speed. However, the inefficiency of a one-hot implementation style means that circuit scalability can be reduced and sizable programs can often be translated into very large circuits. YHAC can make use of Ada's native support for concurrency and will create a new state machine for each task, giving the programmer access to true parallelism. Therefore, YHAC implements Ada's coarse-grained concurrency model instead of the fine-grained model found in Handel-C.

NENYA [33] translates Java bytecodes into a dedicated datapath circuit that can be implemented on hardware that supports partial dynamic reconfiguration (see section 2.5.6). The technique uses temporal partitioning, which separates bytecodes into graphs that do not need to occupy the device at the same time. Then, the system attempts to generate separate data path circuits for each temporal partition that then can be executed by a reconfigurable framework which implements the virtual hardware paradigm (discussed in [148] and [79]). The system extracts control dependency graphs, data dependency graphs and data flow graphs from the bytecodes and uses them to create the partitions. Problems with this approach are centred around its use of Java bytecodes as an input source. The translation of sequential byte codes to hardware can be rather inefficient as it results in circuits that are influenced more by the design of the Java virtual machine than by the source code from which they were generated. This can be compared with NISC (section 2.1.6), which tends to be more effective because it is based on x86 opcodes, but does not consider temporal partitioning.

SPARK [100] is another C to VHDL high-level synthesis system that distinguishes itself from similar systems by attempting to combine the features of a high-level synthesis engine with that of a parallelising compiler to obtain an efficient hardware implementation without the need to extend the input C code (as is required by Handel-C, etc.). For the same reasons as Handel-C, SPARK cannot efficiently synthesise large software systems so it is particularly focussed towards the development of co-processors and signal processing applications. SPARK uses a similar finite state machine model to Handel-C, but it employs a large number of optimisations to the code first in an attempt to extract the highest level of instruction-level parallelism possible. After the optimisations a scheduling and allocation phase maps the software operations onto hardware constructs, and then a code generation phase outputs VHDL for synthesis using standard tools.

The input to SPARK is standard ANSI C, but with the restrictions of no pointers, no function recursion, and no irregular control-flow jumps. Handel-C allows the use of limited pointers, but also prevents recursion and arbitrary jumps because they are highly challenging to implement in hardware. Arbitrary pointers and jumps are difficult to implement because of the static nature of hardware. Consider the following program:

```
void myfunction(void * mypointer) {
   int x;
   x = *(int *)pointer;
}
```

In a software implementation, the `mypointer` is referencing a value in memory, so the program can deference `mypointer` at run-time and access the location to which it is pointing. The cast and dereference translates to only a single opcode in most instruction sets. However, in a hardware implementation `mypointer` is referring to another functional unit elsewhere in the system. The program does not constrain in any way which part of the system this pointer may be referencing, so in the worst case `x` must have a connection from all other units in the entire system, leading to an infeasible amount of routing.

SPARK achieves good performance overall and the optimisations it applies result in an up to 68% reduction in circuit size, but it is restricted by the inherent limitations of C. All systems that attempt to automatically parallelise C can extract some useful parallelism, especially from simple loops that can be unrolled into vector operations, but they cannot extract system and component-level parallelism because C does not allow the programmer to express these concepts. It is for this reason the SPARK limits itself to synthesising a single component rather than the entire system. Also, because of its finite state machine-based implementation strategy it suffers from the same problem as Handel-C in that each extra line of code increases the amount of routing and multiplexing in the final circuit leading to a greater than liner increase in circuit size.

Finally, Catapult C [165] is a synthesis system that is becoming increasingly popular in industry which converts C++ programs to RTL designs. Like SPARK, it requires no language extensions or annotations in the actual input code. However, where SPARK uses extensive optimisations to extract parallelism automatically, Catapult C relies on extensive user input throughout the execution of the toolchain to provide circuit scheduling information. When using Catapult C the user places constraints on various non-functional properties of the design to influence the behaviour of the synthesis tool. For example, the user can set the desired clock speed and Catapult C will insert pipeline stages appropriately in order to ensure that the requested speed can be met. To exploit parallel execution, the user can request loops in the input code to be unrolled, either partially or fully. Catapult C's main limitation is that it is heavily focussed towards the synthesis of datapath designs and it preferentially generates pipelines. The synthesis of state machine-based control logic is still immature. The user is required to use a special class called `Control` which has different implementation semantics to the rest of the class hierarchy. Also, like most other high-level synthesis systems, it is considerably better when synthesising relatively small systems because hardware cannot be easily shared and reused so the final design size grows considerably as the input program lengthens.

2.4.3 System design languages

HDLs tend to lead to a bottom-up design style. The designer uses low-level constructs to build component libraries, and then instantiates those components to create a system. In a rigourous design flow, the designer will testbench each newly-created component to verify that they are working correctly. Then, once the entire library is complete and the top-level system design built, integration tests are performed to ensure that the components function together as intended. It is not as easy to simulate a design top-down, however, because VHDL and Verilog do not allow high-level descriptions of black box components. It is possible to use non-synthesisable features of the languages to achieve a slightly higher abstraction level whilst simulating, but in general the entire design must be fully-reified before simulation can take place.

In contrast, *system design languages* attempt to focus on a more top-down approach that allows designers to give ambiguous, unimplementable high-level descriptions of components which can be very quickly simulated together as a complete system. Then, once these initial simulations are complete the designer can begin iteratively refining the components of the design until they are sufficiently specific enough to be implemented. The advantage of this approach is that it is usually during integration testing that system bottlenecks become apparent. By performing such testing early it is possible to get initial estimates of performance and resource usage that can help guide the entire design process.

It should be noted that system design languages are orthogonal to high-level synthesis languages because their implementable subset is generally equivalent to that of standard HDLs. They could be combined with high-level synthesis to increase their abstraction level, but this is generally not done.

SystemC

SystemC [116] is a system design language that has gained a large amount of support in industry. An IEEE standard, SystemC was primarily developed to allow the design and simulation of systems at multiple levels of abstraction. It is based on C++ [72] and implemented as a library of classes and macros which allows it to keep the same syntax as standard C++. An unmodified C++ compiler can compile a SystemC simulation model. The SystemC language can be viewed as both a HDL and a simulation language because whilst its main aim is to verify the design of a system, a subset of the language can be synthesised directly to hardware. However, the synthesisable subset of SystemC is equivalent in terms of expressive power to that of VHDL or Verilog [95] so it does not offer the extra layer of abstraction in the final hardware description that high-level synthesis languages provide (section 2.4.2).

SystemC models concurrent hardware units as *modules* that can communicate and exchange data through *ports*. Inside modules, *processes* are the main unit of computation and are simulated concurrently. Processes pass data and synchronise using *channels*. A library of built-in primitive types are supported, or the user can define their own composite types. SystemC encourages layered simulation, where higher layers are less accurate but very fast to evaluate and lower levels approach cycle-accurate simulation but take correspondingly longer to run. SystemC also allows for layers to be mixed - some parts of the system are simulated cycle accurate and some at a higher-level. Example layers are as follows:

- **Layer 3 - Message Layer:** Entirely untimed simulation where all communication is assumed to be point-to-point, so therefore bus arbitration is not considered. Essentially the only operations supported are *send message* and *receive message*. Simulations at this level are inaccurate but very fast and are used to obtain rough estimations about the functionality of the system and to begin to see where the system's main bottlenecks are likely to be.
- **Layer 2 - Transaction Layer:** This level begins to take the implementation architecture into account so busses are simulated and some approximate communication timing is considered. For example, the bus can be modelled as a SystemC module and written such that communications are serialised with appropriate priority and scheduling mechanisms applied. Transmission time may be modelled as a simple function of message length.

- **Layer 1 - Transfer Layer:** At this level the simulation contains all the code that will execute on the target platform so detailed software simulation can be performed, but cycle-accurate timing cannot be obtained because the hardware is not fully modelled.
- **Layer 0 - Register Transfer Level:** This level is the lowest level of abstraction and includes a full model of the target hardware and the complete system software. It is cycle-accurate, but also the most computationally-demanding. This is equivalent to the simulation of RTL-level VHDL.

Layers such as these allow designers to very quickly create a complete system model that can still be informally tested for functional correctness. Then, as the design is reified the simulation becomes more accurate (but slower to execute) and true verification can start to be performed. At the lowest levels of reification, the simulation is equivalent in accuracy and speed to a full HDL simulation. Many projects have used SystemC for transaction-level modelling of various systems [32, 191, 177].

A SystemC program compiles to a standard software executable, which when executed performs the simulation of the system being developed. If a hardware implementation is required then a specialist translation tool is required to turn the SystemC code into a HDL or netlist format. This is in contrast to HDLs which primarily describe hardware and are simulated by a dedicated simulation engine.

Most of the criticisms of SystemC are related to its base language, C++. As C++ is a sequential language, describing concurrency and timing properties requires the use of preprocessor macros and library calls that can seem counter-intuitive, whereas in other HDLs these constructs are first-class parts of the language. This can be seen in figure 2.16, a SystemC description of a NAND gate. Many preprocessor macros are used (`SC_MODULE`, `SC_CTOR`...) in order to implement syntax that is not available in normal C++. Also, due to fact that the SystemC macros are attempting to give C++ declarative semantics (like VHDL or Verilog) it can be difficult to separate code that actually describes hardware from code that exists solely to aid the simulator. Figure 2.17 shows an equivalent VHDL description for comparison.

Also, as noted by in [67], it is easy to inadvertently develop a non-deterministic SystemC model due to the fact that the simulator must mimic hardware concurrency on a sequential processor. This leads to a slightly different execution order each time the simulation is run resulting in race conditions and different simulation results for each run. These problems are very difficult to detect from the simulation alone and may persist into the final hardware design.

Other system design languages

Whilst SystemC is probably the most well-known system design language there are a number of similar languages that are frequently used. SpecC [84] is a superset of ANSI-C which was developed to fulfil the same goal as SystemC and so it supports the same top-down transaction-based modelling approach. Whereas SystemC models structure and behaviour separately (with modules and processes respectively), SpecC consolidates these together with a single *behaviour* construct. As a result, SpecC supports behavioural hierarchies which are harder to model in SystemC. Also, SpecC provides slightly more control over the scheduling of the simulation's execution sequence by adding support for static scheduling and explicit

```
#include "systemc.h"

SC_MODULE(nand2)     // declare nand2 sc_module
{
   sc_in<bool> A, B; // input signal ports
   sc_out<bool> X;   // output signal ports

   void the_nand2()   // a C++ function
   {
      X.write( !(A.read() && B.read()) );
   }

   SC_CTOR(nand2)    // constructor for nand2
   {
      SC_METHOD(the_nand2); // register do_nand2 with kernel
      sensitive << A << B; // sensitivity list
   }
};
```

Figure 2.16: A SystemC specification of a two-input NAND gate.

```
use ieee.std_logic_1164.all;

entity nand2 is port
(
   a,b: in std_ulogic;
   x: out std_ulogic
);
end nand2;

architecture struct of nand2 is
begin

   process (a,b)
   begin
      x <= a nand b;
   end process;

end struct;
```

Figure 2.17: A VHDL specification of a two-input NAND gate.

Metric	Hardware	Software
Throughput	high	low
Power usage [1]	low	high
Silicon area [2]	high	low
Design time	high	low
Maintainability	hard	easy

Figure 2.18: The tradeoffs that hardware / software co-design attempts to balance. [1]For small functions, an ASIC will use less power than a processor. However, the power usage of a processor is capped and extra functionality can be added by placing more code in memory for only a very small increase in power consumption. [2]As above. A processor is a one-off logic cost beyond which extra functionality is cheap. New functionality in an ASIC design is costly.

description of state machines. SystemC instead must rely on dynamic scheduling which can lead to the problem of non-deterministic simulations in some cases, as previously discussed.

SystemVerilog [214] is high-level abstraction of Verilog which provides similar modelling techniques to SystemC and SpecC. However, unlike these two languages its main focus remains gate-level synthesis. Whilst SystemC and SpecC are frequently used to verify systems only, SystemVerilog is primarily a HDL and so is almost always used to generate an actual hardware implementation.

2.4.4 Hardware / software co-design

Hardware / software co-design is an active field of research which is primarily motivated by the observation that, for most embedded systems, much of their functionality can be either implemented as dedicated hardware or as a software routine running on some form of embedded microcontroller. Both approaches carry inherent advantages and disadvantages (see figure 2.18) and co-design attempts to balance these to find a sufficient design that meets restrictions placed on various metrics such as execution time, hardware density, power consumption or build cost. Unlike system design languages discussed in the previous section, co-design attempts to perform this system partitioning automatically though the use of a co-design *framework* and a partitioning algorithm based on search.

In the classic description of co-design [246], the operation of a system is specified in an implementation-independent way along with a quantitative list of requirements that the final implementation must meet. The co-design engine then creates the hardware / software partition by assigning the various functions of the system to either hardware or software. This can be done using a variety of search methods such as simulated annealing [75] or Tabu search [71]. The hardware functions are synthesised to dedicated hardware and the software functions are compiled to opcodes for execution on a predetermined processor core. Once these processes are complete, the entire system can be evaluated by a co-simulation engine that returns a set of performance metrics. If these metrics show that the system does not meet its specification the process is repeated. This time, however, the hardware / software partition is moved and so the implementation method for some functions is changed. This new implementation can then be regenerated and reevaluated until a solution is found that meets the system's initial constraints.

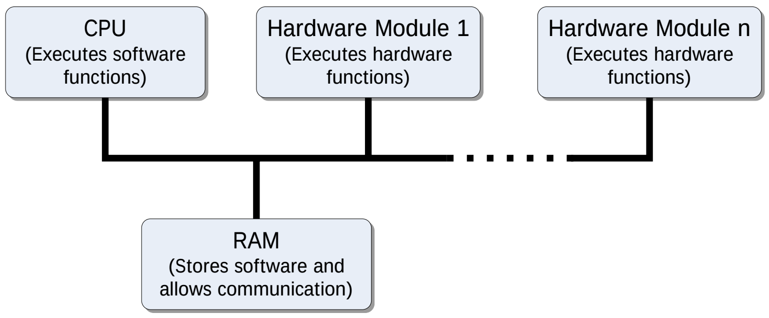
CPU
(Executes software functions)
Hardware Module 1
(Executes hardware functions)
Hardware Module n
(Executes hardware functions)
RAM
(Stores software and allows communication)

Figure 2.19: The target architecture of classical hardware / software co-design

The target architecture for such classical co-design systems involves a single shared system bus, upon which sits a single processor (to execute the software tasks), a number of custom hardware co-processors (to implement the hardware tasks) and a block of shared memory (see figure 2.19). More recent work has reduced this restriction; for example work by Niemann and Marwedel [172] describes a system that supports multi-processor architectures and Kalavade and Lee [124] examine the partitioning problem when applied to more general architectures.

Recent work [200] has looked at co-design from a software engineering point of view by extending the object hierarchy of the RTSJ to include new classes that encapsulate hardware and software implementations of threads. The object framework automatically marshals the communications between threads, allowing the final implementation choice to be changed transparently by the designer, even at a very late stage of development. However, the target architecture for this work is still a single embedded Java processor with a number of hardware accelerators connected to a common bus, and it does not provide integration with high-level synthesis engines for generating the hardware accelerators automatically.

Due to the lack of a truly implementation-independent language for expressing a system's functionality, all co-design frameworks tend to be either software-based, or hardware-based, depending upon the format in which the design is initially specified. In a system such as Cosyma [75], the entire design is described in a specially-created superset of C called C^X. C^X is still a software language, it simply augments C with a few necessary concepts, such as that of tasking. Therefore, initially a Cosyma design is entirely implemented in software. When the system is run, it uses a specified cost function to evaluate the design and then begins to automatically move parts of the code into dedicated hardware through the use of a synthesis engine. Conversely, the work by Gupta and De Micheli [99] is hardware-based because the initial system specification is a circuit design written in the language HardwareC, a language which adopts most of the semantics and syntax of C but is modified to allow unambiguous hardware modelling. In this work, only when the system cannot meet its stated requirements are sections of hardware moved to the software partition. This is achieved by translating from HardwareC to standard C using a code generation engine and then compiling the new code for a generic microprocessor. Despite their differing approaches, it appears most co-design systems achieve similar levels of overall success.

More modern work [239] attempts to avoid this dependence on either hardware or software by focusing on a single target domain (high-volume data processing, e.g. video) and restricting the flexibility of their task model. Applications are represented as directed acyclic graphs of tasks using a synchronous dataflow model [141]. Inter-task communication is strictly limited to use only pipeline buffers. Tasks still have to be written in C/C++ for software or Verilog for hardware, but the system model is implementation-agnostic, allowing for a fairer partitioning system. However, like most systems this work only targets the software tasks to a single processor.

2.4.5 Problems with co-design

There are a number of barriers to progress in the co-design field, but perhaps the most pressing is that the task of searching all possible partitions for an optimal solution has been shown to be a case of integer linear programming [171] and so therefore NP-hard. In the general case such a problem requires an exhaustive brute-force search. Wolpert's "No Free Lunch" theorem [248, 249] can be used to state that there is no single search algorithm that will be able to perform better than an exhaustive search of all possible partitions in the worst case. As a result, heuristic-based search algorithms must be used and it is this that has been the focus of much work [125, 70, 71, 227]. This problem can be mitigated in practice, as the *optimal* solution is rarely required, merely one that is good enough to meet the system requirements.

Secondly, most co-design frameworks rely on an accurate measurement of the performance metrics of the partitioned design to guide their heuristic search. In other words, the system makes decisions commonly based on the worst-case execution time (WCET) of both code and hardware and the expected size of compiled code and synthesised hardware. There is a massive body of work concerned with calculating the WCET of software but such analysis is very time consuming and relies on the target code being relatively small, written in an analysable language, compiled with a simple compiler and executed on a predictable processor with a minimal amount of caching, branch prediction or otherwise complex behaviour. Also, WCET analysis is rarely fully automatic.

A major issue with co-design is that the vast majority of systems all target the same architecture (a single processor with a single bus that contains a set of hardware accelerators). As mentioned, some work has been done to try to move this to a multiprocessor model [172, 124], but it has not been heavily expanded upon. Clearly a single processor will eventually prove to be a bottleneck in future systems, so this problem must be solved.

Similarly, it is difficult to accurately predict how large the outcome of hardware synthesis will be without actually performing it. Without detailed implementation-specific knowledge utilisation figures must be estimated. These analysis errors mount as the size of the design increases, making the co-design framework increasingly less effective. Similarly, it can be very difficult to predict the effect of communication and synchronisation delay in a partitioned system, and a few sources of such delay can negate the speed up that would otherwise be gained [246].

2.5 Implementation fabrics for embedded systems

This section gives an overview of the implementation fabrics that are commonly used for embedded systems. The choice of fabric is important for the system designer because it can have a drastic effect on the efficiency of their design.

2.5.1 Application-Specific Integrated Circuits (ASICs)

Application-Specific Integrated Circuits (ASICs) [204] are integrated circuits that are designed to fulfil a specific purpose, in contrast to the various series of industry-standard integrated circuits such as the 7400 series [218] of standard logic circuits. Collectively they represent the most common implementation choice for modern systems.

Being application-specific, ASICs are custom-built at dedicated silicon fabrication plants. As a result, there is an incredibly high initial cost for setting up the plant, which involves the generation of a *photomask*. Photomasks, or commonly just *masks*, are used by the ASIC fabrication process to lay out the various layers of silicon that compose the final design. The problem with ASIC development is that mask costs are extraordinarily high, and they increase further for higher-density manufacturing processes. A 90nm mask may cost around \$0.75m, a 65nm mask \$1.5m, and a 45mn mask as high as \$3m. This means that ASICs are only cost-effective if a very large number will be produced that can offset the mask cost. However, the per-unit cost of an ASIC is very low as once the mask is created it is possible to fabricate large volumes of the circuit for a relatively low cost.

The main advantage of ASICs as an implementation fabric is that they provide the highest transistor density available. ASIC designs contain well over 100 million transistors in a tiny area, and have lower power requirements and a higher maximum clock frequency than a similar system built from stock parts. As a rule of thumb, a design implemented as an ASIC is two to three times faster than the same design implemented in an FPGA of a similar technology node [155]. For certain designs, this disparity can be even greater.

Aside from their high set up cost, the main disadvantage of ASICs is that they are completely fixed at fabrication time. It is essential that the design being fabricated is perfect, because if errors are found later they cannot be corrected. As a result, the high mask cost is in addition to equally high costs of simulation and verification that must be performed on the design before it is sent to the plant to be built. It is this lack of flexibility that led to the development of *programmable logic devices* (PLDs).

PLDs attempt to keep the speed and integration level of ASICs but provide some amount of flexibility to achieve the following two goals. First, if the designer can program the devices then the expensive mask-making stage can be avoided. Second, if the device can be multiply-reprogrammed then it becomes possible to correct errors without purchasing replacement hardware and testing and verification becomes cheaper and easier. Early PLDs were very simple, allowing the synthesis of only a single combinatorial logic function. However, as integration increased the effective logic density of these devices also increased leading to the development of FPGAs, which can be thought of as truly ‘reprogrammable ASICs’.

2.5.2 Field-Programmable Gate Arrays (FPGAs)

An FPGA is an example of a class of programmable logic devices known as *gate arrays*. In a gate array architecture, transistors, logic gates, and other active devices are placed in a regular lattice pattern and connected with interconnect wires. These wires are configurable and can be arranged to connect the resources of the device in a structured manner. By placing the interconnect lines correctly, a process known as routing, the components on the device can be connected to form almost any desired circuit.

FPGAs were initially developed in the mid 1980s [50] and were marketed as an alternative way of evaluating ASIC designs. Previously, evaluating a designed circuit required that either the design was manually built from connecting discrete components, or it was fabricated as a custom-built ASIC. Both methods were time consuming and very costly. FPGAs changed this by giving the designer an implementation fabric onto which designs could be programmed. Evaluation could begin almost immediately, and once errors were found and corrected the device could be reused. This prototyping method drastically increased the efficiency of ASIC design, but also opened up new possible implementation methods. As the size and speed of FPGAs increased and their unit costs decreased, more and more embedded systems were developed that included an FPGA in the final circuit design, rather than an ASIC. This avoided the heavy set-up costs associated with creating a custom IC and is very suitable for products that are sold in low to medium volumes or that do not require the (currently much higher) logic density and clock speed of a custom-built ASIC.

On a basic FPGA, the primary resources are Configurable Logic Blocks (CLBs), interconnect and input/output blocks (IOBs). (See figure 2.20) CLBs make up the majority of the components on the FPGA and are used to create sections of logic that implement the primary functionality of the device. They are constructed from Look-Up Tables (LUTs) and flip-flops and can be programmed to perform one of a large set of logical functions on their inputs. CLBs are connected to each other by programmable interconnect which can be configured to selectively route signals across the FPGAs. It is the vast amount of interconnect that actually consumes the majority of the silicon area of an FPGA [27], up to 70% on some architectures.

In all modern FPGA architectures interconnect follows a hierarchical model. The majority of interconnect is named ‘local interconnect’ and is constructed from short wires that may only span a small number of CLBs. This is most commonly used to connect the inputs and outputs of adjacent CLBs to form a single large logic function, such as a multiplier or shift register. In order to connect distant parts of the FPGA, ‘global interconnect’ exists which comprises longer wires that may span the entire width of the FPGA. Due to area constraints, global interconnect is much less common than local interconnect and so can often be a very limited resource. Finally, global clock nets are a special type of global interconnect which exist solely to propagate clock signals throughout the FPGA. Due to the complex hardware involved in reducing clock skew, there are generally only a few clock nets available on a device, 4 on the Xilinx Spartan-IIe [253] for example.

2.5.3 Advanced FPGA architectures

Whilst all FPGAs are composed of LUTs and interconnect, modern FPGAs contain a number of additional embedded modules for performing specialised tasks. These allow for greater design flexibility as they operate at a high speed and can be used to implement functions that would take up much of the normal FPGA fabric. For performing complex control operations many high-end FPGAs include processor cores as part of their architecture, such as the Xilinx Virtex-4 [257] which contains four PowerPC 405 cores. The surrounding logic presents the cores with information and collects the results once processing is complete. Due to the fact that these cores are implemented as embedded ASICs rather than derived from the normal FPGA fabric, they can be clocked at much higher rates than processors synthesised from the

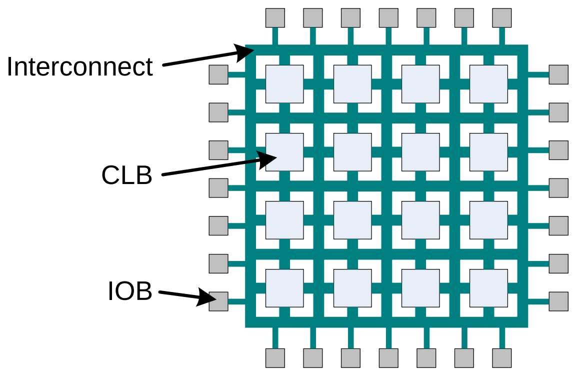

Figure 2.20: Early FPGA architecture showing CLBs surrounded by interconnect and interacting with the outside world though IOBs [251].

FPGA fabric (which are known as softcores).

Most applications require some form of memory to store programs or data. Whilst RAM can be synthesised by combining LUTs appropriately, this is very inefficient. Therefore integrated RAM blocks are a common feature in FPGA architectures. In Xilinx and Altera FPGAs these are called BlockRAMs and on the largest Virtex-5 FPGA [255] there are over 11 Megabytes of them. BlockRAMs are highly configurable with different widths, depths, and numbers of access ports. [6]

In addition to embedded softcores and distributed RAM, many other design elements are commonly integrated into FPGA fabrics. Some devices include dedicated multiplier units that can perform calculations much faster than similar logic synthesised from the CLBs, clock management circuits distributed across the FPGA help to manage clock skew and create stable clock dividers and high-speed I/O modules such as the Xilinx RocketI/O modules allow off-chip communication at 11 GBit per second [252].

2.5.4 The FPGA design process

FPGAs store their current configuration in special configuration memory. As this memory is volatile, the device must be reconfigured with a configuration file (known as a bitfile) each time it is powered up. The process of creating a bitfile to program an FPGA with is shown in figure 2.21.

1. **Design:** The required design is expressed in a form that is acceptable to the FPGA design tools. This may be a schematic, an EDIF netlist, or hardware description language (section 2.4.1).
2. **Translate:** The user input is translated to logic gates, essentially converting all forms of input to the schematic form.
3. **Map:** The resulting logic gates are mapped into CLBs and other atomic elements of the target FPGA fabric.

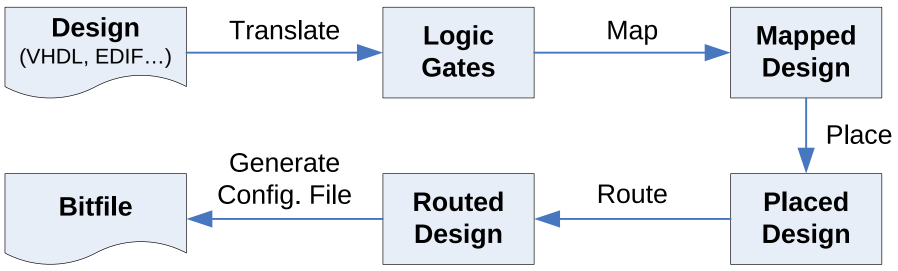

Figure 2.21: The standard FPGA design flow.

4. **Place:** The mapped CLBs are placed onto the device. The mapping algorithm will attempt to keep logically related CLBs together to minimise routing. The placement algorithm is a version of the bin packing problem and is NP-complete. [267] As a result this stage can take a long time to execute and at times of high utilisation the placer must resort to a simple exhaustive search.

5. **Route:** The interconnect between the placed CLBs is finalised. The routing algorithm attempts to use the shortest interconnection lines possible to reduce propagation delay and power consumption. Due to the large amount of interconnection required by most designs, it is possible for a design to fit onto a device at the placement stage but for routing to be impossible due to lack of space. Again, this problem is NP-complete.

6. **Bitfile generation:** The placed and routed design is converted to a bitfile that can be used to configure the target FPGA. The final bitfile can only be applied to the exact FPGA model for which it was created.

2.5.5 Coarse-Grained Reconfigurable Arrays (CGRAs)

There are two major disadvantages that can be observed with the use of FPGAs for digital design:

- **Interconnect cost:** As mentioned previously, unlike ASICs, interconnect makes up the majority of the area of an FPGA. The ideal FPGA architecture could potentially connect any CLB to any other CLB in the device as this gives the most flexible implementation fabric and simplifies the design tools significantly. However this cannot be implemented on any reasonably-sized device because the routing costs would be too high. FPGA architects must attempt to balance ease of use with the resultant routing costs.

- **Complexity of place and route:** The place and route stage of FPGA development is a very computationally-expensive step because the placer must examine a huge number of potential layouts.

Both of these problems are associated with the small size of the reconfigurable units in the FPGA fabric. The idea behind Coarse-Grained Reconfigurable Arrays (CGRAs) is that if these units are made larger the device becomes less flexible, but the design tools have a much simpler solution space to examine. Typical CGRA-based systems are the MorphoSys [201] and

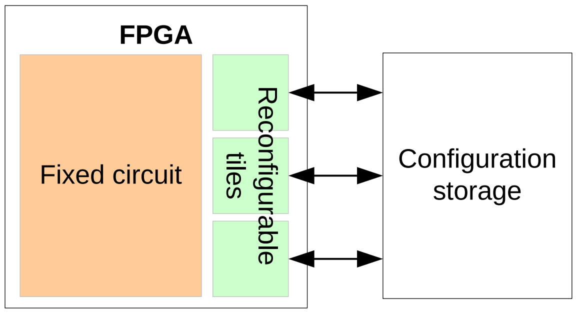

Figure 2.22: Partial dynamic reconfiguration - self-contained tiles of the design can be swapped between the FPGA and external storage in the same way that conventional virtual memory swaps pages of data between main memory and magnetic storage.

RaPiD [66] systems. Due to the limited flexibility of CGRAs, control logic can be difficult to implement. As a result, many CGRA systems are designed to be tightly coupled with a standard processor, such as ADRES [163] and REMARC [167]. CGRAs are limiting for general-purpose development and have not seen the penetration into industry that FPGAs have, but they fill an important niche in providing flexible and easy to use co-processors for data streaming applications.

2.5.6 Partial Dynamic Reconfiguration (PDR)

A major benefit of an FPGA-based implementation fabric is the ability to reconfigure the device at run-time. The configuration engines of modern FPGAs allow a running device to be stopped and reconfigured with a different bitfile, effectively turning it into a different circuit. This gives rise to entirely new application areas and allows for an FPGA design which can respond to a mode change or other significant event in the system by altering its behaviour drastically. This dynamic reconfiguration allows for a number of mutually-exclusive features to be implemented on the same hardware at different times, thereby reducing silicon area. A good example application of this ability is a hardware video decoder. A dedicated ASIC is normally required to decode each supported video format, but if an FPGA is used instead then different bitfiles for the various formats can be stored in a ROM and then programmed in when required. Also, firmware updates can be issued to add support for new formats after the device has been shipped.

Recent FPGAs also allow for a more fine-grained reconfiguration mechanism. Rather than reconfiguring the entire device as described above, it is possible to load a partial bitfile which only affects a subset of the device whilst the unaffected areas continue to run. This technique is known as *partial dynamic reconfiguration* (PDR) and was introduced in the mid 1990s with the Xilinx 6200 FPGA [51]. PDR allows designers to ‘swap’ in and out ‘pages’ of hardware in a manner that is analogous to the way that virtual memory systems swap pages of virtual memory out to disk when physical memory is full. This has created the concept of *virtual hardware* that, like virtual memory, allows a reconfigurable array to appear to the designer larger than it actually is by swapping unused areas of hardware to external storage.

One difficulty with the use of PDR is that no FPGA allows for individual logic cells of the device to be reconfigured. Reconfiguration must instead be done on a tile-by-tile basis, where the size of a tile is defined by the FPGA family. The reconfiguration tiles of the first FPGAs that supported PDR were columns that ran from the top of the device to the bottom. This meant that complex arrangements for providing cross-device communications were necessary. Essentially, PDR was one-dimensional. Later FPGAs, such as the Virtex-4 [257] increased the granularity of PDR to allow a true 2D grid of tiles, thereby simplifying its use.

Another consideration when using PDR is the time associated with performing the reconfiguration. Bitfiles are large and must be passed in to the FPGA though the configuration port, which is usually clocked much slower than the FPGA itself. Consequentially, even with partial bitfiles reconfiguration is of the order of milliseconds [197]. Whilst not prohibitive, this is much longer than processor context switch times and so care must be taken to ensure that timing constraints are still met. Further, it is much easier to perform PDR if the designer first shuts down the FPGA (stop the clock but retain internal configuration and state) and then starts it up again once configuration is complete. This reduces the possibility of metastability from signals crossing reconfiguration boundaries. However, this affects the timing of the entire circuit. PDR does not require the device to be shut down, but the complexity of the resulting design is increased.

Although PDR is still not commonly used in industry due to poor tool support and the vastly increased complexity of the final design, numerous research projects have begun to exploit its potential. Recent work [226, 20] has developed a framework for using PDR to reduce the number of control systems present in modern automobiles. New cars can contain over 70 microprocessor systems for controlling the vast array of electronics that are now fitted as standard. Some of these processors are safety-critical and are not considered by this work, but the majority are of low-importance and are only infrequently required by the user. Examples include the controllers for the electric windows or the sunroof. The presented work partitions the FPGA into two areas. One area is a control section responsible for parsing incoming CAN messages and driving the reconfiguration. The rest of the device is a set of ‘slots’ into which control units can be configured. The number of slots therefore determines the maximum number of parallel operations that can be executed at any one time. If the control section receives a message for a device which is not currently occupying one of its slots then it decompresses a stored bitfile for that device and configures it into a free slot using a least-recently used policy similar to that of memory caches. The resulting system has been shown to vastly reduce the hardware requirements for soft real-time components without a perceptible degradation in responsiveness from the user’s perspective, with most response times less than 10ms. A system diagram is shown in figure 2.23.

A common paradigm that PDR is applied to is that of a custom instruction set processor, a processor which can dynamically load new operations at run-time. DISC [245] is a processor entirely implemented on a standard FPGA, using a portion of the device as a fixed execution controller and the rest of the device as space in which to store custom instructions, thereby accelerating the application by up to 24 times. The Chimaera reconfigurable processor [102] is an ASIC-based processor which contains an amount of embedded FPGA-style reconfigurable hardware. Chimaera uses this reconfigurable hardware to store the execution units of custom machine instructions, whilst the ASIC section performs all standard operations. PDR has also been used to create a reconfigurable co-processor that sits alongside a standard processor [103].

NoC-based designs have been proposed [158, 110] that may allow PDR to be used in a more

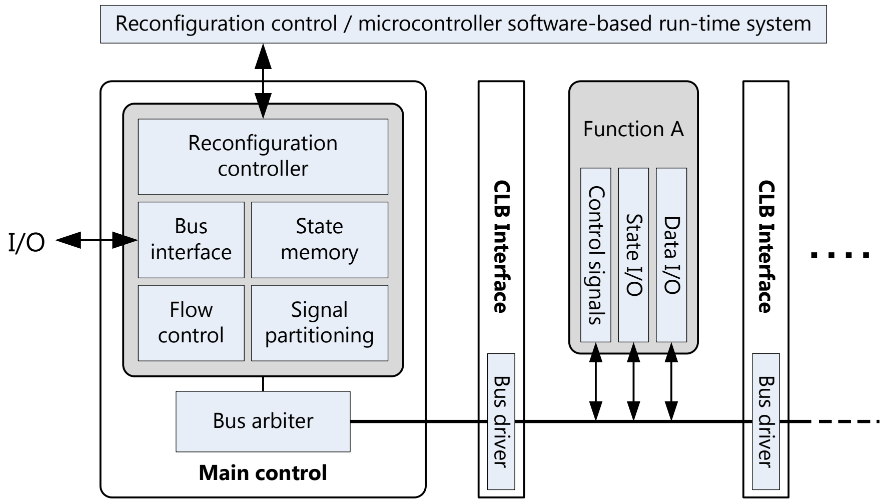

Figure 2.23: Run-time system with run-time reconfiguration support and a soft processor. [226].

general sense than the systems above. The problem that this work attempts to solve is that it is very difficult to retain on-chip communications between dynamically changing components as they are swapped in and out. These systems work by ‘hooking’ rectangular reconfigurable tiles onto a static network layout as they are configured into the device. Essentially, a static grid-based network of interconnect is laid down on the chip with router components placed at regular intervals. This may be in one or two dimensions. This part of the design is static and is not affected by run-time reconfiguration. Most of the routers are not connected to a tile and so sit idle, but when the system wants to swap in a hardware tile it can select a free router and ‘hook’ the incoming tile off the selected router. If the tile is very large it may overwrite a number of other routers in the surrounding area, but the grid layout will be maintained. This way, incoming tiles can always communicate with each other, as shown in figure 2.24. Recent work [52] has considered the problems associated with creating bitfiles that can be located anywhere in the reconfigurable device and uses a ‘bitstream filter’ to give the illusion of relocatable bitfiles. The problem with these NoC-based techniques is that the limitations of PDR granularity did not initially allow these to be efficiently implemented. Even though the layout of the network grid remains the same, the act of reconfiguring a tile (even with the same data) can potentially cause metastability in signals that pass across a tile boundary and the contents of any on-chip storage elements will be lost. As the technology matures, however, these problems will be reduced and it is likely that future systems will be based on work like this and the automobile framework above.

2.6 Problems with existing research

This chapter has introduced a number of different approaches that aim to afford the programmer better use of complex embedded architectures. Whilst many of them show some level of success, they display a number of overarching problems that can be broadly separated into

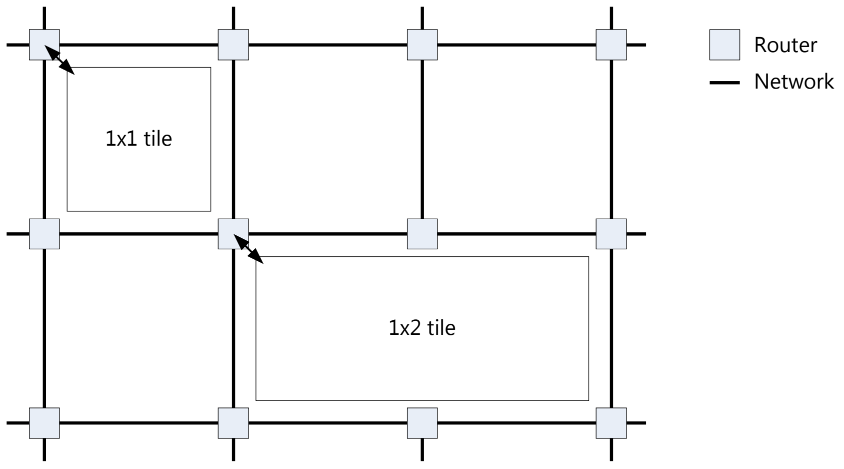

Figure 2.24: On-chip networks are being developed to give a predictable structure to PDR-based systems and thereby make them easier to develop.

two main categories - conceptual and practical. Conceptual problems (sections 2.6.1 to 2.6.3) arise from a fundamental limitation of the approach's premise and are the most severe as they tend to represent a limitation that implementation ingenuity cannot hide. Practical problems (section 2.6.4) are slightly less serious, but may be onerous for the end-user or the developers of compilers and toolchains. These two problem categories are discussed in the rest of this section.

2.6.1 Inappropriate abstraction models

Many of the presented approaches for allowing better use of complex architecture employ either an extension to an existing language (almost always C) or develop an entirely new language. The idea behind this is that existing languages lack the expressive power to describe architectural details so by adding these in the programmer will be able to exploit future architectures more easily. One example of a language extension is the RTSJ, which is a real-time extension to Java which includes a large number of concepts that allow the programmer to discuss the processors and memory hierarchy of the target architecture. An example of a new language is Sequoia which focusses on mapping the program more effectively into the memory hierarchy.

The problem with the approaches presented is that they all use broadly the same abstraction model that programming languages have used since the development of high-level languages decades ago. As required by the standard programming model, features such as operating systems, memory hierarchies and hardware devices were developed to be transparent to the end-user. This resulted in the model developing into a stack of virtual machines, with the higher layers built upon guarantees from the lower layers. This is shown in figure 2.25.

Each virtual machine hides underlying details to simplify programming and insulate the programmer from implementation changes. VM$_1$ is built from the actual hardware and presents a single contiguous address space and in-order execution of opcodes. VM$_2$ is presented by the

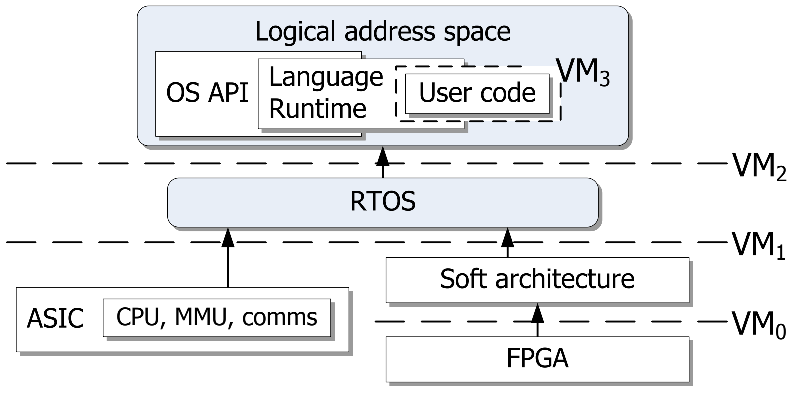

Figure 2.25: The stack of VMs in the standard general-purpose architecture.

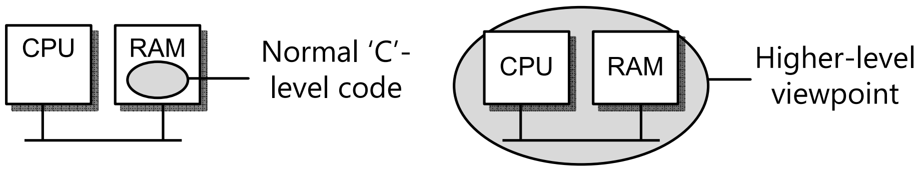

Figure 2.26: Traditional programming languages describe what runs on a processor, not the architecture in which the program executes.

real-time operating system (RTOS) so that each process believes it has sole control over the processor, access to atomic actions, and other RTOS features. VM_3 represents the language-level virtual machines of languages like Java and Smalltalk. Unfortunately these VMs add inefficiency because they do not allow easy access to underlying hardware. It therefore becomes difficult for code at higher layers to access custom hardware without the use of hand-written libraries and techniques that are outside the abstraction model of the programming language. When custom hardware elements are introduced they either cannot be exploited (as with function accelerators) or they cause architectural assumptions of the VMs to fail and user code no longer functions (as with non-contiguous memory layouts).

As a result of this, from a semantic point of view programming languages describe the software running on a single processor (or a collection of tightly-coupled homogenous processors) from within an implicit standard architecture. They do not describe the system as a whole, and the details of how the code is mapped into the architecture are hidden from the programmer. This is shown in figure 2.26.

The presented approaches attempt to solve this discrepancy from within the scope of normal programming languages by keeping the same abstraction model but allowing extra keywords or pragma-style concepts that allow the programmer to 'drill through' the abstraction layers. Whilst this does allow the programmer to achieve the implementation they require, it has a number of problems. Primarily it is not the way that the programmer wants to work. Consider using C to target a multi-processor system where each processor has private memory and there is also a block of shared memory. The programmer may write the code in figure 2.27 but they do not have any control over where the threads `thread1` and `thread2` are actually placed,

```
pthread_t thread1, thread2;
int shareddata[4500];

void *task1(){...}
void *task2(){...}

int main(void)
{
   pthread_create(&thread1, NULL, task1, NULL);
   pthread_create(&thread2, NULL, task2, NULL);
   pthread_join(&thread1);
   pthread_join(&thread2);
   return 0;
}
```

Figure 2.27: C cannot natively describe shared data.

and they do not know to which memory block `shareddata` will be mapped.

To map the shared data into the correct space in RAM, the programmer must communicate their desired mapping to the linker. To do this they tag the `shareddata` declaration with a new section name and instruct the linker to move that section to the appropriate place. Using the `gcc` toolchain the programmer tags the declaration like this:

```
int shareddata[4500] __attribute__((section ("sharedmemory")));
```

And then declares the new `sharedmemory` section in a custom link script which is passed to `ld`. If 0x8C000000 is the address at which shared memory starts the declaration looks like this:

```
. = 0x8C000000;
sharedmemory : { }
```

Whilst this works, it is very complex and requires the programmer to do a lot of work outside of their source language. Also it is architecture-dependent when it does not need to be. It is very rare that the programmer wants to manually place the data at a specific address, instead they wish to express the concept that `shareddata` should be in memory that is accessible to the host processors of threads `thread1` and `thread2`. Without these concepts the programmer is forced to manually place the data. With the amount of processor cores in modern supercomputers [202] this is clearly not sustainable for supercomputing environments. Embedded environments are smaller and less regular so mapping is still feasible, but it is error-prone and must be repeated after any significant software or hardware changes.

2.6.2 Limitations of high-level synthesis

There is currently no high-level synthesis language that has completely replaced the use of HDLs like Verilog and VHDL. The reason for this is that the abstraction models of high-level synthesis languages mimic that of software languages and consequentially do not give the designer the ability to choose between different implementation strategies. When designing systems, the hardware designer may choose between interacting state-machines in raw logic, opcodes on an embedded processor core, dataflow and asynchronous logic circuits, processor and function accelerator pairs, or some combination of these techniques. High-level synthesis systems do not give this freedom, and tend to select a single strategy which will work very well for some designs and very poorly for others.

For example, Handel-C creates interacting one-hot state machines, but for designs that are best implemented using mainly asynchronous logic it can only describe an inefficient solution that is unnecessarily large and takes longer to execute than required. Similarly, Lava is based on a function composition paradigm that describes signal processing circuits well but cannot express state easily.

It appears that high-level synthesis has hit a self-imposed limit and it will not improve without a change in the way it is realised. Synthesis works well within a narrow design space, but rapidly diverges from the optimal solution outside of this space, to the point where often large classes of designs cannot be implemented at all. As a result it may be tempting to simply keep adding extensions to the source language, but this cannot completely bridge the semantic gap between design space and implementation fabric because both are changing at a pace that is far too rapid. For example, the globally-asynchronous, locally-synchronous design paradigm [41] discussed in 1984 is now fully embraced by modern SoCs yet programming languages are still not capable of expressing this kind of paradigm well. The Ada Distributed Systems annex [28] appeared in 1995, but it is too heavyweight for efficiently targeting SoCs with tightly-coupled communication and memory systems.

2.6.3 Poor support for dynamic systems and architectures

Whilst many embedded systems are entirely static, as embedded platforms become increasingly parallel the issue of dynamic migration of computation is introduced. Embedded systems commonly include the following dynamic features:

- **Multiprocessor systems:** As with desktop machines, devices with multiple execution units support a form of POSIX-style threading model that schedules threads over many cores, perhaps allowing migration at run-time.
- **Power scaling:** Reducing power consumption is essential for many embedded devices. A device that is under low load may switch some cores off and migrate all their computation onto a smaller active set.
- **Fault tolerance:** As cores fail, if possible it is desirable to migrate any threads that were executing on them to other areas of the system. This also applies to the failure of on-chip communications media.
- **Load balancing:** Load balancing systems allow for systems to better deal with infrequent bursts of high volumes of computation (e.g. in a network switch).
- **Dynamic reconfiguration:** FPGA technologies allow embedded architectures to change at run-time through partial dynamic reconfiguration [226], leading to high amounts of dynamism.

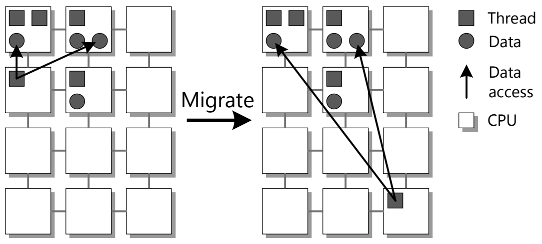

Figure 2.28: Unguided thread migration can lead to increased inter-thread distances and poor performance.

A key issue, therefore, is how standard programming languages may allow efficient exploitation of these dynamic features of the target platform by the programmer. This is difficult in typical languages (eg. C) because the inherent programming models do not represent the underlying hardware architecture. The solution used in general multiprocessor and distributed architectures is to use middleware-based solutions like MPI [97], CORBA [183] or PVM [88]. However these are not appropriate for embedded systems due to their inefficiency and need for complex Operating System and communications support. For programming multiprocessor embedded systems therefore there are two distinct issues to consider: *program structure* and *memory coherency*.

Program structure

Programs are generally expressed in units such as threads and data objects. However, the relationship between threads and data (ie. which data objects are required by a thread) is not present in most programming languages. Then, when a thread migrates from one processor to another, the language runtime lacks the necessary structure that would allow it to migrate with it any related threads and the data they use. This unguided thread migration can lead to increased inter-thread distances and therefore higher communications latency and poor memory access times, as shown in figure 2.28. This problem affects embedded systems particularly, because they are rarely homogenous grid architectures of the kind found in supercomputing environments.

In an attempt to mitigate this problem, some languages allow the programmer to bind threads to only execute on a subset of available processors. Thread affinities in RTSJ [61] are one such example. Affinities describe which processors a thread may be scheduled upon at runtime, but there is no way to bind items of shared data to that thread or to state that threads should be grouped and moved together. It may sometimes be possible to infer this data to a limited extent with static analysis, but this does not help in the general case.

Chapel's locales can group threads and data, and then a locale can be bound to specific nodes of the architecture. However locales are low-level, forcing all threads in a locale to execute on the same processing node and limiting the sharing of data between locales. Also, Chapel's shared-memory model and a regular grid architecture are rather heavyweight for many embedded systems. UPC allows threads to have private and shared data, and it also introduces the concept of data affinity which states that a particular thread 'owns' a particular data item. Unfortunately it does not allow clustering of threads as the model is flat rather than hierarchical, and like Chapel it is not an embedded language and was designed for heavyweight supercomputing architectures.

Coherency

Whilst embedded architectures employ complex, hierarchical, heterogenous models, almost all existing languages assume a contiguous global address space. The programmer cannot effectively map the data of their program onto the memory hierarchy, and has little control over data transfers. The result of this single address space assumption is that many software languages place heavy demands on cache coherency algorithms, usually that the entire system is kept coherent. This is very expensive (in terms of execution time and required hardware) and it does not scale to support large numbers of caches. On embedded platforms it is even worse because their application-specific nature means that frequently the programmer actually only needs to keep a few small areas of the system coherent, as determined by their application.

Lightweight schemes that limit coherency into islands are more scalable than complete coherency solutions [149], but they require significant support from the programming environment. This support is typically not available. C assumes that the entire program is within a single contiguous address space, and its non-analysable pointer arithmetic requires perfect coherence across all memory. Furthermore, because threads are not a first-class part of the language they cannot be reasoned about in terms of the data that they use. Ada's Distributed Systems Annex allows for the notion of separate memory spaces, but it produces very heavyweight partitions that cannot communicate or share data without the use of explicit communication. Inside a partition the memory model is similar to C. The Java language has a flat memory model, as this is what is exposed by the JVM. The RTSJ [91] allows the programmer to provide more information about memory regions using the concepts of scoped and immortal memory. There is no concept of threads and data being related and coherency is still presumed to be across all memory. As mentioned previously, UPC allows threads to claim an item of data as its own, but the model requires all shared data to be accessible to all threads.

Some recent work has considered ways to limit this coherency problem. Huang et. al. [109] augments the programmer's source code at points where locks are requested and released with library calls to determine what shared data is required by which parts of the design. Similarly, Virtual Tree Coherence (VTC) [73] uses a tree-based coherency system to limit coherence actions to the necessary subset of the nodes of an on-chip network.

In general, however, existing languages do not allow the programmer to limit their coherency demands and therefore allow the implementation of a more efficient system that better reflects the needs of the application.

2.6.4 Practical issues

Along with the conceptual problems listed above, there are a number of practical problems that most existing solutions exhibit. The most obvious is that whenever a new language is designed it cannot be used without the development of an associated compiler. This problem is also evident with language extensions, because if an open-source compiler for the base language exists then it can be extended, but it will take a lot of verification and validation effort before the new compiler is trusted enough to be used in an industrial or safety-critical environment. This applies to other parts of the software toolchain also, such as debuggers, profilers, and development environments.

Related to this problem, reuse of legacy code is often not possible when a new language or language extension is developed. Clearly with a totally new language all code must be rewritten, but this problem also applies to language extensions that augment the target domain of an existing language. A good example of this is Handel-C, an extension of ANSI C for performing high-level synthesis of hardware. Handel-C is almost a complete superset of ANSI C (only a few C operations are not permitted) but it is not possible to reuse existing C code to any degree because whilst the result may be functionally correct, it is very likely to display undesirable non-functional properties.

For example, standard software engineering encourages as much code reuse and generalisation of functions as possible because caching effects cause such code to execute faster. However, in a naive hardware implementation this actually has the opposite effect. Code reuse can result in wide multiplexers on the input and output ports of the hardware functions (because many other parts of the system use it). This consumes logic and results in slower maximum clock speeds. Less general coding will result in multiple hardware functions where there originally was only one, but they will have narrower multiplexers and run faster. In essence, whilst developing in Handel-C may look like developing in C, it is a very different paradigm and requires different skills from the programmer.

The issue of programmer expertise is also of interest to industry. Retraining programmers to develop in a new language is wasted time from a company's perspective, and it may take months before programmers can produce code of the same quality as in the previous language.

From a more research-oriented view, because there is such a huge number of new languages and language extensions (as presented in this chapter) the community is in danger of becoming fragmented. Research does not benefit the majority because everyone is using their own language and must reimplement the new results in their own compilers and toolchains.

Finally, one clear problem with many of the research languages presented here is that they have a tendency to be domain-specific, or to only tackle a single problem. Sequoia is a good example, as it has solely concentrated on allowing developers to program complex memory hierarchies, but it does not attempt to expose custom hardware elements to the programming language. System design languages such as SystemC are very good at exploiting custom hardware elements, but their synthesisable subset is a lower level of abstraction than high-level languages like Java, which can limit programmer productivity. It is not possible for the programmer to use the interesting features of all these languages, they must select a single language and accept the disadvantages of that language also.

Chapter 3

Compile-Time Virtualisation

3.1 Overview

Chapter 2 discussed the observation that existing software development models incorporate a stack of abstraction layers (figure 2.25) that are unsuitable for the programming of non-standard system architectures. This conventional abstraction stack evolved at a time when underlying hardware consisted solely of a single processor with a single, logically-contiguous, memory space. Software languages began to hide these details from the programmer because they were static, resulting in an implicit target architecture. Modern languages still use this model, so do not allow code to target varying architectures without breaking existing abstractions.

Most recent research has attempted to solve this problem by extending software languages with architectural concepts, but as discussed in section 2.6.1 this technique has currently only met with limited success because of a fundamental semantic discontinuity – software languages describe the actions to be performed by a processor, they do not describe the system in which that processor exists. Furthermore, section 2.6.4 highlighted that it is desirable to use existing languages and compilers if possible due to the rapid rate of hardware evolution. It is infeasible to develop a new language and associated toolchain for each new paradigm that appears, and currently many newer hardware concepts are still either unsupported by languages (such as scratchpad memories and partial dynamic reconfiguration) or poorly supported (non-uniform memory architectures).

The approach taken in this thesis and presented in this chapter is to use a more appropriate programming model for describing embedded software that can be used to express the architectural mapping concerns that are inherent in embedded development. The model uses clustering to introduce controlled dynamic behaviour that can be used to increase scalability of the final system whilst preserving thread and data locality. This is supported by virtualisation-based techniques that present the programmer with an intermediate platform for which software can be developed using standard programming languages. This is termed the *Virtual Platform* (VP) and is tailored to fulfil the architectural assumptions of the chosen source language, thereby aiding development. Unlike standard virtualisation, the mappings from this

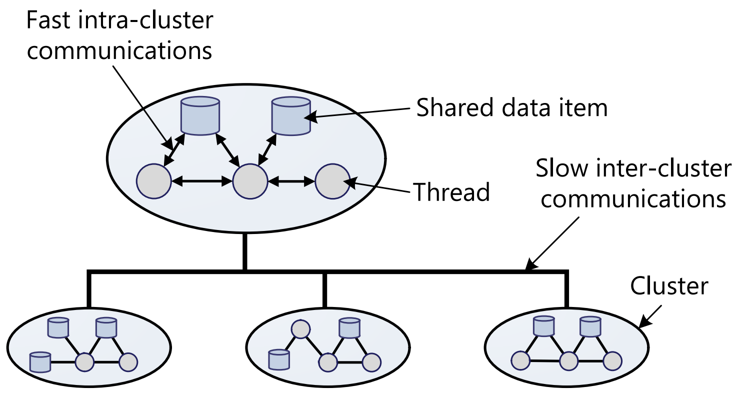
Fast intra-cluster communications
Shared data item
Thread
Slow inter-cluster communications
Cluster

Figure 3.1: Clusters in a large software system.

intermediate platform to the target hardware are exposed to the programmer for high-level alteration. To maintain efficiency, all virtualisation takes place at compile-time rather than run-time (as with existing techniques). Therefore, the technique introduced by this thesis is called *Compile-Time Virtualisation* (CTV).

Section 3.2 introduces and defines the clustering model, which is then refined into a complete system model in section 3.4. Section 3.4.5 discusses the way that three existing languages (C, Ada and Java) can be represented in this system model. CTV is then introduced in section 3.5 as a way of implementing the clustering system model in resource constrained embedded systems. CTV's system model is detailed in section 3.6, and section 3.7 describes a novel communications layer called the *Object Manager model* that is used by CTV to distribute OS-style services in a scalable and efficient manner.

3.2 Clustering

Section 2.6.3 discussed the problems that existing languages have when addressing the dynamism that is present in modern embedded systems. One of the identified issues is that the programmer cannot group threads into cooperating thread groups along with the data that they use. In essence, the programming model is flat rather than hierarchical. This causes two main problems when attempting to distribute the program over a heterogeneous embedded architecture. First, during thread migration tightly-linked threads can become separated from each other and from their data, leading to longer memory access times and greater inter-thread communication latency. Second, the flat memory models used by standard languages force the system to maintain cache coherency across the entire architecture. This is unnecessary, because for most programs coherency is only required between a small number of cores per item of shared data.

Rather than considering the whole system at a single level of granularity (like the threading models of Hoare's original CSP, MPI-based systems, and POSIX pthreads[1]), this thesis in-

[1]POSIX threads are not completely flat as they can be spawned hierarchically (one thread creating sub-threads) but from that point onwards they execute flat and independently.

troduces a model that specifies the system as a set of interacting *clusters*. This is shown in figure 3.1. The aim of this model is to allow the computation and communication of the system to be better expressed. The clustering model [94][2] is a system specification model formed from the PGAS languages 2.1.4 and using elements of MPI [97] and ccNUMA (e.g. [152, 143]). These all acknowledge the importance of system partitioning, in which partitions are used to delimit communication requirements and break up the memory model into logically distinct sections.

A cluster is composed of:

Threads A possibly dynamic set of programmer-defined application threads. Whilst this primarily refers to language-level threads (like those of Java or the tasks of Ada) they might also refer to more finely-grained mechanisms, such as cobegins or parallel for-loops.

Shared data Items of shared data that are used primarily by the threads of the system.

Two relationships are defined by the clustering model:

Sibling $\leftrightarrow$ Sibling: *Siblings* are threads and data items contained within the same cluster.

Parent $\leftrightarrow$ Child: The threads and data items contained within any given cluster are referred to as the *children* of that cluster. Accordingly, the cluster can be referred to as the *parent* of those items.

The clustering model is used to allow a system specification to express the relative importance and frequency of the various communications that take place in the system. This model is not a semantic one and it does not change, for example, the scoping rules of the source language. It provides additional information that is not included in most existing languages that can be used to *influence* the implementation. This extra information is called *coupling*. Coupling is a relative partial ordering between the threads and data items in a system and affects the following two properties:

- **Communications:** If two items (threads, data items) are *tightly-coupled* then they require frequent, or low-latency communication. This typically means that they coordinate on the same sub-problems, share the same input and temporary data, and exchange frequent, low-level messages and locks. For example, a thread is tightly-coupled with its local data items. Conversely, if two items are *loosely-coupled* then their communications are less frequent, higher-level, and not as critical to the overall throughput of the entire system.
- **Coherency:** If two items are tightly-coupled then they are afforded a greater level of cache coherency than items that are loosely-coupled. Cache coherency is frequently treated as a binary property (either present or absent) but this is not the case. Many different coherency implementations exist that provide different levels of service at differing costs. For example:

[2]The paper cited here used the term *Islands of Coherency* to refer to an earlier version of what is now the clustering model.

– Hardware-based cache coherency systems (see section 3.6.2) offer the fastest coherency but consume a large amount of die area and do not scale well to large numbers of coherent units.
– Automatic software-based coherency uses a run-time system to monitor shared data accesses and invalidate cache lines accordingly. This does not require any extra hardware but increases the workload of the processor as it must now check for potential coherency problems.
– Explicit software-based coherency (such as that used by the Rthreads system [65]) uses offline analysis to determine potential coherency problems at compile-time. This results in the lowest processor overhead, but pessimistic analysis can result in increased communications requirements and therefore increased latency.

A flat model presumes that the same coherency model is applied uniformly across all components of the system. The clustering model provides extra information at compile-time from the programmer that may be used by the implementation to relax this restriction. For example, loosely-coupled items may use a coherency model that provides slower coherency but at the resulting reduced implementation cost.

3.3 Clustering model motivation

The clustering model must be considered in terms of complex, non-unform architectures for its purpose to become apparent. In more regular SMP-style architectures, use of the model is not as important, although it supports dynamic systems well (see later). Whereas these architectures have either a shared bus or otherwise uniform communications, in a complex architecture this is not the case and all manner of communication and memory topologies may be introduced. Consequentially, thread and data positioning has a large effect on system performance, yet source languages for the most part do not provide coupling information to guide these choices. Clustering model allows the programmer to provide this missing information.

The clustering model does not limit variable scope or in any way change the source programming model. If a variable is visible across the entire system (such as when using a language like C) then it still is when using the clustering model. Instead, the clustering model should be viewed as providing modelling information about where that data is most likely to be used. The requirements of the language must still be fulfilled by the implementation, but it may choose to provide a reduced service for loosely-coupled communications and concentrate on tightly-coupled areas (clusters) by placing data in a memory space close to the processors that will access it. Processors that host loosely-coupled threads may still access it, but the communication will be slower.

The model provides good support for scalable development. With a flat model all threads are defined at the same level. Therefore, because the programmer cannot express otherwise, a flat programming model implies that cache coherency and communication requirements apply equally over the entire system, limiting scalability. The necessity to discover this information is often then pushed to an automatic profiling or hot-spotting system that can tune a system during runtime. In contrast, the clustering model allows relative bounds to be placed on the communication and coherency requirements of the application at compile time. This can allow the compiler and run-time systems to use weaker, more scalable coherency solutions for

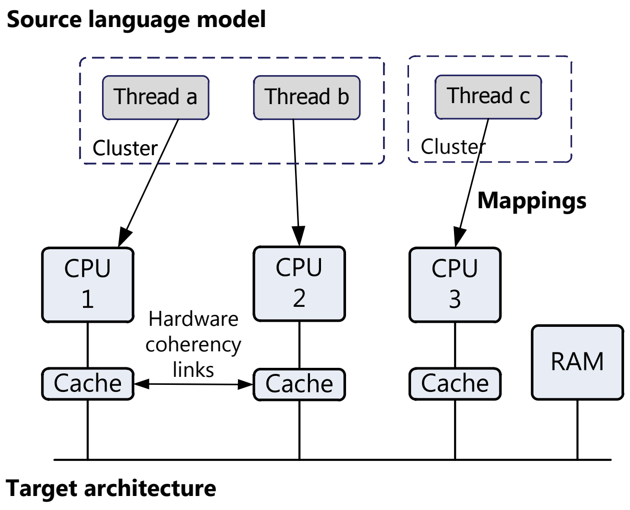

Figure 3.2: Example showing how the clustering model influences mapping to target architectures.

communications that are less time-critical.

Finally, the clustering model supports dynamic systems in which threads and data may migrate (due to OS activity, power management, fault tolerance etc.). In a flat model migrations are undirected. In the clustering model, when a thread is migrated the system can choose to also migrate its siblings in order to maintain the coupling of the system. Constraints can be expressed on migration in an architecturally-neutral way. Section 5.3 investigates the costs associated with directed and undirected migrations and demonstrates that for the majority of systems the clustering model is preferable when compared with undirected migration.

Figure 3.2 gives an example of how the clustering model can influence the implementation toolchain without introducing architecture-specific constraints. In the example system depicted there are three threads - a, b and c. Threads a and b are placed inside a single cluster and so are more tightly-coupled with each other than they are with thread c. Consequentially, if this system is implemented on a system in which only two of the processors support hardware cache coherency this specification provides a hint that these cores should be occupied by threads a and b. Communication with c is less time-critical so its placement is less constrained.

3.4 System model

This section defines the system model used by this thesis that implements the clustering model of the previous section. This model is used to represent high-level features of embedded software, the typical embedded architectures that such software runs on, and the mappings between the two.

The techniques developed in this thesis support embedded system development using a *single-program model*, in which the programmer provides a single program that wholly de-

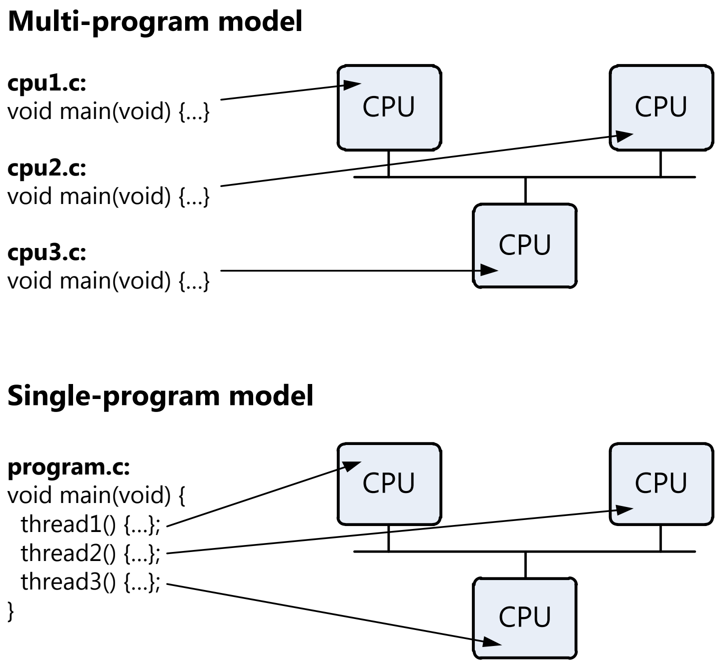

Figure 3.3: The single-program model compared to the multi-program model.

scribes the operation of the entire system. This is in contrast to a *multi-program model* in which the programmer has to provide a separate program for each execution unit (processor, co-processor, etc.) of the system. These are shown in figure 3.3. The single-program model is often preferable from the developer's perspective because it supports architecturally-neutral development (it is possible to develop software without knowledge of the number or type of available processors in the final system) and high-level language features can be used to describe communication and coordination between concurrent units. A multi-program model forces explicit programming of these interactions.

This thesis assumes a single program model with the following characteristics:

Concurrency As a consequence of the single-program model, the languages used for system development must support concurrency otherwise it would not be possible to describe the operation of a system with more than one processor. This concurrency is considered by the system model.

Shared memory threading Due to the stated focus on real-time embedded systems development, the system model is based on the characteristics of three languages frequently used in this domain - C, Ada and the Real-Time Specification for Java (RTSJ). Accordingly, an imperative programming style is used with explicit, coarsely-grained parallelism. The model is still generic enough to support other languages and programming styles. Note that whilst Ada and Java both include concurrency as integral to the language, C does not and it must therefore be added through libraries. Most existing concurrency libraries such as TBB [187] and the Boost threads library [242] rely on the use of C++, so it is therefore assumed that the C programs discussed make use of the POSIX pthreads

[115] library. The pthreads library allows the expression of concurrency though operating system threads rather than pure language threads, so pthreads expresses a flat threading model.

Program structure The model describes the structure of the input program rather than the individual operations that it performs. This is because it is used alongside the programming language and not as a replacement for it.

Target architecture description The target architecture of the embedded system is modelled so that its characteristics are available to the system. As with the previous point, this modelling is high-level and not a complete description of the hardware.

3.4.1 System model layers

The system model is composed of three layers:

- *Program layer* (section 3.4.2): Describes the input program in terms of the threads and data items that it is composed of.
- *Logical layer* (section 3.4.3): Sits below the program layer and allows the definition of clusters that the programmer can use to describe the coupling of the system. The program is mapped into these clusters.
- *Target layer* (section 3.4.4): Describes the target architecture of the system, onto which the clusters of the logical layer are mapped.

The model is depicted in figure 3.4. The program layer is mapped into clusters of the logical layer, the clusters are mapped to the target layer. These mappings are described by the logical layer (section 3.4.3).

This model is intended for use *in addition* to the source code of the modelled application, so it is not intended to fully detail the low-level operation of the system but rather to describe the high-level features of the code. The following three sections explain each layer in detail.

3.4.2 Program layer

The program layer (depicted as the top layer in figure 3.4) expresses the input program as three sets of objects:

- Concurrent objects: A set of the units of concurrency of the source language. As noted from the requirements of the system model (section 3.4), concurrency is an essential modelling primitive, motivated by the trend towards highly-parallel architectures.
- Shared objects: Passive constructs that expose services that are called by the concurrent objects of the system. Shared objects allow multiple concurrent elements to synchronise with each other and coordinate their execution to avoid race conditions and the corruption of shared data (for example, mutexes, condition variables or protected objects).

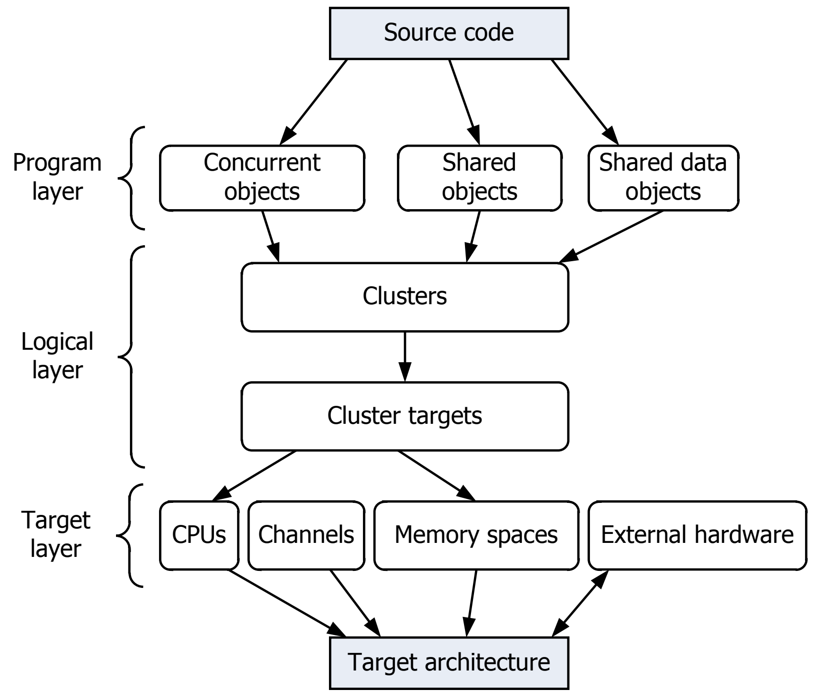

Figure 3.4: The layered system model.

- Shared data: Data items that are read and written by the concurrent objects of the system. Synchronisation is not assumed and mutual exclusion should be enforced with a shared object if required.

Note that the program layer does not explicitly model the communications between concurrent objects. For example, a CSP-based model would represent channels, over which the inter-process communications are conveyed. Such an approach would assist when mapping the application to a target architecture, but it is not supported by the languages used in embedded development that this work is focussing on (C, Ada, RTSJ). Consequentially, communications channels are modelled by the target layer as an implementation target rather than a system specification.

The elements from this layer are now defined:

Concurrent object A concurrent object models programming constructs that are used by the programmer to define units of coarsely-grained parallelism. They have the following features:

- An independent thread of control. The concurrent object executes until the object is destroyed (or stopped by another thread) or the application terminates.
- Local variables (stack, heap). Whilst executing, the object may arbitrarily create and delete local variables on either a private stack or private heap.
- Single logical address space. Whilst executing, the object may arbitrarily access shared data (for reading/writing). The programmer does not have to make any

special considerations for data accesses that are remote, they are accessed just like normal variables in the code.

- System-wide communications. The object can use the facilities of any shared objects without consideration of the connectivity target architecture.
- Arbitrary complexity. The model does not imply anything about the relative complexity or workload of a concurrent object.
- Mapped to processing elements. Concurrent objects are ultimately mapped to the processing elements of the target architecture as threads of control. If multiple concurrent objects are mapping to the same element then the use of a multiprogramming kernel is assumed. Multiprogramming concepts such as scheduling, priorities, etc. are not covered by the general system model, but may be supported by an implementation of the model.

Shared object: A shared object models language features that are used to coordinate the execution of concurrent objects, and to pass data between then in a controlled and thread-safe manner.

- Passive thread of control. The primary difference between a shared object and a concurrent object is that a shared object does not have an independent thread of control. They exist from application start to termination, but do not execute or use processor time unless they are requested by a concurrent object. From a logical view, the request is evaluated by the concurrent object which performed it.
- Provides services. A shared object exposes a set of services (in the form of procedures) that are called by the threads of the system. As a shared object is passive, the execution of these procedures is notionally executed by the calling thread, as with a procedure call in most languages. The procedures exposed vary depending on the source programming language. Some languages may only expose certain types of shared object, some may allow the programmer to define their own. For examples of shared objects in existing languages see section 3.4.5. The calling thread provides a set of arguments for the call, and after the call has been executed it receives a reply. Arguments and replies are typed using the type system of the source language. Objects must support arguments passed by value, implementations can choose whether or not to also support passing arguments by reference in limited circumstances.
- Local variables. Like a concurrent object, shared objects have access to a stack and heap and may create and store local variables whilst executing. This stack and heap are notionally separate to other objects (as the shared object may migrate into another memory space) so direct pointers should not be passed between shared objects.
- Internal locality. A shared object can be assigned to a single processing unit of the system, it cannot be split across multiple units in the system model. (An implementation may still choose to do this, providing that it is transparent to the user.)
- Internal coordination. Shared objects are used to coordinate the execution of concurrent objects and so provide coordination over the procedures that they export. Functions execute under mutual exclusion, and may obtain one of two locks in the object, the *read lock* and the *write lock*. The read lock can be held by any number of functions at any one time, but if any function holds the read lock then none can

acquire the write lock. Similarly, if any function holds the write lock then none can acquire the read lock. The write lock can only be held by one function at any one time. In addition, all exported procedures can be guarded to allow execution only when a given guard evaluates to true. The guard must be a logical predicate with no side-effects over variables local to the shared object. If a concurrent object attempts to call a guarded function which evaluates to false, the call is queued until the guard becomes true. Queued threads do not hold any locks. Guards are re-evaluated after the completion of an exported function call. Accordingly, more complex guards can be implemented by defining a new local logical variable that controls the guard and re-evaluating this variable's value at the end of all functions that might have affected its state.

- Mapped to processing elements. Like concurrent objects, shared objects are ultimately mapped to the processing elements of the target architecture. As passive objects, they are executed in response to messages from other parts of the system, so may be implemented in a message handling thread, or an interrupt handler. To avoid a heavyweight implementation, an implementation may collapse all shared objects on a given processor into the same handling thread. This is the approach taken in this thesis, see chapter 4.

Shared data: Items of shared data model language-level items of shared data. They have the following characteristics:

- Globally-static, locally-dynamic. Individual shared data items cannot be created or deleted (globally-static), but they may grow or shrink arbitrarily according to the restrictions of the language (locally-dynamic). For example, a shared data item can implement a dynamically-sized array (but note the later restriction about memory spaces).
- Typed. Shared data items are typed using the same type system as data items from the source programming language. An array or other compound type creates a single shared data item.
- Located in a single physical address space. The shared data item cannot span multiple physical address spaces, either initially or after expansion.
- Mapped to memory spaces. Shared data items are ultimately mapped to (and therefore placed in) the memory spaces of the target architecture.

Note that shared data items are a restricted case of shared objects, effectively a shared object with 'get' and 'set' operations that manipulate a single variable. Shared data items are provided in order to model the situations in which shared data is used without a wrapping object (such as when lock-free algorithms [105, 14, 229] are used) or if the language abstraction model does not support arbitrary shared objects (such as C).

3.4.3 Logical layer

The logical layer sits between the program layer and the target layer. The items defined at this layer are not part of the input source code and do not have a physical implementation in the target architecture. They are abstract, logical items that are responsible for defining the clusters of the clustering model and for expressing controlled dynamic behaviour in the form of thread and data migration. It is this layer which performs architectural mapping of the input code and defines a formal link between software and the hardware upon which it executes.

The layer defines two sets:

- Clusters
- Cluster targets

A cluster is as defined in the clustering model, and is used to express the coupling of the source application. Clusters are used to state that a group of concurrent objects, shared objects and data items are tightly-coupled. They are similar to Chapel locales (section 2.1.1) in that they are used to inform the toolchain and run-time of an implementation that its constituent threads and data should execute closely together. Locales are more constraining than clusters as they are closer to the implementation, making them bound to specific processors or processor layouts and requiring a flat memory space. Clusters, on the other hand, are closer to the system specification and so do not restrict implementation choices beyond the coupling (as defined in the clustering model, section 3.2) that they express.

A cluster target (CT) is an abstraction of the processing elements and memory spaces of the target layer. Informally, CTs are used to state (from knowledge of the implementation architecture) that a given group of processors and memory spaces in the target layer (section 3.4.4) are a good location for executing a cluster. CTs are distributed across multiple physical processors, so that code executing on a CT may actually execute on any of its physical processors. Similarly, data stored in a CT may be actually stored in any of its physical memory spaces.

This layer also contains the following formal mappings that are used to distribute items from the program layer onto elements of hardware in the target layer:

- $concurrentobjects \cup sharedobjects \cup shareddata \xrightarrow{assignedto} clusters$: Each concurrent object, shared object and item of shared data must be assigned to exactly one cluster, according to the coupling of the system. (Many items may be mapped into the same cluster and there may be many clusters in the system.)

$$\begin{gathered}\forall x \in (concurrentobjects \cup sharedobjects \cup shareddata)\cdot \\ \exists l \in clusters \cdot x \in l \wedge (\forall l_2 \in clusters \cdot x \in l_2 \implies l = l_2)\end{gathered}$$

  – The mappings at this level represent the level of control over architectural mapping that is afforded by the system model.
  – If the programmer places all items in the same cluster then the result is a flat programming model with a single shared memory space. Source layer items cannot be individually mapped throughout the target architecture. The cluster can still be placed throughout the system, but only as a single monolithic unit.
  – If each source level item is mapped to its own cluster then the programmer can manually place each thread, shared data item etc. throughout the target architecture.
  – Therefore, the mean cardinality of the clusters in a system represents the degree to which elements of the application can be individually mapped by the system

model. As this approaches one, more control is afforded and mapping becomes less monolithic but requires more effort from the programmer and provides less clustering information.

- $cluster \xrightarrow{executeson} CT$: Each cluster must be assigned to at least one CT to specify where in the target architecture it is executed. Multiple clusters can be mapped to a single CT.

$$\forall l \in clusters \cdot \exists t \in clustertargets \cdot l \in t$$

  - Assigning a cluster to more than one CT allows the run-time system to migrate the cluster between the CTs at run-time. When to do this is defined by the implementation.
  - Migration policies are not specified or modelled by the system model. They are provided by either the implementation or application.
  - If a cluster l is assigned to exactly one CT then the threads and shared data items of l may migrate throughout the CT but not elsewhere.

- $CT \xrightarrow{comprises} processor \cup memoryspace$: Each CT must comprise of at least one processor which has access to least one memory space from the target layer.

$$\forall t \in clustertargets \cdot \exists p \in processors, m \in memoryspaces \cdot p \in t \wedge m \in t \wedge m \in memory(p)$$

  where $memory(p)$ is the set of memory spaces in the address map of processor p.

  - Let C be the set of clusters mapped to a CT t. The union of the threads and shared objects assigned to the clusters of C are mapped to the processors assigned to t, and may migrate throughout them (according to an external migration policy).
  - Equally, the union of the shared data items assigned to the clusters of C are mapped to the memory spaces assigned to t, and may migrate throughout them.
  - A typical use of CTs is to define a target that encompasses all the cores of a multi-core processor. This allows the hardware designer to specify that the implementation toolchain may freely place threads on any of the cores.

These mappings are depicted in figure 3.5.

For the modelled system to be totally static (without migration) the following invariant must hold:

$$\Big(\forall p_1, p_2 \in processors, l \in clusters, t_1, t_2 \in CTs, x \in (concurrentobjects \cup sharedobjects) \cdot$$
$$(p_1 \in t_1 \wedge t_1 \in l \wedge x \in l) \wedge (p_2 \in t_2 \wedge t_2 \in l \wedge x \in l) \implies p_1 = p_2\Big)$$
$$\bigwedge$$
$$\Big(\forall m_1, m_2 \in memoryspaces, l \in clusters, t_1, t_2 \in CTs, d \in shareddata \cdot$$
$$(m_1 \in t_1 \wedge t_1 \in l \wedge d \in l) \wedge (m_2 \in t_2 \wedge t_2 \in l \wedge d \in l) \implies m_1 = m_2\Big)$$

Informally, this states that each thread and shared object of the system must be located on exactly one processor, and each shared data item must be located in exactly one memory

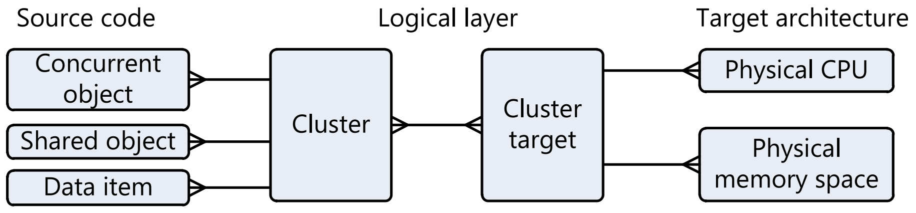

Figure 3.5: The logical layer.

space. This is true when, for every processor and shared object of the system, if l is the cluster that the item is in, the cardinality of the processors assigned to the CTs that are assigned to l must be one (and equivalently for shared data and memory spaces).

The two-layered approach of the logical layer has a number of advantages, as illustrated in figure 3.6 which gives an example of their use. In the depicted architecture, the programmer wants to express that due to the slow link in the centre of the architecture, tightly-coupled threads should execute as a group on either processors 1 and 2 *or* processors 3 and 4, but not a mixture of the two as this will require large amounts of data to be transferred over the centre link. Equally, the data items used by these threads should be stored in the appropriate memory bank. This is a common problem that has not yet been addressed by existing thread affinity sets or locale-based approaches. Clustering can tie threads and data together, but is not sufficient to express the locations on the physical architecture that are suitable mapping targets. With this model, the programmer can partition their program into clusters of interacting threads and data, and their target architecture into clusters of processors and memory spaces that can execute those clusters. Then, the logical layer mappings allow precise control of computation and data placement where it is required, and powerful cluster-based control for more dynamic areas of systems.

The second benefit to this approach is that it provides a clean separation of concerns between the application programmer and the hardware designer. The source layer and the clusters into which it is split are completely architecture-neutral. They are used to express the coupling of the software alone, and can be determined by the programmer without any knowledge of the final target architecture. Similarly, CTs are defined by the hardware engineer without any knowledge of the software that will be executed on the target layer. It is only the mapping between clusters and CTs that requires knowledge of the entire system. This is in contrast to locale and affinity -based approaches where hardware-specific mapping information is prematurely included in the software specification.

3.4.4 Target layer

The target layer provides a high-level view of the target architecture as four sets of architectural elements:

- Processing elements: The processors of the target architecture.
- Communication channels: Hardware features that transfer data between processing elements.

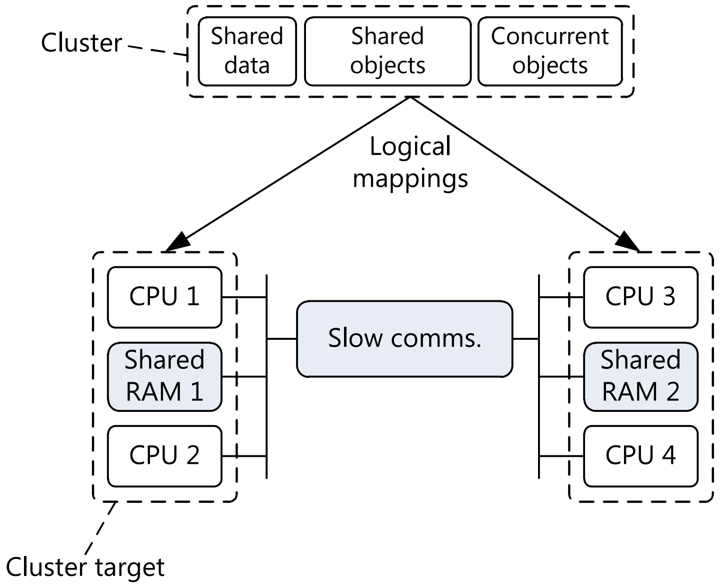

Figure 3.6: The use of clusters and cluster targets.

- Memory spaces: The distinct memory spaces in the system.
- Other hardware elements: Custom hardware elements of the target architecture (such as function accelerators) and I/O devices.

To allow code and data to be mapped to these sets, they are mapped into cluster targets by mappings that were defined in the logical layer (section 3.4.3). The architectural element sets are defined as follows:

Processing element The processing elements set models the features of the hardware whose behaviour is controlled directly by the concurrent objects of the input source code. Most commonly this refers to processors and processor cores.

- Processing elements must be connected to at least one memory space that is used for code and data storage. They may be connected to more.
- The specific element types that are supported are determined by the implementation and toolchain, but examples include standard processors, the individual cores of a multi-core processor, DSP cores, application-specific processing devices and soft processing cores on reconfigurable logic.
- The toolchain is responsible for compiling the code of the concurrent objects into a form that is suitable for the processing element(s) to which it is mapped.

Communication channel The set of communication channels models the data transfer mechanisms of the architecture (such as buses, on-chip networks, FIFO mailboxes, etc.). Note that this set only expresses channels that are useful for sending data between concurrent objects. Real architectures are likely to contain a number of hardware elements that might be considered communication channels, but are not directly accessible by a processor for the use of sending data to another processor. For example, cache coherency links are not modelled as channels because the programmer's code can not make use of them directly.

- Channels include at least two *endpoints*, which are connection points that are used by processors to send and receive data.
- For this reason, external I/O (which would only expose a single internal endpoint) is expressed as a custom hardware element.
- Endpoints may be buffered or unbuffered. Buffered endpoints provide flow control to ensure lossless data transfer.
- Endpoints are memory-mapped (or mapped into processor I/O space for processors that have dedicated peripheral buses) and so therefore can be accessed by the processors from driver code.

Memory spaces The memory spaces set models the memory and data storage features of the target architecture.

- Memory spaces may contain data, code or both, according to the architecture of the processing elements used. The compiled code of concurrent objects and shared objects and the bit-level representations of shared data items are stored in the memory spaces of the system.
- Memory spaces may be accessible by multiple processing elements.
- Caches are not described as a separate memory space because their use is transparent to software and the processor (although not to analysis which must still consider their presence). Any memory space may be cached, but this is a feature of the connection between a processing element and a memory space rather than just of the memory. The exact way that caching is specified is implementation-specific, see section 4.2.

Other hardware The other hardware elements set models features of the target architecture that are accessible from the processing elements of the system but are not expressed by the previous three sets. This includes I/O devices, which are required for interfacing with the outside world, and application-specific hardware such as function accelerators, radio transceivers or real-time clocks.

- Similar to the way that channels are connected to a processing element via endpoints, hardware elements are connected to processing elements via *ports*.
- A hardware element must have at least one port.
- Like channel endpoints, ports are either memory-mapped or I/O space mapped so that they can be manipulated by the processing elements of the system.
- Ports can be either synchronous or asynchronous, and different ports of the same hardware element can be of different types. Synchronous ports provide blocking read and write semantics, which means that the processor accessing it must wait for the operation to complete before it continues with its execution. Asynchronous ports provide non-blocking semantics and allow the processor to trigger an action which will complete later.

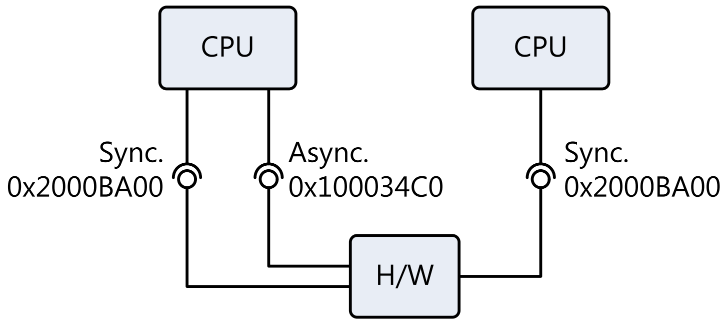

Figure 3.7: Ports describe the connection between a hardware device and a processing element.

Hardware type	**Port configuration**
Std. function accelerator	1 sync. port
Mailbox	1 sync. for sending, 1 async. for receiving
CAN Network	n async. ports
CSP channel	2 sync. ports

Figure 3.8: How external hardware is modelled by the target layer.

- The use of asynchronous ports requires that the hardware element can interrupt the processing element to which the port is connected.
- A single hardware device may have ports that are connected to many different processors, or multiple ports that connect to the same processor. Ports are not a physical feature of the target architecture, they are a logical abstraction of the ways in which the hardware can be accessed, as shown in figure 3.7.

Examples of how some kinds of external hardware can be represented in this model are shown in figure 3.8.

3.4.5 Expressing existing languages

In order to demonstrate that the presented system model is suitable for expressing the structural concepts of existing embedded development languages, this section will detail how programs written in three languages commonly used in embedded systems (C, Ada, and Java) can be represented. This is done by showing how the concurrency, coordination, and data sharing features of each language are represented by the sets of the program layer. Implementation details are not considered in this section. For a full description of a C-based implementation see chapter 4.

Note that this section concentrates on how existing languages are expressed in terms of the CTV source model. Due to the fact that none of these three languages provide explicit clustering semantics, the clustering of the logical layer is not demonstrated. In an actual implementation, this missing clustering information may be obtained directly from the programmer (using pragmas or an external language), inferred from the program structure, or from automatic profiling and analysis.

For each language mapping, the set of shared objects is extended to include implicit 'thread helper' shared objects, one for each thread in the application. These helper objects are exactly the same as standard shared objects and are used to implement the direct inter-thread communications of the chosen language. Shared objects are used for this (rather than extending the semantics of concurrent objects) because different source languages demonstrate different inter-thread communication models. By using implicit shared objects the system model can remain constant and does not have to be redefined for each language.

3.4.6 Expressing C with the system model

Due to the fact that parallelism is not a part of the C language, the combination of C and the POSIX pthreads library is discussed here.

- **Concurrent objects:** pthreads are represented using concurrent objects. A concurrent object is used to model each instance of the `pthread_t` type, and one to model the `main` thread (the application entry point). Concurrent objects execute immediately after creation. pthreads, on the other hand, can be declared but are not scheduled for execution until the `main` thread makes the appropriate `pthread_create` call to start the thread in question. Therefore, concurrent objects are created by `pthread_create`, rather than by the declaration of a `pthread_t` instance. Similarly `pthread_exit` causes the concurrent object to be destroyed.
- **Shared objects:** C does not support the structured programming constructs necessary to allow the programmer to define their own shared objects. Consequentially, coordination between threads is provided by pthreads' mutexes and condition variables. Each instance of the `pthread_mutex_t` or `pthread_cond_t` types are represented by a shared object. Mutexes are created and destroyed by calls to `pthread_mutex_init` and `pthread_mutex_destroy`. Condition variables are created and destroyed by calls to `pthread_cond_init` and `pthread_cond_destroy`. The operations and state provided by mutexes and condition variable shared objects are defined by the pthreads standard, but the common operations are listed below for reference:
  - **Mutexes:**
    - Created by: `pthread_mutex_init`
    - Destroyed by: `pthread_mutex_destroy`
    - Interface:
      - `pthread_mutex_lock`
      - `pthread_mutex_timedlock`
      - `pthread_mutex_trylock`
      - `pthread_mutex_unlock`
    - State:

        - Current state of the mutex (either ‘free’, or which pthread holds the mutex)
        - Queue of pthreads currently waiting on the mutex.
  - **Condition variables:**
    - Created by: `pthread_cond_init`
    - Destroyed by: `pthread_cond_destroy`
    - Interface:
      - `pthread_cond_wait`
      - `pthread_cond_timedwait`
      - `pthread_cond_signal`
      - `pthread_cond_broadcast`
    - State:
      - Queue of pthreads currently waiting on the condition.

  For both types, mutual exclusion over all of their exported functions is required so they all acquire the shared object’s write lock. This ensures that, for example, concurrent calls to `pthread_mutex_lock` cannot result in two pthreads receiving the mutex.

  Shared objects are also used to model threads joining and querying the status of other threads. For each thread in the system there is also a corresponding ‘thread helper’ shared object with the following properties that can be accessed by other system threads:

  - **Thread helpers:**
    - Created by: `pthread_create`
    - Destroyed by: `pthread_exit`
    - Interface:
      - `pthread_join`
    - State:
      - Current state of the thread (created, running, terminated, etc.)
      - Queue of threads waiting to join this thread.
- **Shared data:** As shared variables in C lack any form of auxiliary semantics (locking, coherency, etc.), they can be directly represented by data items in the system model.

Shared data can only be passed between C pthreads in one of two ways: global variables or pointers. For a variable to be in scope of two threads it must be declared in an enclosing scope. However, C does not allow nested functions, so the only scope above a thread body is global scope. Alternatively, scoping can be bypassed by creating a reference to a shared variable and passing it into the thread when it is created. Due to the complexity of arbitrary pointer usage, an implementation may need to restrict the use of pointers to an analysable subset.

An important issue to be considered is that pthreads semantics are not provided by the language compiler or runtime, but by the pthreads library functions, which are themselves calls into a kernel executing on the target processor. The kernel is required to provide thread dispatching, scheduling, etc. As a result, using C with pthreads introduces the requirement that the implementation include either a full embedded operating system or a microkernel that implements a subset of the POSIX standard.

The following C code shows the standard producer / consumer pattern implemented in pthreads.

```
#include <pthread.h>
#include <queues.h>

queue_t the_queue;
pthread_cond_t condvar;

void* producer_body(void*);
void* consumer_body(void*);

int main(void) {
   pthread_t prod_thread;
   pthread_t cons_thread;

   pthread_mutex_t shared_mutex;

   pthread_cond_init(condvar, NULL);
   pthread_mutex_init(mutex, NULL);

   //Create the threads, pass the mutex to them
   //Error handling not shown
   pthread_create(&prod_thread, NULL, producer_body, (void*)shared_mutex);
   pthread_create(&cons_thread, NULL, consumer_body, (void*)shared_mutex);

   pthread_join(prod_thread, NULL);
   pthread_join(cons_thread, NULL);

   return 0; //Success
}

void* producer_body(void* mux) {
   pthread_mutex_lock((pthread_mutex_t*)mux);
   //...Check if queue is not full...
   //...Add an item to the_queue...
   pthread_cond_signal(condvar);
   pthread_mutex_unlock((pthread_mutex_t*)mux);
}

void* consumer_body(void* mux) {
   pthread_mutex_lock((pthread_mutex_t*)mux);
   pthread_cond_wait(condvar, (pthread_mutex_t*)mux);
   //Process an item from the queue
   pthread_mutex_unlock((pthread_mutex_t*)mux);
}
```

This code is modelled using the following system model shown in figure 3.9. There are three threads in the example code, the `main` thread that is the application entry point, and two created threads. These are all modelled as concurrent objects. Two explicit shared objects are used, a mutex and a condition variable. These are the variables `pthread_cond_t condvar` and `pthread_mutex_t shared_mutex`. Other variables are modelled as shared data items, `queue_t the_queue` in this case. It can be seen that whilst instances of `pthread_cond_t`, `pthread_mutex_t` and `pthread_t` can be viewed as shared data, they are really only used as identifiers that are passed to the pthreads API. Accordingly they are treated specially, as

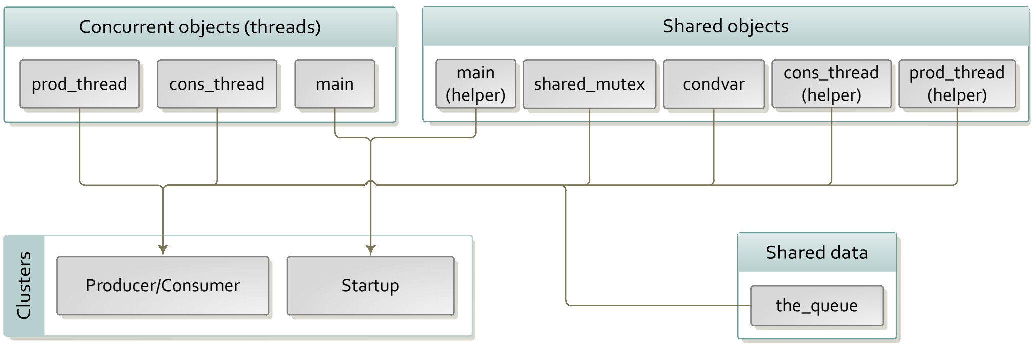

Figure 3.9: A suitable system model representation of the C example.

shared objects.

Also, three implicit thread helper shared objects exit for the three threads of the system. The only shared data used is an instance of a queue datatype that is used by the producer and consumer threads. Note that the entire datatype is represented by a single shared data item, even though it is internally represented by a compound structure type with many constituent primitive types.

Also shown in figure 3.9 is an appropriate clustering of the application. As the majority of the communication and data transfers are between the producer and consumer threads, and these threads also primarily use the shared data, mutex and condition variable, all of these constructs are assigned to a single cluster. The main thread, however, has low communication requirements as it simply has to initialise the system then wait for the other threads to complete. Accordingly, it is assigned to a separate cluster.

The system model presented in figure 3.9 captures code structure (in the concurrent objects, shared objects and shared data) and captures the application's coupling (through clustering) but the programming model used is still that of the target language - in this case C with pthreads. As discussed previously, the C programming model assumes a single processor architecture with a uniform address space so the program is limited to implementation on such a platform. For example, it cannot be distributed so that the two clusters are located on different processors with physically-separate memory without considerable rewriting and architecture-specific coding. This problem is not limited to C, and is the case with the following two languages presented in the next two sections. The work presented in this thesis is a potential solution to this problem.

The presented solution can be compared to a post-partitioning approach [16] in which a single input program is distributed at compile-time to multiple target programs. However, post-partitioning requires that the input program has already been partitioned into distributable units (such as by using the Ada Distributed Systems Annex [28]). This can restrict development to heavyweight partitions, limit program reuse, and it introduces architectural specification early in the software design. The work in this thesis aims to allow architectural specification to be performed at a very late stage of application development and with high amounts of code reuse. It does this by allowing programmer-directed mappings between the programming model of the source language and the target architecture through the concept of a Virtual Platform, introduced later in section 3.6.

3.4.7 Expressing Ada with the system model

Ada is a much larger language than C and provides a range of additional features that the programmer can use to express the behaviour of their application. In the context of this work, the primary difference between the two is that concurrency is a native part of Ada rather than provided through the use of an external library and OS as it is with C.

Ada provides application-level concurrency with the use of native *tasks*. Tasks are lexical constructs that have their own thread of control, may be nested, and can communicate with each other to share information and services. Ada manages the use of shared data through the provision of *protected objects* (POs). POs are passive objects that contain a private state and export a set of functions and procedures which can be called by the tasks of the system. POs are based on monitors [107] and consequentially provide automatic synchronisation over the exported functions and procedures. Only one protected procedure may be executing at any one time. Protected function calls may be executed concurrently (because Ada functions are pure functions that cannot affect the state of the PO) but never concurrently with a protected procedure call.

Tasks also provide *entries* for direct inter-task communication without the use of an intermediary PO. Entries are similar to Handel-C channels, in that they provide communication with blocking semantics. Both the sender and receiver must be ready before the communication can take place. One blocks until the other is ready. Unlike Handel-C channels, entries can provide two-way communication. An entry allows two tasks to synchronise at a specific point, transfer data, coordinate for a series of operations, and then diverge again afterwards. Entries may be guarded by a boolean expression.

Finally, tasks provide a similar range of operations to pthreads. Tasks can start, join, query, and terminate other threads. Like the way that C is modelled in the previous section, these operations are provided by an implicit shared object with an interface that is defined later.

The system model is used to model Ada programs as follows:

- **Concurrent objects:** Concurrent objects represent Ada tasks. Tasks start executing immediately once the declaring task (which may be the 'main' task) reaches its `begin` keyword. This matches the semantics of concurrent objects. Execution of task activation is strictly defined in Ada and these rules must be observed by a CTV implementation by ensuring that tasks are created at the correct time and by the runtime of the correct task.
- **Shared objects:** Shared objects represent POs. In C, the programmer can not define their own shared objects because it does not have the required constructs. As a result, C programs only require two types of explicit shared objects (mutexes and condition variables). In Ada, the programmer defines their own POs that can provide arbitrary interfaces and more complex synchronisation semantics, but the model remains the same.
  - **Protected objects:**
    - Created by: PO instantiation
    - Destroyed by: PO leaving scope / explicit deallocation
    - Interface: The interface of the shared object contains all protected procedures, protected functions, and entries of the PO. In accordance with Ada's rules, pro-

tected functions will attempt to obtain the shared object's read lock while protected procedures will attempt to obtain the shared object's write lock. Entries will attempt to obtain the object's write lock, and may also be guarded by a boolean expression which is modelled using shared object guards where possible.

    * State:
        - All local variables of the PO. Note that due to the fact that Ada allows arbitrary nesting of blocks, POs may declare internal tasks or other POs. These items can be considered part of the state of the PO, but they must still be modelled using concurrent objects and shared objects respectively. See below for more notes on nesting versus clustering.
        - A queue of tasks waiting on each entry.

Implicit 'task helper' shared objects are created along with each task / concurrent object to allow the modelling of the task rendezvous and other directly inter-task operations. Essentially, the helpers are modelling the parts of the Ada run-time that are used to implement inter-task communications. CTV's system model does not allow direct communication between concurrent objects, so this functionality is provided by helpers. The advantage of doing this is that such services can then be sensibly mapped onto the communication channels of the target architecture. If they were part of the concurrent object then this would not be possible.

  - **Task helpers:**
    * Created by: Task creation
    * Destroyed by: Task completion / termination
    * Interface:
        - This section uses 'associated task' to refer to the task that this task helper is associated with. 'Remote task' refers to any other task of the system that may be interacting with the associated task.
        - `accept`: Called by the associated task to accept an entry. The task will block until a corresponding `entry` call is made by a remote task.
        - `entry`: Called by a remote task to call an entry. The remote task will block until a corresponding `accept` call is made by the associated task.
        - `selectaccept`: Called by the associated task to selectively accept an set of entries.
        - `selectentry`: Called by a remote task to indicate it is selectively calling an entry of the associated task. The entry call may not complete if the remote task takes a different branch in the select.
        - `terminate`: Called by the associated task to terminate itself.
        - `abort`: Called by a remote task to forcibly abort this task.
        - `requeue`: Called by a remote task to requeue itself on a given entry.
    * State:
        - Current state of the task (Unactivated, Runnable, Sleep, Terminated)
        - A queue of tasks waiting on each entry.

- **Shared data:** Shared data items are used to represent variables that are shared between tasks but that are not wrapped inside the state of a PO. Recall that these items provide no synchronisation.

Ada is a nested block-structured language. A function body may, for example, declare other functions, procedures, tasks and protected objects. The declared items may themselves also declare child items. Concurrent Ada programs are frequently written using subtasks within tasks because the run-time features of the tasking model can be relied upon to provide automatic synchronisation and coordination. A parent task does not complete until all child tasks have completed.

It is important not to confuse the nesting of the input program with the clustering of the logical layer. Nesting affects the semantics of the program (as shown in the example above) and is used by the programmer for scoping, encapsulation, and expressibility. Clustering does not affect any of these issues and is instead additional information about the *intent* of the programmer and they way in which the code might be best implemented. The two may be related in some programs, but contradictory in others. Clustering could be automatically derived from the nesting of the program, but this is outside the scope of this thesis.

```
procedure ProdCons is
   task BufferTask is
      entry Append(x: in Item);
      entry Remove(x: out Item);
   end BufferTask;

   task body Buffer is
      Count : Integer = 0;
   begin
      loop
         select
            when Count < MaxSize => accept Append(x : in Item) do
               --Store x in buffer
            end Append;
         or
            when Count > 0 => accept Remove(x : out Item) do
               --Set x to the next item
               --Remove it from the buffer
            end Remove;
         end select;
      end loop;
   end Buffer;

   task Producer;
   task body Producer is
   begin
      for i in 0..10 loop
         --Produce an item
         Buffer.Append(theitem);
      end loop;
   end Producer;

   task Consumer;
   task body Consumer is
   begin
      for i in 0..10 loop
         Buffer.Remove(theitem);
         --Consume the item
      end loop;
      abort Buffer;
   end Consumer;

begin
   --The above tasks will be created and started when this procedure is elaborated.
   --The procedure will exit when the tasks exit.
end ProdCons;
```

This example shows an implementation of the producer / consumer pattern in Ada using tasks and the task rendezvous to pass data. The example has four tasks in total, the `Producer` task, the `Consumer` task, the `Buffer` task, and the main task which is executing the procedure `ProdCons`. `Producer`, `Consumer` and `Buffer` are all created when the body of `ProdCons` is elaborated by the main task and the procedure will not exit until all the internal tasks have completed. No explicit shared objects are used, but there are four implicit task helper objects, one for each task.

The interfaces exposed by these objects are defined above. In the example, the `Producer` task and the `Consumer` task both call an entry of the `Buffer` task. This is modelled as the `Buffer` concurrent object calling the *accept* method of its task helper shared object, and the `Producer` and `Consumer` concurrent objects calling their `entry` method. `Buffer` exports two entries which are guarded by simple boolean guards based on an internal state variable `count`. These guards are evaluated by the task helper and the entry calls processed accordingly. The guards are reevaluated after each entry call is completed. The `Consumer` task calls the `abort` procedure of `Buffer`'s task helper to halt its execution once the program is complete. The tasks are all nested inside the `ProdCons` procedure, however, this does not require them to be placed in the same cluster.

The previous example uses a ‘buffer’ task to pass data between the producer and consumer tasks. Since Ada 95 a more common way of doing this is to use a protected object, as shown in the followng example.

```
procedure ProdCons2 is
   protected Buffer is
      entry Append(x : in Item);
      entry Remove(x : out Item);
   private
      Count : Integer := 0;
   end Buffer;

   protected body Buffer is
      entry Append(x : in Item) when Count < MaxItems is
      begin
         --Add x to the buffer
      end Append;

      entry Remove(x : out Item) when Count > 0 is
      begin
         --Remove an item from the buffer
         return theitem;
      end Remove;
   end Buffer;
   task Producer;
   task body Producer is
   begin
      for i in 0..10 loop
         --Produce an item
         Buffer.Append(theitem);
      end loop;
   end Producer;

   task Consumer;
   task body Consumer is
   begin
      for i in 0..10 loop
         Buffer.Remove(theitem);
         --Consume the item
      end loop;
   end Consumer;
begin
   --The above tasks will be created and started when this procedure is elaborated.
   --The procedure will exit when the tasks exit.
end ProdCons2;
```

This version of the code is modelled in the system model using three tasks, `Producer`, `Consumer` and the main task. It uses four shared objects, the explicit `Buffer` object (which is now a protected object) and three implicit task helper objects. The `Buffer` shared object has the following interface:

- entry Append: Receives a single item, obtains the object's write lock, guarded by the value of `Count`.
- entry Remove: Takes no parameters, returns a single item, obtains the object's write lock, guarded by the value of `Count`.

Use of the write lock ensures that only one entry can be executed at any one time to match the programming model of Ada's protected objects.

3.4.8 Expressing Java with the system model

As a highly object-oriented language, Java is differentiated from the previous two languages in that truly raw data is not present in a Java program. Since JSR-201 in 2004 [29], all primitive types are automatically 'boxed' by the compiler. This means that an integer literal may be declared using the `int` keyword, but this will be converted to an instantiation of the `Integer` class, which is a wrapper object for a native integer. This is important because rather than simply being raw data, instances of the class `Integer` contain methods that can be called by threads in the system. An example of this is the `Integer.toString` method which converts the boxed native integer into a string object. In order to capture these methods, a Java program when represented using the system model does not use raw shared data at all. Instead all of its shared data is represented as shared objects because shared objects can include arbitrary methods.

- **Concurrent objects:** Concurrent objects are used to model the threads used in the Java program. In Java, threads can be created in two ways, either by implementing the `Runnable` interface and using the `Thread` constructor, or by directly subclassing thread. In either case, a thread object is created. Execution of the thread constructor triggers the creation of the concurrent object. The thread will not actually begin executing until its `start` method is called to remain consistent with Java. (Thread constructors are executed by the creating thread as normal.) The concurrent object is destroyed either when the thread completes its `run` method, when the thread exits due to an unhandled exception, or when the entire application completes (in the case of a daemon thread). As with the previous two languages, a helper shared object is created along with each concurrent object to allow other threads to interact with it and join it.
- **Shared objects:** Shared objects are used to represent all the passive elements of a Java application. Commonly this makes up the vast majority of the code of the system. A shared object is an instance of a class that can be accessed by the concurrent objects (threads) of the system. Normal Java programming style can lead to the use of a large number of such object instances, hence its reliance on a garbage collector, (although embedded Java programming is traditionally more conservative). The large number of shared objects that Java uses means that manually mapping these to the target architecture is likely to become an onerous task for the programmer. Mapping will need to be partially automated, for example if an instance is assigned to a given processor then all of its contained instances are also mapped to the same processor. This is outside the scope of this thesis.
  - **Object Instances:**
    - Created by: Class instantiation.
    - Destroyed by: Instance's reference counter reaches zero.
    - Interface:
      - The interface of the shared object contains all non-static methods of the instance. Methods declared as `synchronized` automatically acquire the shared object's write lock before commencing, the read lock is not used. Guards are not required as they cannot be expressed in Java.
      - An implicit `_synchronize()` method is provided to allow non-synchronised code to acquire the object's write lock through the use of Java's `synchronize` construct.
      - The `Object` class defines three methods that are used for thread synchronisation that require special attention - `wait()`, `notify()` and `notifyAll()`. When a thread calls the `wait` method of an object it is held suspended in a queue until the object's `notify()` or `notifyAll()` methods are called. This is identical to the condition variable model that is used in pthreads (discussed in the previous section). These three methods are exposed by every shared object.
    - State:
      - All local variables of the instance. As with Ada, nested instances can be considered part of the state of the shared object, but they must still be appropriately modelled using concurrent objects and shared objects.

          · A queue of threads waiting on the object.

  Implicit 'thread helper' shared objects are created along with each thread / concurrent object to allow direct inter-thread operations. The interface of this shared object is as follows:

  - **Thread helpers:**
    - Created by: Thread creation (calling the `Thread.start` method)
    - Destroyed by: Task completion (thread exits its `run` method or encounters an unhandled exception)
    - Interface:
      - `join`: Allows another thread to wait for this thread to complete execution.
      - `interrupt`: Raise an `InterruptedException` in the target thread.
      - `setPriorty / getPriorty`: Manage the thread's priority. This might not have any effect if the thread is mapped solely to a given processor, but if it shared its processor then scheduling will need to be considered by the implementation.
    - State:
      - Current state of the thread (new, runnable, waiting, terminated, blocked, timed_waiting) and its priority.
      - A queues of tasks waiting to join this thread.

- **Shared data:** As previously mentioned, Java's 'autoboxing' prevents the use of raw data in a Java program. The programmer uses only shared objects.

The following example shows a producer / consumer pattern in Java that uses the `wait` and `notify` methods to implement thread synchronisation.

```
public class ProdCons {

 protected LinkedList list = new LinkedList();

 class Producer extends Thread {
  public void run() {
   while (true) {

    //... Produce an object ...

    synchronized(list) {
       while (list.size() == MAX)
         list.wait();

     list.addFirst(justProduced);
     list.notifyAll();
    }

    if (done) break;
   }
  }
 }
 class Consumer extends Thread {
   public void run() {
     while (true) {
       Object obj = null;

       synchronized(list) {
         while (list.size() == 0) {
            list.wait();
         }
         obj = list.removeLast();
         list.notifyAll();

         if (done) break;
       }

       //... Consume the object ...

     }
   }
 }
}
```

In this example there are three concurrent objects, the `Producer` thread, the `Consumer` thread, and the main thread. Five shared objects are used, two implicit objects for the `ProdCons` instance and the `list` instance, and three implicit thread helper objects. Of these, the `list` instance object has the most work to do. Both threads call its `wait()` and `notifyAll()` methods to enqueue themselves and release each other. `list` is responsible for maintaining these queues and waking up the appropriate thread when called to do so. Both threads also call `list`'s `_synchronize()` method to manually obtain the object's write lock and ensure mutual exclusion.

Another implementation of this pattern may choose to subclass the `LinkedList` class and extend it with synchronised 'add item' and 'remove item' methods. Threads can then call these methods directly without having to worry about obtaining locks. The implementation for this is identical, however. Instead of threads calling `_synchronize()` first and then interacting with the list as in the example above, the synchronised methods call `_synchronize()` to obtain the lock automatically. The rest of the code remains the same.

3.5 Compile-Time Virtualisation and the Virtual Platform

The analysis of modern embedded development in chapter 2 demonstrated that existing programming languages provide unsuitable abstraction models when developing code for deployment on complex embedded architectures. The system model introduced at the start of this chapter rectifies this by introducing more appropriate, hardware-targeted abstractions, but it cannot be used without extra support from the language to define clusters, cluster targets, the mappings between source language elements, logical elements, and the target architecture. This could be achieved by the development of a new language and compiler, but this is undesirable. Instead, the work in this thesis presents a way of using the system model with existing languages and toolchains, thereby maximising code reuse and aiding industrial acceptance.

Compile-Time Virtualisation (CTV) is a technique developed and presented in this thesis that uses a virtualisation-based approach to simplify the targeting of code to embedded architectures. CTV supports the clustering model of section 3.2 and the general system model of section 3.4, thereby providing support for the creation of scalable, dynamic, heterogeneous systems. It allows the programmer to continue working with existing embedded programming languages, specifically C, Ada and Real-Time Java, by providing greater control over the software implementation and allowing tighter, more efficient mappings to non-standard architectures.

CTV replaces the existing layers of virtualisation and abstraction that are present in standard software development (as discussed in section 2.6.1) with a single virtualisation layer across the entire architecture, termed the *Virtual Platform* (VP), which has three main features:

Compatibility with the chosen programming model: The VP is a high-level view of the underlying hardware that presents the same programming model as that which is expected by the developer's chosen source language. For example, if the developer is working in C, the VP presents a single logical address space with uniform inter-thread communication and uniform cache coherency. The actual target hardware may be very different. This means that the VP has to distribute code and data throughout the system, and to handle communications transparently. Issues such as caching and coherency must also be considered. In essence, as with standard run-time virtualisation the layer is tasked with ensuring that the programmer's code operates correctly without low-level programmer intervention. Because the layer hides low-level implementation details it allows for code to be architecturally-neutral, as the developer does not need to break the abstraction models of the language.

Flexible mappings from the virtual architecture to actual hardware: Every system based on virtualisation contains a set of virtualisation mappings. These mappings place elements of the software and virtual hardware onto the actual physical hardware. (For example, threads → processors, variables → memory spaces etc.) In a standard run-time virtual machine, these virtualisation mappings are implemented by a run-time system and are both fixed and largely opaque to the programmer. In CTV, the mappings are made flexible, allowing the programmer to use their application-specific knowledge to *influence* the implementation of the code and achieve a better mapping onto the target architecture. For example, the designer can choose to place threads that frequently communicate onto processors that are physically close to each other, or to place global data in memory spaces that are close to where its it going to be needed to minimise copying. The first feature of the VP, 'compatibility with the chosen programming model', ensures that despite how the programmer adjusts these mappings the software will still operate correctly, only its non-functional properties will be affected.

Visibility of custom hardware elements: Custom hardware elements are exported up to the programmer through the VP at design-time and presented in a form that is consistent with the source language's programming model. This allows these elements to be effectively exploited without extra development effort and in a manner that is consistent with the current programming model. For example, function accelerators can be exposed to the programmer as standard functions that are called like a library routine. The virtualisation system has enough information to handle marshaling of data, synchronisation, data

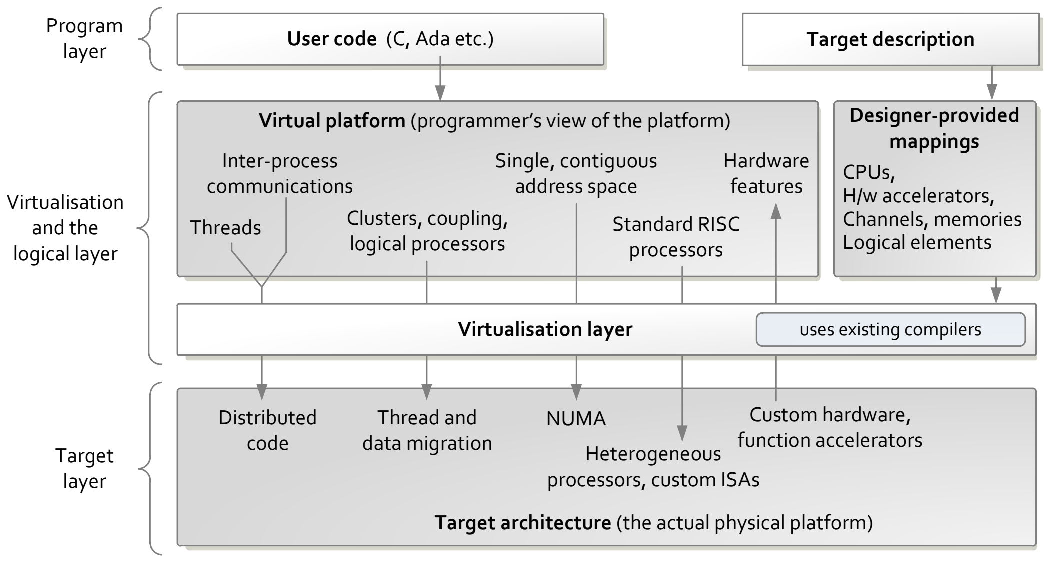
Program layer
User code (C, Ada etc.)
Target description
Virtualisation and the logical layer
Virtual platform (programmer's view of the platform)
Inter-process communications
Single, contiguous address space
Hardware features
Threads
Clusters, coupling, logical processors
Standard RISC processors
Designer-provided mappings
CPUs, H/w accelerators, Channels, memories Logical elements
Virtualisation layer
uses existing compilers
Target layer
Distributed code
Thread and data migration
NUMA
Heterogeneous processors, custom ISAs
Custom hardware, function accelerators
Target architecture (the actual physical platform)

Figure 3.10: An overview of CTV, showing how it relates to the layers of the system model (figure 3.4)

copying issues etc. Programmer intervention is not required. Note that legacy code written without knowledge of custom hardware is not refactored to make use of it. This problem is orthogonal to CTV and developments in the area of automatic hardware mapping could be included, but this is outside the scope of this thesis.

An overview of the CTV system and how it relates to the system model presented earlier in this chapter is shown in figure 3.10. CTV assumes that the mappings of program layer elements to target layer elements are performed manually by the programmer. Automatic mapping is related to the field of co-design and is an orthogonal problem, so is outside the scope of this thesis. Also, CTV does not perform auto-parallelisation of the input code. Examples of auto-parallelising compilers are described in section 2.1.1, and this thesis assumes that the input code is expressed in a sufficiently parallel way such that it can be appropriately mapped to the target architecture. Automatic parallelisation techniques can be used in conjunction with CTV, but this is not discussed.

CTV does not duplicate the work of system design languages like SystemC (presented in section 2.4.3). The design method used by these languages requires the designer to begin at a high level of abstraction and iteratively refine the design until it is of a sufficiently low level to be implementable, and they tend to allow the generation of both hardware and software. CTV instead focusses on the generation of software for a pre-existing architecture by giving the programmer influence over implementation choices that are normally made by the compiler. Nonetheless, because the CTV toolchain is given a large amount of architectural information it can be used to generate the target architecture at the same time as the compiled code (as discussed in appendix C) making it useful as part of a hardware/software co-design framework.

The Virtual Platform of CTV is so named because it is a true virtualisation layer that sits above the actual target platform. The term should not be confused with virtual platforms that are

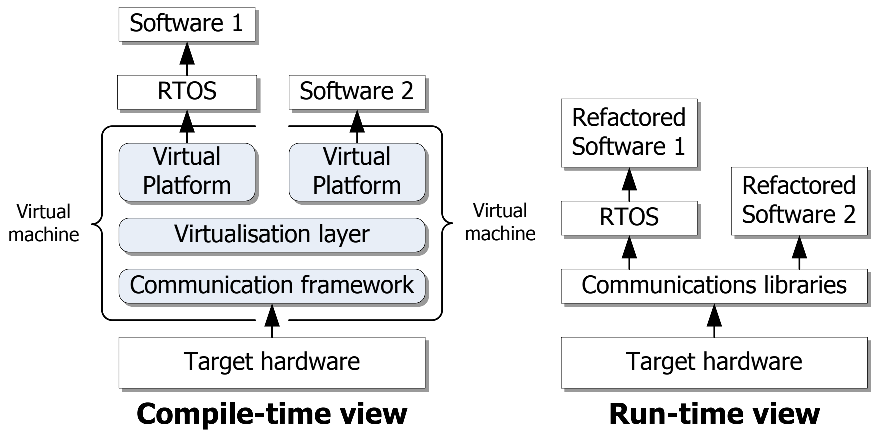

Figure 3.11: How the compile-time view of CTV differs from the run-time view.

used primarily for verification, such as the CoWare Virtual Platform [54, 230], discussed in section 2.3.4. CTV's VP provides a virtualisation layer that increases the abstraction level of the target hardware, thereby allowing for architecture-agnostic software development. CTV's VP is an integral part of the programming model. CoWare's VP uses virtualisation in a different sense. It aims to reproduce the target hardware inside a simulation host, aiding debugging by allowing the designer to simulate their design with greater transparency. CoWare's VP does not change the programming model that the programmer must use, and does not hide any of the implementation complexities of non-standard target hardware.

3.5.1 Moving from run-time to compile-time

As its name implies, Compile-Time Virtualisation is differentiated from run-time virtualisation systems by the fact that its virtualisation layer only exists during compilation. The lack of run-time virtualisation means that overheads are reduced to a minimum, which is essential when developing for resource-constrained embedded systems. Specifically, the system does not have to store code for an interpreter or run-time support system, and each instruction of the compiled program executes natively on the target hardware. This is illustrated in figure 3.11.

To understand the semantic difference between run-time and compile-time virtualisation it is helpful to first consider Java, an archetypal example of run-time virtualisation. Java's Virtual Machine (the JVM) interprets the compiled bytecodes of the user's program. The JVM conceptually sits as a layer between the output of the compiler and the capabilities of the target processor, interpreting the compiler output so that it can be understood by the processor and executed. The JVM has the effect of making it appear to the programmer as if the processor is capable of executing Java bytecodes, where in reality it cannot. In other words, because the virtualisation exists at run-time, it extends the *run-time* capabilities of the system.

In contrast, CTV's virtualisation exists at compile-time which allows it to instead extend the *compile-time* capabilities of the system. By sitting as a layer between the user's uncompiled code and an unmodified target compiler, CTV makes it appear to the programmer as if a given compiler can efficiently target a range of architectures that it normally cannot without extending the input language. This allows CTV to solve one of the main problems identified in chapter 2 - that hardware architectures change too rapidly to develop new languages and toolchains for each new technology that appears. Indeed, Anvil (the example implementation of CTV presented in chapter 4) uses unextended ANSI C and an unmodified version of the `gcc` compiler yet it allows the programmer to effectively target complex hardware from the source language level that would normally require the use of hand-coded assembly, custom link scripts, and other abstraction-breaking techniques.

CTV is a way of extending the capabilities of a pre-existing language and compiler in a controlled way to meet changing future demands. This vastly helps code reuse, because existing code is not made obsolete when a new architectural paradigm is developed that would previously have required a new language or language extension. Instead, the capabilities of the virtualisation layer are expanded. Also, all users of the base compiler benefit when it is improved, unlike the current situation where each language extension uses a fork of the original compiler. New features can spread to all uses much more easily.

For completeness it should be observed that run-time virtualisation and CTV are orthogonal techniques, and it is possible to combine both in the same design process. However, the benefits of doing so in a resource-constrained embedded systems are limited as both techniques attempt to solve similar problems but from different starting points.

3.6 The CTV system model

3.6.1 Model overview and rationale

The CTV system model implements the system model presented in section 3.4, and is shown in figure 3.12. Source programs are represented as sets of concurrent objects, shared objects, and shared data items. Target architectures are represented as sets of processors, communication channels, memory spaces and unique hardware elements. The source program is grouped into a set of clusters that define the coupling of the application, and the target architecture is grouped into a set of cluster targets, which are sites that are particularly suitable for the implementation of a cluster. The semantics and definitions of these sets are all identical to the previously presented system model.

CTV's system model, however, deviates from the general model in two important ways. The first is the introduction of the VP layer between the program and target layers. The VP layer is responsible for the distribution of the program layer over the target architecture and is more implementation-focussed than the purely abstract system model previously presented. The VP implements the clustering and mapping of the logical layer, and contains a model for the integration of unique hardware elements (function accelerators etc.) into the programming model of the system. The exact requirements of the VP are enumerated and discussed in section 3.6.2. To operate, the VP defines a novel communications architecture called the Object Manager model, which is discussed in section 3.7.

The second deviation from the general system model is that due to the stated focus on the development of embedded software, the CTV system model is compile-time static. This means that the following restrictions are applied to the system:

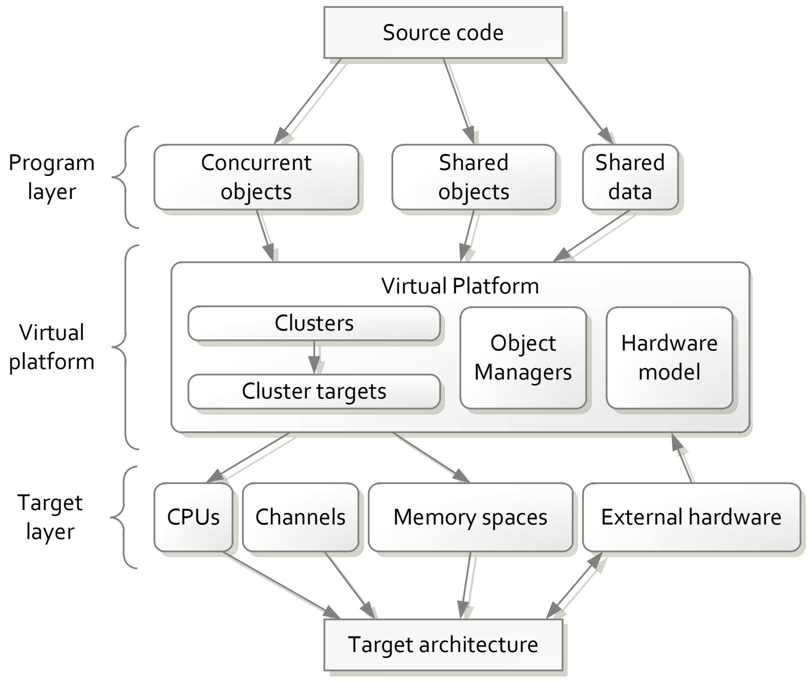

Figure 3.12: CTV's system model shows the introduction of the Virtual Platform

- Static program layer. Concurrent objects, shared objects and shared data items cannot be created or destroyed at run-time. Essentially, it must be possible to statically analyse the input code to determine all the instances that are required. Individual shared data items (like a linked list structure) may still grow or shrink dynamically, although they are restricted to reside entirely in a single memory space. Concurrent objects can also be arbitrarily started and stopped.
- Static logical layer. The clusters and cluster targets of the logical layer must be defined when the application is programmed and when the architecture is designed respectively. They cannot be adjusted as the program is executing.
- Static target layer. The target architecture cannot arbitrarily change at run-time. Existing elements may be powered on and off at any point (due to power scaling or faults) but, in general, new hardware cannot be introduced.
- Static clustering. The assignment of program layer objects to clusters and target layer objects to cluster targets must remain fixed throughout the execution of the program.
- Static mappings. The mappings between clusters and cluster target must remain fixed.

This restricts the run-time behaviour of the applications represented by this model, but in practice such restrictions are often placed on embedded applications in order to preserve timing behaviour, allow correctness analysis, or to guarantee memory and power constraints are met. The static system model does not result in a static system at run-time. The logical layer still permits the use of a large amount of run-time dynamic behaviour in the system, but that behaviour is first specified at compile-time. Also, for truly dynamic applications that require non-analysable creation of threads or shared data items these restrictions can be relaxed in implementations of the model. One example of when this is done is discussed in section 4.12, which shows the use of dynamic shared memory structures in a CTV implementation.

The justification for the static restriction is as follows. Standard development processes for C, Ada and Java provide a highly-permissive system model that can allow a large amount of dynamic behaviour. This dynamism can be broadly categorised into dynamism of the application specification and dynamism of the application implementation - examples of which are as follows:

- **Specification dynamism:** Run-time changes to the specification of the application:
  - Control: Run-time variation of the threads of control in the system. e.g. dynamic native thread creation (Ada, Java) or threading provided out of the scope of the language (C with pthreads). Fine-grained parallelism constructs, such as the `par` statement of Occam and Handel-C (see section 2.4.2) do not in general introduce dynamism as they are static language features.
  - Data: Run-time variation of the data shares of the system. e.g. Run-time memory allocation (`malloc` of C, `new` of Ada and Java) and then passing a reference to the newly-created data (using pointers in C, access types in Ada or object references in Java).
  - Structural: Computational reflection [203, 206, 60] allows the programmer to change the structure of the program (including defined functions, class methods, declared variables etc.) without changing the source code. e.g. `java.lang.reflect` in Java or the reflective capabilities of most interpreted languages such as Smalltalk, Python, Ruby and many others.
  - Semantic: Reflection can also extend to allow the programmer to alter the semantics of the application at run-time by reflecting on the language interpreter or compiler. e.g. Smalltalk and Lisp.
- **Implementation dynamism:** Run-time changes to the way in which the application is implemented on a given architecture:
  - Mapping: The physical locality of control and data in the system (threads → processors, data → memory spaces). e.g. Dynamic thread migration over multicore architectures (implemented in most major OS kernels for load-balancing and fault tolerance and by some hardware architectures such as hyperthreading processors [159]). Also includes the use of dynamic memory architectures [111].
  - Recompilation and just-in-time compilation: Recompiling applications due to run-time profiling information, ‘hotspotting’ and adaptive optimisation [10], or to make use of dynamic processor capabilities (see next).
  - Dynamic hardware: Features of the hardware that may change at run-time due to the behaviour or requirements of the application. FPGA-based partial dynamic reconfiguration (see section 2.5.6) can introduce additional hardware as required [226] or dynamically change the features of processors according to run-time profiling [154].

No existing language models allow the programmer to explicitly delimit their use of specification dynamism to within a sub-partition of the whole application. Specific language subsets can be defined that exclude undesirable features (e.g. the Ravenscar subsets of Ada [30] and Java [140]), but these subsets remove the features completely from the entire application (and may require new compilers or tools to check compliance). This is a problem because an embedded programmer will generally avoid the use of dynamism, apart from in non-critical areas of the system where its use greatly simplifies software development. However, because language run-time systems must always be built to support the dynamic features the additional overhead must be paid even in static areas of the system. (In this context, 'overhead' refers both to the code footprint and execution time of run-time libraries, but also to the predictability of the software and how easily it can be analysed for correctness.)

In contrast, in a fully-static system, programs use a fixed number of threads and shared objects that do not migrate between processors. Consequentially, because the hardware architecture is also modelled, communication resources can be allocated offline. When used with analysable subset languages (like Ravenscar Ada / Java) the resulting implementation is easier to analyse and verify for correctness in safety-critical contexts. It will also require less run-time support, leading to lower memory overheads and faster execution times.

Use of the clustering system model still allows for useful dynamism in the final implementation, but this dynamism is explicitly enumerated and modelled. The programmer can create fully-dynamic systems (one program-wide cluster mapped onto one architecture-wide CT), fully-static systems (one cluster per thread / coordination primitive / data item, one CT per processor-memory pair, one-to-one mappings) and anything in between. The implementation is given much more information about the intended behaviour of the program, and can tailor it accordingly. Static parts of the system require only minimal support libraries, and are not affected by parts of the system that implement dynamic behaviour.

This represents a conceptual change from the way that systems are normally designed. With the assumption of no dynamic behaviour the 'default' run-time support required is minimal. Any extra dynamic behaviour that is enumerated by the programmer can be explicitly included by the run-time system, only for the areas of the system that need it.

3.6.2 VP requirements

The purpose of the VP is to allow the architecturally-neutral software of the source application to execute correctly on highly non-standard target architectures. The requirements of the VP therefore can be determined by enumerating the ways in which the architecturally-neutral code can fail and remedying them. In all cases, code will fail to execute correctly when the underlying architecture does not meet the implicit assumptions made by the language's programming model. The targeted languages (C, Ada, Java) therefore require the following features of the VP:

- Provision of a single logical address space over a NUMA
- Universal communications between all system elements
- Cache coherency

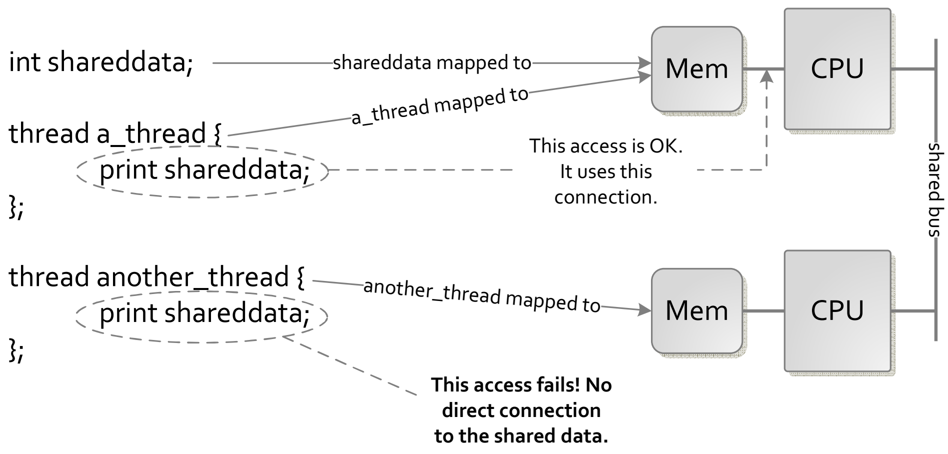

Figure 3.13: Problems caused by the assumption of a single logical memory space in non-uniform architectures.

- Thread and data migration

These requirements are linked to the system model and examined in detail in the following sections.

Single logical address space

C, Ada and Java, like the majority of common imperative languages, assume the presence of a single logical memory space in which the data of their program is stored. This is a realistic assumption for general-purpose architectures, but not so for embedded architectures in which non-uniform memory architectures (NUMAs) are common. The language model assumes that any data item in scope can be accessed from any part of the code. In an architecture in which all processors have access to the same memory (such as SMP or cc-NUMA) this assumption holds. However, consider the system shown in figure 3.13. Two threads are allocated to different processors, and each processor has its own local memory space. Data allocated to the local memory space of one processor cannot be accessed by the other, and the code will fail. If the data is duplicated so that it appears in both memory spaces, then updates to the data from one processor will not affect the data in the other and consistency will be lost.

In terms of the system model, for each processor p of the target layer, the VP is required to ensure that every shared data item (from the program layer) that is accessed by the concurrent objects that may execute on p appears to be located in a memory space attached to p. The set of concurrent objects (COs in the formula) that may execute on p is as follows:

$$execute(p) = \sum_{c \in COs} \exists l \in clusters, t \in targets \cdot c \in l \wedge l \in t \wedge p \in t$$

Distributed shared memory (DSM) systems [211] have been proposed as a solution to this problem. Implemented either as part of the operating system [114], as middleware, or as an application library, a shared memory system allows the consistent sharing of data between multiple computation nodes by monitoring the use of shared memory segments and distributing changes to the memory appropriately. There is a wide range of algorithms and shared memory paradigms in use that are appropriate for different usage models, programming models, and target architectures [174].

DSM algorithms can be sorted into four main categories:

- Centralised: The data is physically stored in one place only. Accessors must read from and write to this location serially through a central 'data server'.
- Non-migrating replicated: The data is replicated between all accessors, changes to the data are propagated between accessors.
- Migrating: The data is stored by its accessors, but only one copy is maintained at any one time. When another accessor wishes to access the data it is migrated to the new accessor and the old accessor deletes its copy.
- Replicated migrating: The data is stored by all accessors, but access semantics operate like the migration algorithm. An access token is passed between accessors, along with local changes to the data.

Furthermore, DSM systems can be subdivided depending on their granularity:

- Page-based: The distributed units are pages (address ranges) of memory. These systems are the most flexible as they require no specific software support and can be implemented transparently. They suffer from fragmentation issues and cannot optimise as aggressively due to a lack of source code knowledge. Two examples of this kind of system are Teamster [40] and Strings [192].
- Object-based: Distributes language-level objects (integers, arrays, structures etc.). Requires language support, but allows for more efficient implementation than a page-based system. Existing examples of such systems are Adsmith [144] or Rthreads [65].

The VP is therefore required to implement a shared memory system which can allow all threads of the system to access the data items that they require, transparently. Due to the level of source code information that CTV has, an object-based solution is a better choice than a page-based solution. The main advantage of page-based systems (transparency) is already provided by CTV's virtualisation layer, the VP.

Universal communications

Related to the assumption of a single shared memory space, languages assume the availability of universal communications between all computing nodes of the architecture. The implicit architecture is that of either a totally-connected grid of processors and memory spaces, or a single shared bus. The result of this assumption is that, in terms of the system model, the architecturally-neutral code of the concurrent objects assumes that it can communicate with

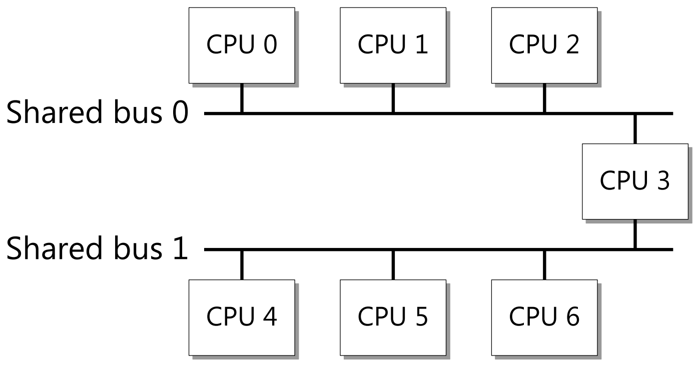

Figure 3.14: Example architecture that requires multistage permutation routing.

all the other concurrent objects and shared objects in the system. The system model does not explicitly model communications, they are implicit based on the shared objects and shared data items that each concurrent object accesses.

Figure 3.14 illustrates an architecture in which this assumption is invalid. In this architecture there are two shared buses along which processors can communicate. Only a single processor, CPU3, is connected to both buses. Therefore to send a message from CPU 0 (situated on one bus) to CPU 4 (situated on the other bus) it is necessary to transmit in two hops, using CPU 3 as a relay. On larger architectures this generalises to communications that may involve multiple hops to reach their destination and may also have alternate routes. In the literature this routing problem is referred to as *multistage permutation routing* [216].

This problem may be solved using either an online or an offline solution. Online solutions use routing protocols commonly based on the OSI seven layer model [117] that is used in the internet and other large networks. Nodes store routing information about possible destinations and construct spanning trees at run-time so that they know where to send packets. This results in a flexible system, but overheads and implementation costs are high due to the complexity and storage requirements of the routing hardware. Also, such algorithms are best suited for dynamic networks that may change their topology, but the majority of an embedded architecture is static and only small areas may require adaptability.

Accordingly, many researchers have considered offline routing protocols that use information about the structure of the network to inform routers at design time how to route their packets. Systems such as SoCBUS [240] use this principle. In SoCBUS, each node knows whether any other node is generally North, South, East or West so it knows which interface to relay the message onto. Also, many real-time NoC systems fully route their messages statically at design time to allow for offline analysis of traffic patterns, latency, and bandwidth. Early work [18, 156] showed that with sufficient information about the expected traffic beforehand it is possible to generate scalable and efficient routes for multistage permutation routing. This work has led to the development of wormhole routing [170]. These problems are still being considered for more complex network topologies [47].

The VP is required to implement a routing system that allows universal communication between all nodes of the system. The static nature of the CTV system model can be exploited to allow the implementation an offline system with lower overheads than an online system. A small component of the system must still operate online, to account for the dynamism of the logical layer.

Coherent caches

When shared memory is accessed through caches by multiple processors it becomes necessary to keep those caches coherent. In terms of the system model, this situation arises when all the following are true:

- In the target layer, there exists a memory space m that is accessible from two processors, p_1 and p_2.
- There exists two concurrent objects, c_1 and c_2 that may at some time execute on p_1 and p_2 respectively.

  $$\exists c_1, c_2 \in concurrentobjects \cdot c_1 \in execute(p_1) \wedge c_2 \in execute(p_2)$$

  $execute$ is the set of all concurrent objects assigned to a processor and was defined above.
- There exists a shared data item d that is accessed by both c_1 and c_2.
- p_1 and p_2 have data caches in their implementation.

If all of these are true then cache coherency must be considered. Coherency schemes vary in their implementation medium (whether they are hardware, software, or hybrid), the mechanisms by which they detect coherency issues, and the algorithm they use to resolve them.

A common subdivision is to split cache coherency solutions into hardware-based and software-based schemes [208]. Hardware-based systems are implemented entirely as dedicated hardware units, usually connected to the memory bus of the system. This is the kind of coherency solution found inside coherent multicore processors, like the ARM Cortex [83]. Such solutions have the advantage that their use is entirely transparent to the programmer, but they can require a large amount of power-consuming coherency hardware and their lack of software knowledge leads to overly pessimistic coherency assumptions. They have no way of knowing whether shared data will be needed in the future, so they are forced to keep data that was once shared coherent long after it has been deallocated by the program. Similarly, without software knowledge they do not know which processors will access the shared data in the future so they are forced to pessimistically keep all cores coherent. Two examples of hardware-based schemes are snooping protocols [127] and snarfing protocols [7]. Snooping involves the monitoring of memory bus traffic by a dedicated hardware unit of the cache controller to detect potential coherence problems. When detected, the affected cache lines are invalidated. Snarfing is similar to snooping except when a remote processor modifies data that the snarfing cache controller has cached, the controller updates its local copy.

Software-based approaches use software libraries to explicitly create shared memory regions that are to be kept coherent. Such an approach requires the programmer to manually maintain the cache, but can provide a lower-overhead solution that is more scalable than a purely hardware-based scheme. They reduce many of the problems of hardware-based schemes because the extra software information can be used to reduce overly pessimistic coherency assumptions. The main drawback of a software-based scheme is that they require processor time to execute, thereby potentially reducing the throughput of the system.

An example of a software-based mechanism is directory-based coherence [1, 36]. Directory-based coherency stores a table (the directory) of all items for which coherency is to be maintained. Processors explicitly request and update entries in the directory. This is more scalable than snooping-based mechanisms which determine this information by inspecting transactions on the memory bus (at either the software or hardware level). Unfortunately, programming language assumptions of universal coherency still require this directory to be available to all cores of the system. As identified previously in section 2.6.3, recent work [149] has showed limiting coherency into coherency regions is more scalable, but requires explicit software support.

Inevitably as embedded architectures become more complex, hybrid schemes will become required as parts of the target architecture provide hardware coherency whilst other sections do not. The VP is required to implement a cache coherency system that allows the processors of the system to use shared data over complex architectures without compromising the coherency of that data. The extra amount of software support afforded by CTV allows the VP to implement a low-latency scheme that is appropriate for varying target architectures.

Migratable threads and data

The final requirement of the VP is that to support the logical layer of the system model it must be able to implement thread and data migration throughout the processors and memory spaces respectively of the target architecture. Recall that the model defines two forms of migration:

- A cluster may migrate to any of the cluster targets to which it is mapped.
- A concurrent object c may migrate between the processors of the cluster target to which c's cluster is currently mapped. Equally, a shared data item d may migrate between the memory spaces of the cluster target to which d's cluster is currently mapped.

Thread migration is the task of moving an executing thread from one target processor to another. It involves moving the thread's code, state and its local data (stack). On shared memory architectures this can sometimes be implemented simply by passing a reference to the data. Without shared memory the data has to be explicitly moved. Migration may occur inside a cluster and between clusters depending on the implementation used. This is discussed more in section 4.13.

After migration the system must resume in a consistent and coherent state. In a general-purpose computing environment this can be very challenging, as the presence of arbitrary pointers (such as in C) vastly complicates the migration and memory translation process. In languages that do not allow raw pointers (Ada and Java), the task is simpler as only the reference has be to translated.

Thread migration techniques can be coarse-grained or fine-grained. Course-grained systems such as UPVM [136, 137] provide the migration of *user level processes*, which are coarsely-grained computation units that do not share memory and only communicate using message passing. Therefore, the migration mechanism only needs to transfer the state of the user level process, shared memory does not need to be updated. The downside of such systems is the requirement for message-passing as the top-level communications mechanism, and that the state of the processes can be very large.

Fine-grained systems such as MILLIPEDE [82, 119] and Ariadne [161] allow the migration of lightweight threads that exist inside a shared memory environment. Such systems generally require the transfer of smaller amounts of state (threads only use a small amount of local stack space compared to full processes) but the presence of shared memory complicates matters as any references to migrated data must be updated.

Given CTV's target domain of heterogeneous embedded systems, there are a number of additional requirements that must be taken into account. In a homogeneous architecture the source and target processor might have different ISAs. The binary for one processor of the system might not work for another. To solve this problem, the potential migrations must be determined in advance, and the code recompiled for each potential target processor. Related to this, if the processors have different architectures then it is not possible to transfer the state of one processor to another directly. Either limitations must be placed on the migration targets (for example to only migrate between broadly compatible processors) or they must only take place at predefined 'checkpoints' in the code where migration is safe. Finally, if the processors use different representations (big-endian vs little-endian), data-marshalling for local data must be performed.

Clearly in an actual implementation, unrestricted thread migration from any processor to any other processor is impossible in the general case. However, the level of source information that is afforded to the designer by CTV's system model allows for appropriate restrictions on migration to be codified and checked, so that it can be supported in the areas of the architecture that are amenable.

3.7 The Object Manager model

The previous sections detailed the features that must be provided by the VP. In order to implement these, the VP defines a unique communications framework called the Object Manager (OM) model that makes use of the compile-time nature of CTV and the extra information provided by its system model to provide a scalable implementation.

It is not possible to use a standard OS kernel to provide the required communication, shared memory and thread migration services. Doing so would introduce a bottleneck in the system, as all threads would have to communicate with the OS kernel to use the provided services. The OS is typically located on a single core (or a single group of cores) and so as systems grow larger this limits scalability.

In response to this problem, *distributed operating systems* (DOS) have been developed, of which there are a number of recent examples [238, 17, 173]. In a DOS, the OS is composed of two layers, the *microkernel* layer and the service layer. The microkernel provides the minimal functionality required for the operation of the rest of the node. Commonly this includes features such as threading, scheduling, interrupt handling, timing, callback routines, and heap management. These features are localised to within a single processing core and do not require communication with other cores.

Running on top of the microkernel is the service layer which provides unified OS services to the rest of the system. In a DOS, provision of these services is distributed across the entire multicore system. This distribution requires coordination to maintain elements of global OS state between all the OS-carrying nodes of the system. For example, when a thread locks a mutex this information must be distributed to all other nodes of the system so that further attempts to lock the same mutex are blocked. The problem with this approach is that maintaining this distributed state can require a large amount of communication. Existing DOS approaches have been designed with the goal of providing a transparent, general-purpose solution. The lack of compile-time information about the intended application forces the DOS to pessimistically maintain state across the entire system in case it is later required.

The OM model leverages CTV to take a different approach. Applied at compile-time, the OM model has access to the full source code of the application, along with the program, target and logical layers of the CTV system model. It makes use of this unique clustering and mapping information to limit propagation of state and the provision of OS services to the subset of the system that actually requires them. Rather than attempting to provide all services at multiple points throughout the system and then coordinate to achieve a consistent state, the OM model only provides each service at a single node, but that node can be mapped physically close to the threads that require it. Services are also provided at a fine level of granularity. For example, rather than all mutexes being coordinated by a dedicated 'mutex manager', individual mutexes are provided as separate services and can exist on different cores of the architecture, close to the threads that require it. Other threads in the system that do not access that mutex do not affect the scalability of the final implementation. The result is an application-specific system for the provision of OS services, that is capable of much lower run-time overheads than a general-purpose system (as highlighted in figure 3.15).

It can be seen therefore that the OM model replaces the service level of the traditional distributed operating system model, it does not replace the microkernel. Existing microkernels can be used by the OM model without modification so there is no need to redefine them.

The OM model has three main advantages. First, the scalability of the model is limited only by how interconnected the source application is. If all mutexes, shared data items etc. are accessed by all threads of the system then the OM model performs no better than a standard OS. However, for better-constructed programs in which shared items are only required by a subset of the system threads, communication requirements are accordingly reduced. Second, the model is decentralised and can support large architectures. Hardware-permitting, simultaneous service requests can execute in true parallel, due to the fact that OMs do not need to propagate and maintain a single OS state. This is illustrated in figure 3.16. Finally, the implementation of the OMs can be more efficiently mapped to the target architecture than a general-purpose system that is required to support a range of architectures. As detailed in the following section, each OM is mapped to elements of the target layer from the CTV system model. This allows the services provided by the OM to be located physically close to the threads that require them.

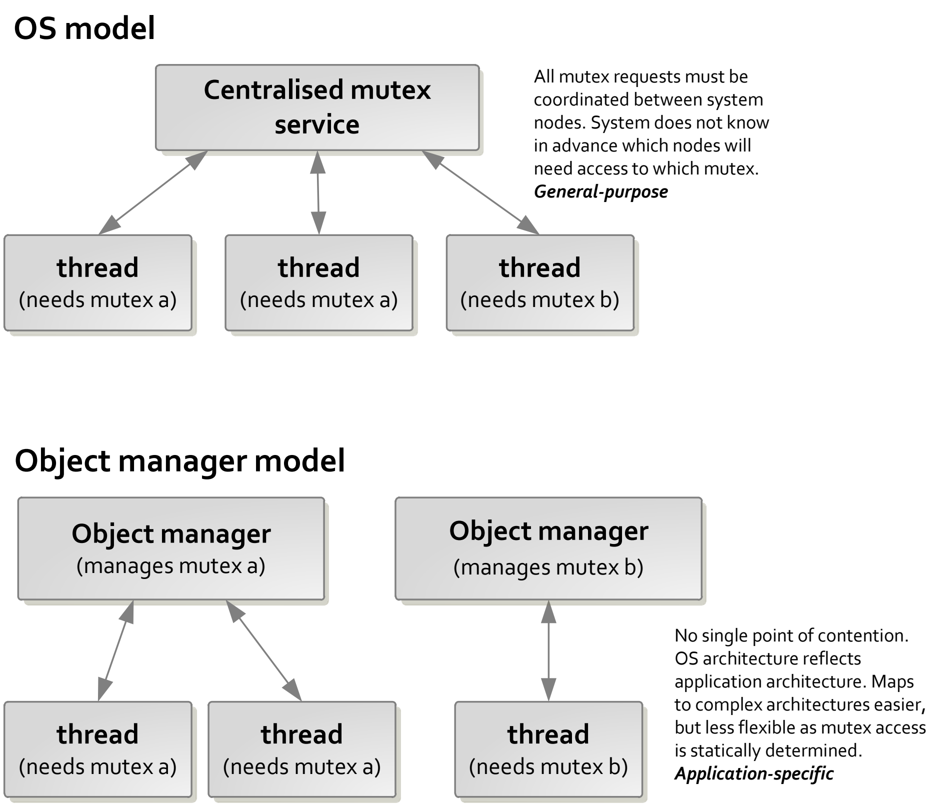
OS model
Centralised mutex service
thread (needs mutex a)
thread (needs mutex a)
thread (needs mutex b)
All mutex requests must be coordinated between system nodes. System does not know in advance which nodes will need access to which mutex.
General-purpose
Object manager model
Object manager (manages mutex a)
Object manager (manages mutex b)
thread (needs mutex a)
thread (needs mutex a)
thread (needs mutex b)
No single point of contention. OS architecture reflects application architecture. Maps to complex architectures easier, but less flexible as mutex access is statically determined.
Application-specific

Figure 3.15: The difference in approach between existing OS systems in which services are provided at a uniform location and the OM model.

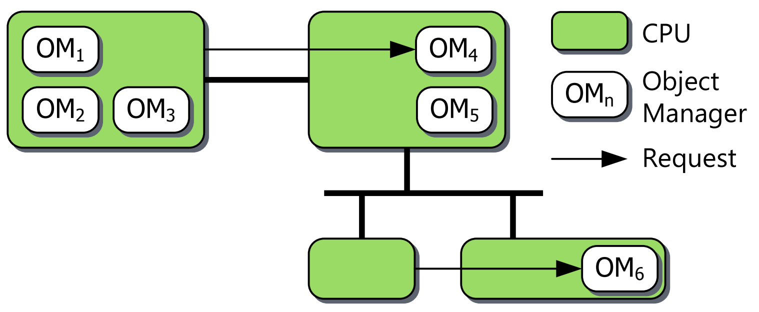

Figure 3.16: The OM model is totaly decentralised. Hardware-permitting, simultaneous service requests execute in parallel.

3.7.1 OM specification

Overview

The OM model defines at compile-time a static set of object managers (OMs) that are distributed across the target architecture. An OM has the following features:

- Manages shared objects and shared data. Each shared data item and shared object (including implicit 'thread helper' shared objects, see section 3.4.5) from the program layer must have exactly one OM. An OM can manage any number of items. 'Managing an item' means providing the OS services associated with that item, defined in detail later in this section.
- Passive. OMs are passive objects. They only perform actions in response to a communication from a concurrent object elsewhere in the system. This allows them to be implemented in interrupt handlers.
- Single thread of control. OMs contain a single thread of control (because they execute at a level of abstraction lower than the kernel). Therefore if an OM manages multiple objects, the requests are serialised and not reentrant. Accordingly, commonly-used services should be assigned to their own OMs on their own processing element. Less frequently-used services may share OMs and processing elements.
- Mapped to the target architecture. OMs can be implemented in either software or hardware, as described later in this section. Software OMs are mapped to a set of target processing elements from the target layer. At run-time, the OM may migrate between the set of processors to which it is mapped in a way exactly the same as concurrent objects mapped to a cluster target in the logical layer. Hardware OMs are mapped as custom hardware elements of the target layer. Migration of OMs might be omitted by an implementation if desired. In this case, the system will allow concurrent objects, shared items and shared data to migrate, but not OMs, and if a processor which is hosting an OM shuts down the system may fail to operate correctly.
- Static. As the set of OMs in the system is compile-time static, OMs cannot be created or deleted at run-time.
- No heap. Unlike shared objects, OMs may contain only static state variables. These cannot be (and need not be) accessed by other objects in the system, including by other OMs. A software implementation should endeavour to limit the size of this state so as to affect software on the host processor as little as possible. The OM will still make use of the stack for implementing function calls etc.
- No remote access. OMs cannot directly access shared data or shared objects during their execution. To do so would involve the OM calling services on other OMs which can lead to deadlock, as well as resulting in unpredictable execution behaviour. (On complex architectures OM's will still coordinate to provide distributed communications, but this happens at a level lower than that of the input program.)
- Very low abstraction level. OMs must be able to execute without using any OS features at all (for times when they are mapped to a processing node without a microkernel). This means that dynamic memory allocation and similar features must be avoided.

- Generated at compile-time. Because of CTV's static system model, the behaviour of each OM can be largely determined at compile-time. The CTV system model specifies the objects in the input program, the layout of the hardware architecture, and the mappings between the input program and the hardware architecture. From this, and information on where in the target architecture each OM is to be located, each OM can statically determine the communications routines required to send messages to other threads, perform offline routing for more complex architectures, and include only the driver code which is required by the chosen architecture. If the architecture or input program changes, the OM layer must be regenerated and recompiled (or resynthesised in the case of hardware OMs).

Services provided by an OM

Recall that the purpose of the OM communications model is to implement the requirements of the VP as detailed in section 3.6.2. The set of services that each OM is required to provide is determined by the objects that it manages. OMs can manage shared objects (including implicit 'thread helper' shared objects detailed in section 3.4.5) and shared data items. OMs must manage the union of the services that are required by its managed items. Recall that OMs do not provide all of the services that are traditionally provided by OS kernels and are frequently (but not necessarily) combined with a microkernel or full kernel to provide the user application with access to services like threading, scheduling and dynamic memory allocation. This section details the services that OMs are required to provide.

- **Services provided by all OMs**
  - Offline, multistage permutation routing for all communications. The OM can determine offline which other OMs and processors the OM will potentially need to communicate with. The routing for these communications is performed offline.
  - Message forwarding. Related to the provision of offline message routing, many architectures will require OMs to be able to act as routers, passing messages from one interface to another across the system.
  - Migration support, detailed later in this section.
- **OMs that manage shared objects:** From the definition of shared objects given in section 3.4.2, the following services need to be provided:
  - Remote procedure / function calls for the interface of the shared object.
  - Internal locks (the read lock and write lock) that may be automatically acquired by the interface subroutines.
  - Interface guards that are correctly observed and reevaluated.
  - Migration support, detailed later in this section.
- **OMs that manage shared data:**
  - Object-based shared memory system with migration semantics. Allows the processors which access this shared item to migrate the data (or a portion of it) into their local memory space, access it, and then migrate the changes back to the remote

copy. Correct concurrent operation does not have to be guaranteed, it can be assumed that the input program will correctly obtain and use mutual exclusion in all situations in which shared memory is accessed by multiple clients.

- Cache coherency across all processors that access the shared data and have caches. The specific algorithm is not specified, but as all requests for shared data pass through the OM the system can efficiently implement a directory-based algorithm. Cache coherency hardware should be used by the OM if available.
- Migration support, detailed later in this section.

• **OMs that manage 'thread helper' shared objects**

  - Thread helper objects are implicitly-defined shared objects and so are supported in exactly the same way as shared objects. The interface exposed is determined by the language mapping, as discussed in section 3.4.5.
  - Migration support, detailed later in this section.

Implementation styles

An OM exists at a physical point in the hardware architecture, but the actual implementation of an OM is undefined by the OM model. There are two main implementation styles that can be adopted by the OMs - software or hardware.

Software-based solutions implement an OM as an interrupt handler in the processing elements of the system. The handler is triggered by the receipt of communications from concurrent objects in the system requesting access to the OM's provided services. This allows for maximum flexibility, as OMs can be created and removed easily (at compile-time). However this causes interference to the execution of the processor's threads. Software OM's allow local requests (requests in which the concurrent object and the OM are hosted on the same node) to execute very quickly.

Hardware-based solutions use custom, dedicated hardware units that are custom-built to implement the required services and to be accessible from the processing elements of the system. This is similar to existing hardware-provided OS services [3], but application-specific rather than general-purpose and of a much lower granularity (individual objects if required). Hardware implementations allow higher response times than software and does not affect software execution, but requires new hardware to be generated for each new OM mapping. If the designer allocates an OM to manage an additional service, or moves it to another location, the hardware must be regenerated. Mixed approaches are also possible in which high-traffic OMs (like the manager of a heavily-used mutex) are implemented in dedicated hardware and other OMs are implemented in software.

Accounting for migration

Dynamism defined by the logical layer allows concurrent objects, shared objects, and shared data items to migrate at run-time over a subset of the target architecture. The migration of an object is provided as a service by its OM. Migration consists of two phases - the actual

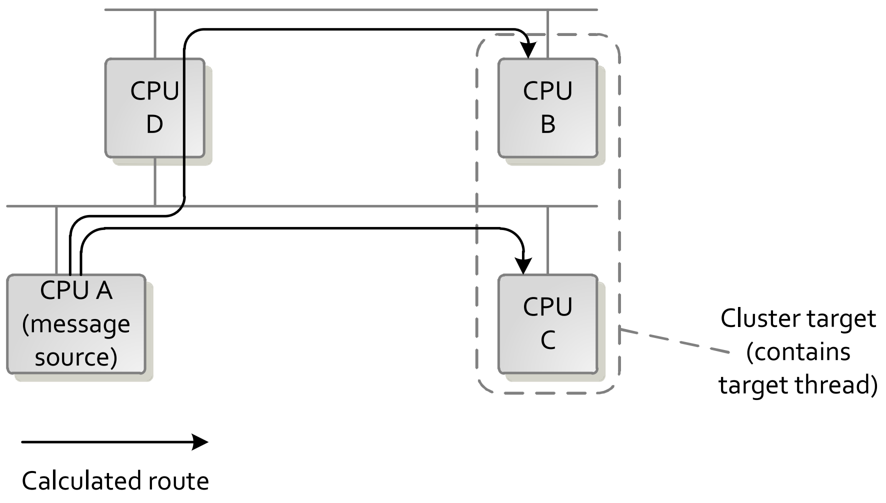

Figure 3.17: Offline generation of routes to objects that may migrate.

migration, and then the updating of communication links with the new location of the migrated object.

To perform the actual migration, the migration service must be able to pause the execution of the object, move it and its internal state to the target location, and restart its execution. Assuming the absence of race conditions, the system execution should proceed identically (from a functional perspective) as it would have done without the migration. In an homogeneous architecture this is possible, but arbitrary migration across heterogeneous architectures can be hard to implement, so an implementation will place strict limitations on the type and scope of migrations possible. Also, OMs may be mapped to more than one physical processor from the target layer, meaning that the OM may choose migrate itself as well as the object it manages. Support for this is optional and is specified by the implementation.

Once the migration has been completed, other parts of the system should be able to continue executing without being disrupted. The OMs form a statically-routed communication network across the target architecture, so after a migration these links must be updated. This is implementation-neutral and so is defined by the OM model. The situations that must be considered are as follows:

- **Concurrent object migration:** Recall that the system model does not provide direct inter-concurrent object communications. All communications use an intermediary shared object (including implicit 'thread helper' shared objects). Therefore the only feature of the system that is affected when a concurrent object moves is its manager, which still needs to be able to interact with the thread to perform operations such as interrupting or terminating it. Therefore, the concurrent object's manager must, at compile-time, prepare routes to all possible cores that the concurrent object may migrate too. When targeting migration across a broadcast network the routes will be identical, but non-standard architectures might result in more complex scenarios, as depicted in figure 3.17. [3]

[3]Note that as shown in section 3.4.5, for the presented languages C, Ada and Java the interactions between a

- **Shared object migration:** Migration of shared objects is more complex than migration of concurrent objects because other objects of the system can communicate directly with the migrated item. There are two suitable algorithms that can be used to solve this problem described below. These algorithm have different costs and benefits, so the most suitable choice depends on the behaviour of the system being implemented.

  - **Location propagation:** In this algorithm, the OM of the migrating item is responsible for storing a list of all objects that can communicate with it. This list must be obtained at compile-time from static analysis. It is not enough to build a list at run-time because the object may migrate before the first communication from a remote object. When the object is migrated by its OM, the OM must send a message to all objects in its communication list to inform them of its new position.

    The advantages of this algorithm are that it can support arbitrarily complex architectures, and it can operate in a system in which all OMs migrate and there is no provision for global broadcast. Also it does not impose any overhead on normal communications because each object maintains a consistent system view. The disadvantages are that the communications list can become rather large in some applications and impose a large memory footprint on the OM of the migrating item. Also, after a migration many communications are required to propagate the new location to all accessors.

  - **Intermediary manager** The intermediate manager algorithm uses a second OM assigned to a static location, called an intermediate manager (IM) to store the current location of the migrating object. The IM must not migrate throughout the execution of the application so that accessors can statically route their communications to it. Objects that wish to communicate with the migrating object first communicate with the IM, which then forwards the communication to the appropriate location. During a migration, the migrating object's manager informs the IM of the new location. If the OM of the migrating object does not itself migrate, then the IM and the OM can be combined into a single location.

    This algorithm introduces a small communications overhead as objects must have their communications routed by the IM to the actual object. However, the communication requirements after a migration are reduced and decoupled from the layout of the application making the algorithm more scalable. Also the potentially large communication lists do not need to be obtained and stored.

  These algorithms are only required when migrating objects across an irregular architecture. If migrating throughout a broadcast network (such as CAN or Ethernet) then such techniques are unnecessary because the method of access (broadcasting a packet and waiting for the reply) is the same. This can be determined at compile-time so only the required algorithms are implemented.

- **Shared data migration:** Shared data migration is similar to the migration of a shared object. The actual migration is simpler as it only requires the copying of the shared data from one memory space to another. Once this is complete, the same algorithms as above are used to ensure that the rest of the system can still access the shared data item.

concurrent object and its manager are infrequent and require only low bandwidth. Therefore an implementation will likely choose not to migrate the object's manager when possible.

3.8 Conclusion

This chapter has further explored the problems that are encountered when developing software for complex, non-standard architectures. The programming models of existing languages assume a fixed, standard architecture of homogeneous processors and globally-contiguous shared memory that is very different to the majority of embedded systems. As a result, software cannot be adequately mapped over these systems and the programmer is forced to break the abstraction models of their source language with low-level, hardware-dependent code. This is error-prone, time-consuming, and limits code reuse.

Furthermore, existing development models fail to allow the programmer to express the *coupling* of their application. Coupling is defined as a relative partial ordering between the threads and data items in a system. Tightly-coupled items communicate and share data frequently, whereas loosely-coupled items tend to only exchange infrequent, high-level messages. Existing development toolchains do not allow the programmer to provide this information, however it is of great importance when targeting embedded architectures due to their heterogeneity. Coupling information allows the toolchain to place tightly-coupled items closer together and route their communications over higher-bandwidth links. It can also be used to influence task migration policies, thereby supporting dynamic embedded systems.

This chapter has defined the clustering model, which is a hierarchical model for application development that allows the expression of coupling. This is developed into a full system model that models features of the source application, features of the target architecture, and the way that the source items are distributed over the hardware. The system model includes a logical layer that presents an expressive model for task and data migration.

The developed system model requires considerable support from the programming language that is typically not provided. For example, it requires the assumptions of global shared memory and universal cache coherency to be relaxed so that more complex architectures can be supported in which such assumptions would be inefficient and wasteful. To solve this problem, a virtualisation-based technique is developed called Compile-Time Virtualisation (CTV). CTV is a language-agnostic technique that presents the programmer with a Virtual Platform (VP) that is an idealised view of the underlying hardware. The VP supports the assumptions of their chosen language. For example, it presents global shared memory but implements this on a non-uniform memory architecture. Unlike existing virtualisation-based techniques, the VP does not introduce run-time overhead because it is applied solely at compile-time. This makes it suitable for use in resource-constrained embedded systems.

In order to support future architectures, CTV uses a novel communications layer called the Object Manager (OM) model. The OM model builds on the clustering and architecture mapping information provided by CTV to present a scalable solution to on-chip communications. The OM model provides services such as communication, coordination, shared memory and task migration in a distributed manner without the need for a full operating system running on each processor of the system. The OM is entirely decentralised, removing a potential source of bottlenecks and making it as scalable as the architecture of the source application.

Chapter 4

Anvil: An Implementation of CTV

The previous chapter introduced CTV, a virtualisation-based technique to assist the development of software for complex embedded systems by exposing extra architectural details to the programmer. CTV was defined to support a clustering-based programming model that allows the programmer to better target partially-dynamic non-uniform architectures. CTV's system model was defined, along with its novel decentralised communications layer which is termed the Object Manager (OM) model. The OM model is used to provide OS services in a scalable way across complex architectures without the need for a traditional distributed OS and its associated overheads. The OM model relies heavily on the extra programmer support afforded by the CTV system model and its compile-time nature. In this chapter, an implementation of CTV is presented along with in-depth implementation details that show how CTV can be utilised in practice.

Anvil, shown in figure 4.1, is an implementation of CTV that takes applications written in ANSI C as its input. The programmer makes use of the POSIX pthreads library [115] to describe multiple threads of control and their interactions. C was chosen because it is an industry-standard language that has been used extensively for the development of both soft real-time and hard real-time embedded systems. Tool support for C is mature and extensive, and as shown in section 3.4.6 it is easily represented by the CTV system model (once it has been combined with pthreads to express parallelism and coordination). Ada is also extensively used

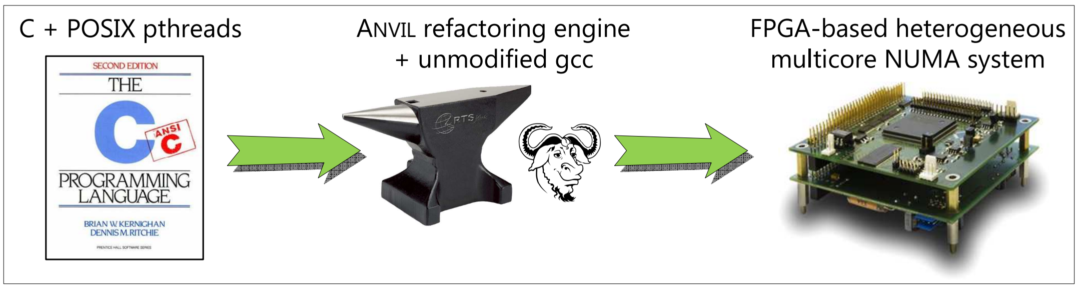

Figure 4.1: Overview of Anvil, an implementation of CTV.

for embedded development, but C was selected because it is a simpler language that is easier to parse and analyse.

Anvil maps its input programs over FPGA-based, complex, multi-core architectures with non-uniform memory, custom on-chip interconnect, and non-standard hardware features such as function accelerators. Due to the fact that the input code is standard ANSI C, Anvil uses an unmodified version of the `gcc` compiler internally for object code generation. Input code is extensively refactored so that `gcc` is presented with source code that will operate correctly on the target architecture.

As CTV is a compile-time technique, restrictions have to be placed on the run-time behaviour of the input code that Anvil can process. These restrictions are collated and summarised at the end of this chapter in section 4.15 and are motivated and discussed throughout in their relevant sections.

The chapter first describes an overview of the Anvil implementation scheme (section 4.1). It then defines *AnvilADL*, a description language that is used by the programmer to describe the target architecture and the behaviour and capabilities of Anvil's VP. Then, the compile-time refactoring process that Anvil uses to generate its run-time libraries and distribute the programmer's code over the target architecture is detailed in section 4.3. Section 4.12 describes how Anvil allows the definition of shared objects to support the use of dynamic data structures across non-uniform memory architectures. The implementation of migration is discussed in section 4.13, and the impact that Anvil's refactoring has on traceability is considered in section 4.14.

4.1 Implementation overview

As a compile-time system, Anvil is centred around a source-to-source refactoring engine which takes the programmer's architecturally-neutral input C code and refactors it for execution on the target architecture. Anvil is comprised of the following parts (shown in figure 4.2):

- AnvilADL. A simple architecture description language that is used by the programmer to describe their code in terms of the CTV system model (section 3.4). AnvilADL allows the definition of all the elements of the source, logical and target layers, and to describe the mappings between them. As it is designed to support a CTV system, the definition and placement of OMs is also performed here. This layer is presented along with the input code to the rest of the Anvil toolchain. AnvilADL is described in section 4.2.
- Refactoring engine. Anvil's refactoring engine is a full C parser and static analysis tool that understands the semantics of the pthreads library. It is used to analyse the input source code and apply source-to-source transformations to turn the input single-program into code suitable for execution on the target architecture. The result of this stage is a set of programs (one for each target processor) which implement the same functionality as the input program. The refactoring is guided by the programmer though the mappings in the AnvilADL description of the system. The refactoring engine is described in section 4.3.
- Object manager libraries. As detailed in section 3.7.1, to support the CTV system model

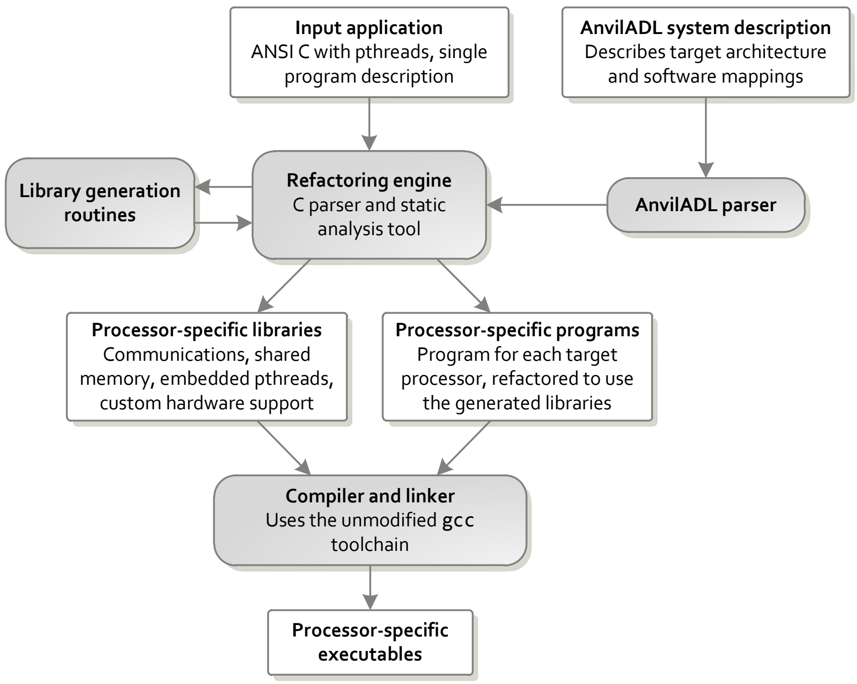
Input application
ANSI C with pthreads, single program description
AnvilADL system description
Describes target architecture and software mappings
Library generation routines
Refactoring engine
C parser and static analysis tool
AnvilADL parser
Processor-specific libraries
Communications, shared memory, embedded pthreads, custom hardware support
Processor-specific programs
Program for each target processor, refactored to use the generated libraries
Compiler and linker
Uses the unmodified gcc toolchain
Processor-specific executables

Figure 4.2: Overview of the main processes in the Anvil system

the Object Managers are required to implement a range of OS services, including cache coherency and shared memory systems. The code to do this is provided in a range of libraries that are included as necessary at compile-time by the refactoring engine. These libraries are detailed in section 4.6.

- Custom hardware support. Anvil includes a library of driver functions that are used for accessing communication channels and other elements of custom hardware according to the CTV hardware model (section 3.4.4). These are integrated into the OM code at compile-time as required.

The rest of this chapter describes the above technologies and details their implementation and use.

4.2 AnvilADL - Virtual Platform description

AnvilADL is a simplified architectural description language that is used by the programmer to represent their system. Primarily it describes the input application, target architecture, and the mappings between them in terms of the system model (section 3.4). The following features are described:

- Features of the source application (from the program layer, section 3.4.2) – concurrent objects, shared objects and shared data.

- Features of the target architecture (from the target layer, section 3.4.4) – processors, channels, memory spaces, custom hardware.
- Logical elements (from the logical layer, section 3.4.3) – clusters, cluster targets.
- Mapping information (source layer objects → clusters, clusters → cluster targets, cluster targets → target layer objects).

Also described are lower-level implementation details:

- Detail of target layer objects to provide more information to the toolchain. For example a processor's ISA, or whether a given memory space is cached.
- The connectivity and topology of the target architecture (the way in which memory spaces and channels are connected to processors).
- The OMs of the system (section 3.7), and which objects are assigned to which OMs.

The ADL is responsible for defining the VP that is used in an Anvil system. The programmer uses AnvilADL to express the way in which their code should be mapped to a given architecture. This is then used to influence the operation of the rest of the Anvil system. The same input code can be mapped differently, or to a different architecture, by providing different AnvilADL. The following sections describe the AnvilADL language.

4.2.1 AnvilADL - syntax

AnvilADL does not have the expressive power of a full ADL [49] because it only needs to provide a high-level view of the software and hardware architectures in terms of the layers of the CTV system model (as defined in section 3.4.1). AnvilADL's syntax is shown in figure 4.3. A section of AnvilADL code is called a *description*, in the way that a section of C code is called a *program*. This is because AnvilADL is not executable or compilable, it is used to describe and model a system.

An AnvilADL description consists of two types of statements, *declarations* and *assignments*. Declarations are used to introduce new *features* into the description. Each feature is categorised into one of the following *feature categories*:

- `processor`, `memory`, `channel` or `hardware` are used to define architecture features that are modelled by the CTV system model's target layer (section 3.4.4).
- `cluster` and `target` are used to define the objects of the logical layer (section 3.4.3).
- `manager` is used to define OMs (section 3.7).

Within the architectural categories, features can be given a type to provide further information about their implementation. For example, a declared feature can be of category `processor`, and of type `PowerPC`. This type is a freeform string, which is used to inform Anvil's refactoring engine which library functions should be used to interact with this hardware element. Giving

```
(* AnvilADL syntax definition *)

(* Top-level non-terminal *)
description ::= {declaration | assignment}

(* Declarations *)
declaration ::= identifier_list ":" feature_category [iptype];
identifier_list ::= identifier | identifier "," identifier_list
feature_category ::= "processor" | "memory" | "hardware" | "channel" |
    "cluster" | "target" | "manager"

(* Assignments *)
assignment ::= lvalue "=" rvalue ;
lvalue ::= identifier "^" attributename
rvalue ::= literal | parameterised_instance | lvalue

(* Complex types *)
parameters ::= "(" list ")"
array ::= "[" list "]"
list ::= list_item |  list_item "," list
list_item ::= parameterised_instance | literal
parameterised_instance ::= identifier [parameters]

(* Base types *)
iptype ::= string
literal ::= number | string | array | boolean | "None"
boolean ::= "True" | "False"

(* Identifiers are case insensitive, start with a letter, and contain
letters, digits or _. Strings are delimited with double quotes
numbers can be in decimal (687), hex (0x2AF) or binary (0b1010101111) *)
```

Figure 4.3: EBNF of AnvilADL's syntax

a feature an explicit type is optional. Each feature has a textual identifier that is used to refer to the feature elsewhere in the description. In AnvilADL, identifiers are case insensitive, must start with a letter, and after which can contain letters, digits or the underscore character.

Note that source layer elements (concurrent objects, etc.) are not in the above list of feature categories. When using AnvilADL, the programmer does not need to explicitly define the source layer objects as this information can be obtained from static analysis of the input code. The CTV system model is compile-time static, which means that the input code must be compile-time statically-analysable. Section 4.15 discusses the limitations that this places on Anvil's input code, but once these restrictions are met the analysis phase can scan the input code to determine the objects present in the code. Section 3.4.6 described in more detail the way in which the features of C with pthreads are represented by the CTV system model, but to summarise:

- Each call to `pthread_create` with a unique instance of a variable of type `pthread_t` creates a new thread. These are represented by concurrent objects. From an implementation perspective, the thread is created in the memory space of the processor to which the thread is mapped. Anvil's shared memory library (section 4.8) distributes data across non-uniform memory spaces to hide such details from the programmer.
- Instances of `pthread_mutex_t` and `pthread_cond_t`, created by `pthread_mutex_init` and `pthread_cond_init` respectively, are represented by shared objects. C does not allow the programmer to natively create their own shared objects, but an Anvil feature described in section 4.12 adds this capability.
- Global variables are represented as shared data items.

Assignments are used to give more detail about an architectural feature, to describe the topology of the target hardware, and to map source layer elements to logical and target layer elements. This is done by assigning values to the *attributes* of the declared features in the description. Attributes can be set by the programmer to add more information into the description. Each feature category defines a set of attributes which apply to all types within that category. For example, all processors have a `memory` attribute which is used to attach memory spaces to it. *Type attributes* are extra attributes that can be defined only for a given instance type. For example, a UART channel type can define a `baud_rate` attribute.

The category attributes are detailed below, and examples of attribute use are given in the following sections.

Processor attributes

Attribute name	Value	Purpose
`memory`	parameterised `memory` instance	Sets the main (boot) memory of the processor.
`extramemory`	parameterised `memory` instance	Defines an additional memory block for the processor.
`threads`	Array of threads	The threads that are directly mapped to this processor.
`cache`	Array of [instruction, data]	The types of caches used by the given processor.

Memory attributes

Attribute name	Value	Purpose
`size`	Number	Size of the memory space (words).
`width`	Number	Width of the memory space (bits).
`variables`	Array of shared data items	The variables that are directly mapped to this memory location.

Channel attributes

Attribute name	Value	Purpose
`endpoints`	array of parameterised processor instances	Defines the endpoints of the channel.
`bandwidth`	Number	Informal measure of average bandwidth, used by the routing algorithm (section 4.7).

Variable attributes

Attribute name	Value	Purpose
`mutex`	Name of mutex instance	The mutex associated with the shared variable (see section 4.8.3).
`readcache`	Size (bytes)	Size of the variable's read cache.
`writecache`	Size (bytes)	Size of the variable's write cache.
`fetchall`	Boolean	Fetch and update the entire variable using bulk transfers.
`targets`	Array of `target` instances	The targets that this variable is mapped to.

Cluster attributes

Attribute name	Value	Purpose
`contains`	Array of source level instances	Define the threads, mutexes, CVs, and variables that are part of the cluster.

Target attributes

Attribute name	Value	Purpose
`contains`	Array of `processor` and `memory` instances	Define hardware that makes up the cluster target.
`coherent`	boolean	Whether the processors of the cluster automatically kept cache coherent by the target architecture.

Manager attributes

Attribute name	Value	Purpose
`manages`	Array of threads, mutexes, CVs, shared variables	Create an OM that manages these items.
`executes_on`	Array of `processor` instances	The processor upon which the manager should execute.

Protected attributes

Attribute name	Value	Purpose
`functions`	Array of function names	The functions associated with the Anvil protected object (see section 4.12).
`state`	Array of variable names	The state of the protected object.
`identifier`	String	The instance identifer used by the protected object.

4.2.2 Processors and memory

The first step in creating a description is to declare the processors and memory spaces of the target architecture. For example, the following description creates two different types of processor and three memory spaces.

```
cpu1 : processor Microblaze;
cpu2 : processor Picoblaze;
cpu1^barrelshifter = True;
mem1, mem2, sharedmem : memory BlockRAM;
```

The declared processors are of type `Microblaze` and `Picoblaze` respectively. In addition, the programmer has used an assignment to set the Microblaze instance's `barrelshifter` type attribute to true to provide more detail to the compiler. `barrelshifter` is a type attribute used to denote whether or not the instance is synthesised with a barrel shifter or not, and the compiler can use this information to make use of the extra opcodes that require a barrel shifter. The memories are all declared as instances of type `BlockRAM`.

The next phase is to use the `memory` and `extramemory` attributes to describe the memory hierarchy. The `memory` attribute is used to define the memory space into which code and data for a given processor may be stored and must be set for all processors in the system. Processors may share memory, if supported by the specific memory type.

The `extramemory` attribute defines memory spaces that are connected but if possible should not be used for code or local data (the stack and heap). An `extraemory` memory space might be slower, or shared by multiple processors, so it is primarily intended for storing shared data items. Anvil will only store code and local data in `extramemory` spaces if the processor's other memory is full. Both `memory` and `extramemory` can be assigned an array of memory instances to describe more complex memory layouts. It is not currently possible to explicitly separate code and data storage, but this could be implemented in a future version if required.

When assigning memory spaces, it is necessary to describe the memory map of the processor. This is done by passing parameters to the `memory` and `extramemory` assignments, as shown in this example:

```
mem1^size = 8388608;
mem2^size = 8388608;
mem1^width = 32;
mem2^width = 32;
sharedmem^size = 256;

cpu1^memory = [mem1(0x00000000)];
cpu2^memory = [mem2(0x00000000)];
cpu1^extramemory = [sharedmem(0x02000000)];
cpu2^extramemory = [sharedmem(0x02000000)];
```

Here `mem1` has been assigned to `cpu1`, `mem2` to `cpu2`, and `sharedmem` as a block of shared memory that can be accessed by both processors. The sizes of each memory space have been given using the `size` and `width` attributes, which are measured in words and bits respectively. In this example, 8,388,608 32-bit wide words equals 32 MiB. The base address of the memory space → processor connection is given as a parameter during the attribute assignment. Note that most processors require boot code to be placed at its zero address so memory should be allocated to cover that address range. This is a hardware design issue however and Anvil assumes that the target architecture as described is correct so it does not check such issues and simply allocates code from the earliest possible address. If memory is incorrectly assigned or insufficiently to store the required program then this will be detected by the linker in most toolchains.

A number of consistency checks can be performed by the Anvil tools at this stage. Anvil will verify that each processor in the system is provided with at least one memory space and will inform the user if this is not the case. Also, the `size` and `width` attributes of each memory space are checked to ensure they are consistent with the memory map provided by the `memory` attribute of each processor. If these or similar checks fail, the user is informed and required to correct their description.

4.2.3 Communication topology

Once the processors and memory spaces of the target architecture have been specified, the communications topology can be defined. This information is required so that Anvil's refactoring engine can correctly route inter-thread communications over the appropriate hardware channels. Instances of category `channel` are defined and then connected to the processors of the system using the `endpoints` attribute, which is assigned an array of the channel's endpoints. Endpoints were defined in section 3.4.4 as the logical connection points that join processors to channels. Endpoints must be parameterised with the following information:

- The address at which the channel's endpoint is located in the processor's memory space. Expressed as a hexadecimal or decimal literal.
- Whether the above address is in I/O space or memory space (because some processors support a separate peripheral space). Expressed as either the string `"mem"` or the string `"io"`.
- The interrupt vector of the endpoint, or `"None"` if interrupts are not present.

The channel's type may also require that each endpoint is given extra parameters after the initial three. For example, a network often requires that each endpoint is assigned a unique ID (for example, a MAC address in Ethernet).

In the following example, a simple UART channel is created and connected between the two processors. The example also shows more examples of type attributes that the designer has used to provide extra information about the channel.

```
myuart : channel UART;
myuart^baud = 9600;
myuart^uartsettings = "8N1";
myuart^endpoints = [cpu1(0x80000000, "mem", 0), cpu2(0x80001000, "mem", 0)];
```

4.2.4 Custom hardware items

Custom hardware items (which describe hardware items from the target layer of section 3.4.4) are instantiated in a similar way to channels. While endpoints connect channels to processors, ports connect custom hardware items to processors. Ports require the same parameters as endpoints, with the addition that each port must be specified as either synchronous or asynchronous. The following example shows the instantiation of two custom hardware features, one which uses interrupts and one which does not (and so must be polled). The CTV system model does not require the use of interrupts, but it supports them when available.

```
cpu1, cpu2 : processor;

cb : hardware Callback;
cb^ports = [cpu1(0x80000000, "mem", 0)];
cb^callback = "functionname";

gpio : hardware GPIO;
gpio^ports = [cpu1(0x8000A000, "mem", None), cpu2(0x8000A000, "mem", None)]];
```

The first hardware feature is of type `Callback`, which is a small timer component that is used by a processor to schedule an interrupt after a variable delay provided by the programmer at run-time. The assignment to its `ports` attribute shows that the hardware feature exposes a single port, located at memory address `0x80000000` and interrupt vector 0. Because an interrupt vector is provided the port is asynchronous. The `Callback` type also defines a custom attribute `callback` which is set to the name of the function that should be called when the callback is triggered (”`functionname`” in this case). This is an example of the way that hardware elements can be brought up to the abstraction level of source code without requiring onerous implementation detail. This example also highlights that whilst channels are required to have at least two endpoints, custom hardware items may expose only one port. More information is given about custom hardware elements in section 4.10.

The second hardware feature is a general purpose I/O component used for external communications. It can be accessed from both processors so it is described with two ports. These ports are synchronous because they do not provide an interrupt vector (the assignment to `gpio^ports` sets the third argument to `None`). Interrupts are not used for this hardware feature.

4.2.5 DMA controllers

Some architectures include DMA controllers to allow the rapid movement of data between two memory locations. It is possible to fully describe a DMA controller as a custom hardware item, but this would require the programmer to activate it manually. So that the shared memory libraries (described in section 4.8) can automatically use any DMA engines present, a special `DMA` type is also provided as a subtype of the `hardware` type.

A DMA engine is described as an instance of the `DMA` type, and has the same `memory` attribute as a processor. The `ports` attribute is still used to describe which processors can access the DMA engine.

```
mem1, mem2 : memory BlockRAM;
dma : DMA Xilinx_DMA;
dma^memory = [mem1(0x00000000), mem2(0x10000000)];
dma^ports = [cpu1(0x80000000, "mem", 0), cpu2(0x80000000, "mem", 0)];
```

In this description, a DMA engine that can access both `mem1` and `mem2` is memory-mapped to two processors, `cpu1` and `cpu2`. The DMA engine’s type is `Xilinx_DMA`, which informs the refactoring engine which driver should be used.

4.2.6 The logical layer - clusters and hardware mapping

Once the processors, memory spaces, channels and custom hardware elements have been instantiated and their attributes set, the programmer has defined the target architecture to a sufficient level of detail. The next stage is to define the logical layer which consists of clusters and cluster targets. Recall from the description of the logical layer (section 3.4.3), the purpose of the logical elements is to identify the clustering of the source application by assigning program layer elements to clusters, and to identify suitable areas for the implementation of a cluster in the architecture by assigning target later elements to cluster targets. Clusters are then mapped to at least one cluster target. (Mapping to more than one target expresses run-time dynamism.)

In AnvilADL, logical elements are defined with the `cluster` and `target` feature categories. Instances of these categories do not have types.

As identified in section 3.4.3, a cluster target must contain at least one processor and memory space. Processors and memory spaces are assigned into a cluster target by setting the cluster target's `contains` attribute, as shown by the following example.

```
cpu1, cpu2, cpu3 : processor;
mem1, mem2, mem3 : memory;
myclustertarget : target;
myclustertarget^contains = [cpu1, cpu2, mem1];
```

When a processor is added to a cluster target, that processor's main memory (the memory assigned to the processor's `memory` attribute) is also implicitly added to the cluster target as it will be used by the processor. This is shown in the following example.

```
cpu1 : processor;
mem1 : memory;
cpu1^memory = [mem1];

myclustertarget : target;

myclustertarget^contains = [cpu1, mem1];
//The above line is equivalent to the following line. mem1 is added implicitly.
myclustertarget^contains = [cpu1];
```

Because a processor must have its `memory` attribute set and these memory spaces are implicitly added to the cluster target, this therefore guarantees that the target will contain useable memory spaces. Extra memory spaces that are not bound to a specific processor may also be added. The computation requirements of all items assigned to a cluster are notionally shared between all the processors of the cluster target to which it is assigned. If the cluster is assigned to more than one cluster target, it will only use one cluster target at any one point in time. Similarly, all the data storage requirements of all the items assigned to a cluster are shared between the memory spaces of the cluster target to which it is assigned.

Threads, shared objects and data items are assigned into clusters also by using the `contains` attribute. However, because the programmer is not required to define the source layer items, the values assigned to `contains` for a cluster will refer to objects defined in the source code rather than in the AnvilADL description. Consider the following example application:

```
int data1;
float data2[50];
pthread_t thread1;
pthread_mutex_t mutex;

int main(void) {
   pthread_create(&thread1, 0, func, 0);
   ...
}
```

This code declares a thread called `thread1`, a mutex shared object `mutex` and two items of shared data `data1`, and `data2`. These can be assigned to a cluster as follows:

```
mycluster : cluster;
mycluster^contains = ["thread1", "data1", "mutex", "data2"];
```

For clusters, the array assigned to `mycluster^contains` is an array of strings where each string corresponds to the name of an item in the input code. In the cluster target example the array assigned to `contains` was an array of feature identifiers. If the `pthread_t` instance is declared in a scope other than global scope, the programmer is required to use a disambiguating name.

```
void func(void) {
   pthread_t thread1;
   ...
```

In this code the thread `thread1` is not in global scope, so must be disambiguated using the period character "." and the function name as follows:

```
mycluster^contains = ["func.thread1"];
```

This technique can also be used to specify threads inside structure types, and the C array indexing operator "[ ]" can be added to uniquely specify a single instance in an array of threads.

As program size increases, the syntax for declaring clusters can become tedious. The syntax used in this thesis is sufficient, but further work might consider either improving this, or developing GUI and tool support to aid developers.

Once the input program has been developed, cluster assignments can be obtained from a variety of sources. For simple programs, programmer experience and code inspection will be sufficient. For larger systems it may be necessary to employ profiling, analysis, or search-based co-design techniques.

After the clusters and cluster targets have been defined, all that remains is to state which clusters should execute upon which cluster targets. This is done with the `targets` attribute of clusters. `targets` is assigned an array of cluster targets to specify the targets that it is permitted to migrate between.

```
clus1 : cluster;
targ1, targ2: target;
clus1^targets = [targ1, targ2];
```

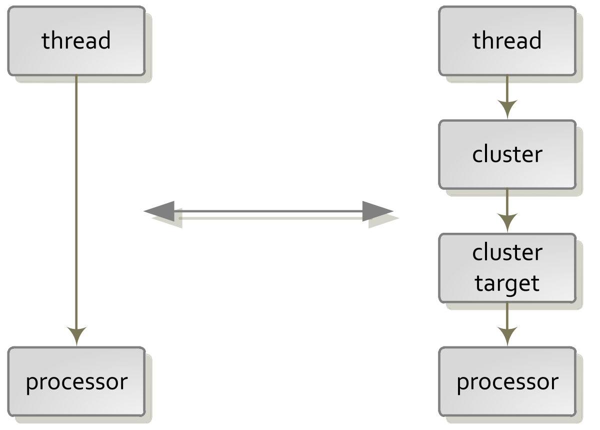

Figure 4.4: Direct mapping is equivalent to creating individual clusters for each mapped item.

4.2.7 Mapping features directly

The previous section detailed how the programmer maps source layer items to the processors and memory spaces of the target architecture using the mappings of the logical layer. Strictly, the CTV system model requires all threads, shared objects, and data to be mapped to clusters, which are then mapped to cluster targets and thereby the processors and memory spaces of the target architecture. AnvilADL fully supports this model, but for convenience it also allows the logical layer to be skipped in the areas of the system where dynamism is not required. This allows a more concise description of the system. Threads and shared objects can be directly assigned to processors, and shared data items can be directly assigned to memory spaces. Doing so is a shortcut for creating a single cluster that contains only the item being mapped, and mapping that cluster to execute on a cluster target that only contains a single processor and memory space. This is illustrated in figure 4.4.

Threads are mapped to processors directly through the use of the `threads` attribute of processor features.

```
void *threadfunc(void) {
   ...
}

int main(void) {
   pthread_t mythread;
   pthread_create(&mythread, 0, threadfunc, 0);
   ...
}
```

```
cpu1 : processor;
cpu1^threads = ["main", "main.mythread"];
```

In the above example, `cpu1` is assigned two threads. `"main"` is the thread that is running at the start of the C program. `"main.mythread"` is a disambiguating name for the pthread instance which is created by the main thread. Mapping variables to memory spaces is performed in a similar way, using the `variables` attribute of memory space features.

```
int globalvar;

int main(void) {
   int localvar;
   ...
```

```
mem1 : memory;
mem1^variables = ["globalvar", "main.localvar"];
```

4.2.8 Object managers

The final phase in creating an AnvilADL description is the definition and placement of the OMs in the system. In a CTV-based system all elements from the source layer must have exactly one OM, and each manager must be assigned to exactly one processor of the target architecture. Managers are defined by creating a `manager` feature and assigning program layer items to its `manages` attribute. The OM can then be mapped to a set of target processors by assigning it to a cluster (to support migration), or by setting its `executes_on` attribute (for direct mapping).

```
cpu1, cpu2 : processor;
om : manager;
clus : cluster;
om^manages = ["globalvar", "main.localvar"];

//Add the OM to a cluster
clus^contains = [om];
//OR map it directly
om^executes_on = [cpu1];
```

If an OM is assigned to a cluster then it is specified as a migratable OM (discussed in section 4.13.1) and at run-time it will be mapped to one of the processors of the cluster target upon which the cluster is currently executing. Anvil automatically selects offline a single processor from each cluster target to form the OMs *target processors* set. It is between these processors that the OM will migrate. The programmer can assign non-migrating OMs to a single processor using the `executes_on` attribute.

It is possible to use `executes_on` to ‘hint’ to Anvil which processors should host a migrating OM. By assigning an OM to a cluster *and* setting its `executes_on` attribute, the OM is declared as migratory, but the processors from `executes_on` are used as its target processors. In this case, `executes_on` should contain at least one processor from each cluster target.

$$\begin{aligned}&\forall o \in OMs \cdot \exists c \in clusters \cdot o \in c\hat{\ }\texttt{contains} \wedge \\ &\quad (\forall t \in clustertargets \cdot t \in c\hat{\ }\texttt{targets} \implies \exists p \in processors \cdot p \in o\hat{\ }\texttt{executes\_on} \wedge p \in t)\end{aligned}$$

To simplify the implementation of OMs, in Anvil the OM of a shared variable must be mapped to a processor that has access to that variable. Therefore, if a variable is mapped to a cluster its manager should also be mapped to the same cluster to allow the manager to migrate along with the data.

The problem of optimal OM specification is difficult. The number of OMs in the system, their locations, and the mappings of elements to OMs can have a large effect on the performance of the final system. An optimal assignment has to balance the following factors:

- Threads should be located close to the data they use most frequently.
- Threads that communicate frequently should be located close together, or ideally on the same processor.
- The OM of a system object is a single point of contact for all threads that wish to interact with that object. As a result the OM should be positioned so that it is close to the threads that most frequently access it.
- The more items that an OM must manage, the greater the burden on the host processor. Request latency will also increase accordingly.
- If data is accessed by threads on multiple physical processors, it should be assigned to shared memory to reduce copying time. This may require the accessing threads to be allocated to processors with access to shared memory.
- To avoid congestion of communication channels, tightly-coupled threads should be assigned so that they do not have to communicate over otherwise heavily-utilised links.

A multi-optimisation problem such this is challenging, and it is worsened by the fact that static analysis can only provide inaccurate indications of the complete system performance. As a result, any attempt to solve the optimisation must either use inaccurate performance information that may differ greatly from the characteristics of the actual system, or rely on very slow simulations. Work has been done to assist in this area. Modelling and simulation frameworks such as Artemis [181], StepNP [179] and MESH [178] have shown how both offline and online modelling can be used to better map applications over complex MPSoC platforms. These approaches would have to be extended so that they are aware of the OM communications layer. Existing work [128] has concentrated on the problems of evaluating task assignments in complex embedded systems. The processes and results presented in these systems could be usefully applied to CTV. Also, there is a lot of crossover from the techniques used in hardware / software co-design (as discussed in section 2.4.4). Because of the volume of work in this area, the problem of optimal OM assignment is considered to be outside the scope of this thesis.

4.3 Refactoring overview

This section documents the Anvil refactoring process. Recall that the purpose of the refactoring engine is to take a single program description of the application and refactor it into a set of programs that are specifically targeted at the processors of the implementation platform. The focus of this stage is to produce a low overhead implementation that is suitable for use in a resource-constrained embedded system. The process is depicted in figure 4.5 and can broken down into the following steps:

1. **Parse input program:** Cast the input program in terms of the CTV input model (section 4.4).

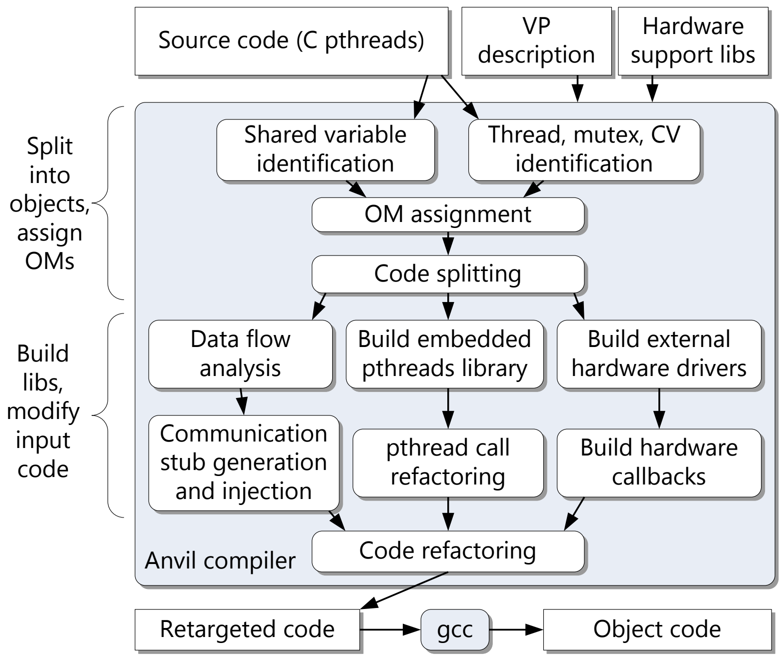
Source code (C pthreads)
VP description
Hardware support libs
Split into objects, assign OMs
Shared variable identification
Thread, mutex, CV identification
OM assignment
Code splitting
Build libs, modify input code
Data flow analysis
Build embedded pthreads library
Build external hardware drivers
Communication stub generation and injection
pthread call refactoring
Build hardware callbacks
Anvil compiler
Code refactoring
Retargeted code
gcc
Object code

Figure 4.5: The main processes inside the Anvil compiler

2. **Code splitting:** Split the single input program into a set of programs, one for each target processor (section 4.5).
3. **Build architecture-specific libraries:** Build an OM-aware shared memory system, an embedded pthreads implementation and custom hardware drivers. These libraries are processor-specific so a set is generated for each target processor (section 4.6).
4. **Refactor processor-specific programs:** to use the libraries generated in the previous stage.
5. **Compile:** Using standard `gcc` (section 4.11).

These stages are now described in more detail.

4.4 Refactoring: Parse input program

The objective of the first phase is to parse the input program and identify the threads, mutexes, condition variables and items of shared data that are used. These correspond to the concurrent objects, shared objects, and shared data of the CTV system model. Recall that the developer does not define these in their AnvilADL, they are determined from the input source. The parser used is an LALR parser implemented using PLY [19], an implementation of the lexical analyser Lex and the parser generator Yacc for the Python programming language. The input C code is parsed in a single pass, taking special care to ensure that after a `typedef` expression has been successfully parsed the new type name is fed back into the lexer so subsequent instances of the type name generate a TYPE token rather than an IDENTIFIER token. The C preprocessor cpp must first be applied to the input code as the parser does not implement preprocessing directives.

Anvil parses the incoming source code, generating the abstract syntax tree and annotating it with symbol tables at each scope. In the representation chosen by Anvil, symbol tables can be located at the top of every compound statement. The generated AST of a small example program is shown in figure 4.6.

From this AST, the refactoring engine recursively searches the symbol tables for variables of type pthread_t, pthread_mutex_t and pthread_cond_t. This identifies the threads, mutexes and condition variables used in the program. If an array of these types is found, the array size must be compile-time static. Lists, and other data structures, are currently not supported. This method finds all possible instances of these types, even if a given execution of the program will not instantiate them all due to run-time branching.

Shared data variables are normal variables, but that are accessed by multiple threads. This is more challenging to identify and requires the refactoring engine to perform three tasks:

- The *thread bodies* of the system are identified. A thread body is the function that is passed to a pthread_create call to start a thread. Instances of pthread_t must be matched with their thread bodies to allow the refactoring engine to determine the entry point for each thread in the system.
- Callgraphs are built for each thread. From this, the engine determines the code that could possibly be accessed by each thread. As discussed in section 4.4.2, this phase requires that if function pointers are used they are declared constant.
- The callgraphs are analysed to determine which non-local data items are potentially accessed by more than one thread.

These three stages are described in the following sections.

4.4.1 Identify thread bodies

When using pthreads, a thread is not started until a pthread_create call is issued. This call takes four arguments, but of interest here are the first and third arguments which are respectively the pthread_t instance of the thread being created and a pointer to the function that should be used as the thread's body. The refactoring engine needs to statically determine these pairings so that it can perform static analysis on the code of each thread. This example shows the simplest case of this problem:

```
int main(int argc, char *argv[])
{
    int x;
    return 0;
}

translation_unit[1]
.  Symbol Table
.  .  main (function): TS=['int'] TQ=[] SCS=[] Line=tree.c [2-6]
.  function_definition[3]
.  .  Symbol Table
.  .  .  argc (object): TS=['int'] TQ=[] SCS=[] Line=tree.c [2-2]
.  .  .  argv (object): TS=['char'] TQ=[] SCS=[] Line=tree.c [2-2]
.  .  declaration_specifiers[1]
.  .  .  type_specifier = "int"
.  .  declarator[1]
.  .  .  direct_declarator[2]
.  .  .  .  direct_declarator = "main"
.  .  .  .  parameter_type_list[1]
.  .  .  .  .  parameter_list[2]
.  .  .  .  .  .  parameter_declaration[2]
.  .  .  .  .  .  .  declaration_specifiers[1]
.  .  .  .  .  .  .  .  type_specifier = "int"
.  .  .  .  .  .  .  declarator[1]
.  .  .  .  .  .  .  .  direct_declarator = "argc"
.  .  .  .  .  .  parameter_declaration[2]
.  .  .  .  .  .  .  declaration_specifiers[1]
.  .  .  .  .  .  .  .  type_specifier = "char"
.  .  .  .  .  .  .  declarator[2]
.  .  .  .  .  .  .  .  pointer = "*"
.  .  .  .  .  .  .  .  direct_declarator[2]
.  .  .  .  .  .  .  .  .  direct_declarator = "argv"
.  .  .  .  .  .  .  .  .  constant_expression_opt
.  .  compound_statement[2]
.  .  .  Symbol Table
.  .  .  .  x (object): TS=['int'] TQ=[] SCS=[] Line=tree.c [4-4]
.  .  .  declaration_list[1]
.  .  .  .  declaration[2]
.  .  .  .  .  declaration_specifiers[1]
.  .  .  .  .  .  type_specifier = "int"
.  .  .  .  .  init_declarator[1]
.  .  .  .  .  .  declarator[1]
.  .  .  .  .  .  .  direct_declarator = "x"
.  .  .  statement_list[1]
.  .  .  .  return_statement[1]
.  .  .  .  .  expression[1]
.  .  .  .  .  .  constant = "0"
```

Figure 4.6: AST produced by Anvil from a simple code fragment.

```
void *threadfunc(void *a) {
   ...
}

pthread_t thread1;

int main(void) {
   pthread_create(&thread1, 0, threadfunc, 0);
}
```

In this example `thread1`'s body is the function `threadfunc`, and this can easily be determined because the arguments of the `pthread_create` call are direct references to the thread instance and the function body. This task becomes more difficult when more complex programs are encountered.

The general problem of determining the values of the arguments passed to `pthread_create` is one of data-flow analysis, particularly that of determining *reaching definitions* [4]. Every assignment is a definition. A definition d *reaches* a statement s in the program if there is a path from d to s in which d is not overwritten. Paths are determined by looking at every basic block of the program and noting the definitions at the start of the block and at the end. Essentially the reaching definitions are a list of all the assignments that can have determined the value of a given variable. To determine the pairings of thread instances and their body functions, it is necessary to determine the reaching definitions of the first and third arguments of every `pthread_create` call. Then, all possible pairings generated by that `pthread_create` call are the cross product of those two sets of reaching definitions. Frequently this will result in the situation where a given thread instance might be assigned any of a set of thread bodies.

A system which implemented the algorithm above could accept largely unrestricted input code, but can lead the programmer into inadvertently developing a very inefficient system. If the reachability analysis cannot uniquely determine a thread's body function, it must assume that all possible functions might be used and compile them all for that thread. In turn, this pessimism will propagate throughout the later analysis stages. Consequentially, this version of Anvil imposes restrictions on calls to `pthread_create` that force the programmer to make such situations explicit.

In Anvil, each individual `pthread_create` call must uniquely identify a thread instance and body function, and those must be declared either at global scope or as a static variable. For each `pthread_create` call, the first and third parameters are resolved by recursively looking up through the stack of symbol tables in the AST. If they do not resolve to an instance of the type `pthread_t` and a function respectively in the global symbol table then an error is raised. This does not severely limit the programmer's expressibility, however, because multiple `pthread_create` calls are still allowed for each `pthread_t` instance. Should the programmer wish to define a program in which the body of a thread is determined at run-time, they can do so as follows:

```
//body1 and body2 not shown

int main(void) {
   static pthread_t thethread;

   if(some_condition)
      pthread_create(&thethread, 0, body1, 0);
   else
      pthread_create(&thethread, 0, body2, 0);
}
```

In this case, the refactoring engine allows `thethread` to have two possible implementations, but has forced the programmer to do state this explicitly. Another potential problem arises from the use of arrays of `pthread_t` instances. Consider the following code:

```
int main(void) {
   int y;
   static pthread_t thethreads[20];

   y = some_function();
   pthread_create(&thethreads[y], 0, threadbody, 0);
}
```

Once the refactoring engine determines after constant propagation that the index of `thethreads` is not compile-time constant it adds `threadbody` as a potential thread body for all indices of `thethreads`. Equally, if the third argument to `pthread_create` is an array with a non-constant index then all functions of the array are set as potential thread bodies. These constructs are allowed because it is a common paradigm to set up an array of worker threads all with the same body. Further code analysis could place bounds on the possible values of the loop variable and thereby improve the accuracy of this algorithm, but as it is not the focus of this work these are not explored further and are left as future work.

4.4.2 Generate call graphs

Once the thread bodies of the system have been determined the analysis generates the potential call graphs of each thread. This is performed statically using the following algorithm, which is an instance of a standard depth-first search with edge colouring to prevent cycles:

```
For each thread instance:
    Push all thread bodies (identified in the previous step) onto a stack

    While stack is not empty:
        Pop the top function off the stack
        Add the popped function to callgraph for this thread instance
        Examine the function, identifying all the call sites
        For each call site:
            If the function has not already been pushed onto the stack for
            this thread instance:
                Push it onto the stack
```

The search keeps a list of previously-visited nodes to ensure that call cycles are not followed infinitely. A list is built for each thread instance showing the subset of the input code that it covers. This algorithm does not operate correctly when non-constant function pointers are passed between threads and functions. For this reason, Anvil does not allow variable functions pointers to be used in the input code. The pointer passed to `pthread_create` should be constant.

After application of the search, the resulting graph can be output using the DOT graph visualisation format for manual inspection, an example of which is shown in figure 4.7.

4.4.3 Determine shared data items

Once the callgraphs of each thread have been built, the analysis can determine which variables are shared between multiple threads and therefore need to be modelled by the CTV system model. The following algorithm is applied:

```
For each function in the input application:
    For each statement in the function:
        For each identifier:
            Follow the identifier back to the symbol table entry it refers to.
            (This allows the analysis to correctly respect scoping rules)
            Annotate the symbol table entry with the name of the current
            function.
```

Once each variable has been annotated with the functions that access it, this information can be trivially combined with the callgraphs generated by the previous step to determine which variables are accessed by multiple thread instances.

Anvil relies on accesses to shared data being statically analysable. However, arbitrary pointer-based access to data is the general case unanalysable. This is a common problem encountered in real-time and safety-critical systems, as verification relies on the behaviour of the code being analysable. Real-time subset languages (or coding styles, such as MISRA C [220]) already carry such a restriction. For example, rule 104 of MISRA C states that ”Non-constant pointers to functions shall not be used” and places restrictions on pointer casting and other non-analysable operations. Anvil is not quite as restrictive as MISRA C as it is intended for use in the entire embedded domain rather than just safety-critical systems, but some of the pointer restrictions are similar.

Note that this problem is largely unique to C because of the low-level access to memory addresses that it permits. Languages like Java and Ada already prevent pointer arithmetic because of this problem, instead using immutable object references and access types respectively.

Using pointers to access shared data is very common in C so Anvil does not simply disallow pointer use. Instead, Anvil allows pointer-based access to shared data providing that the pointer is initialised to point to the data item in question. If the pointer is initialised to a variable,

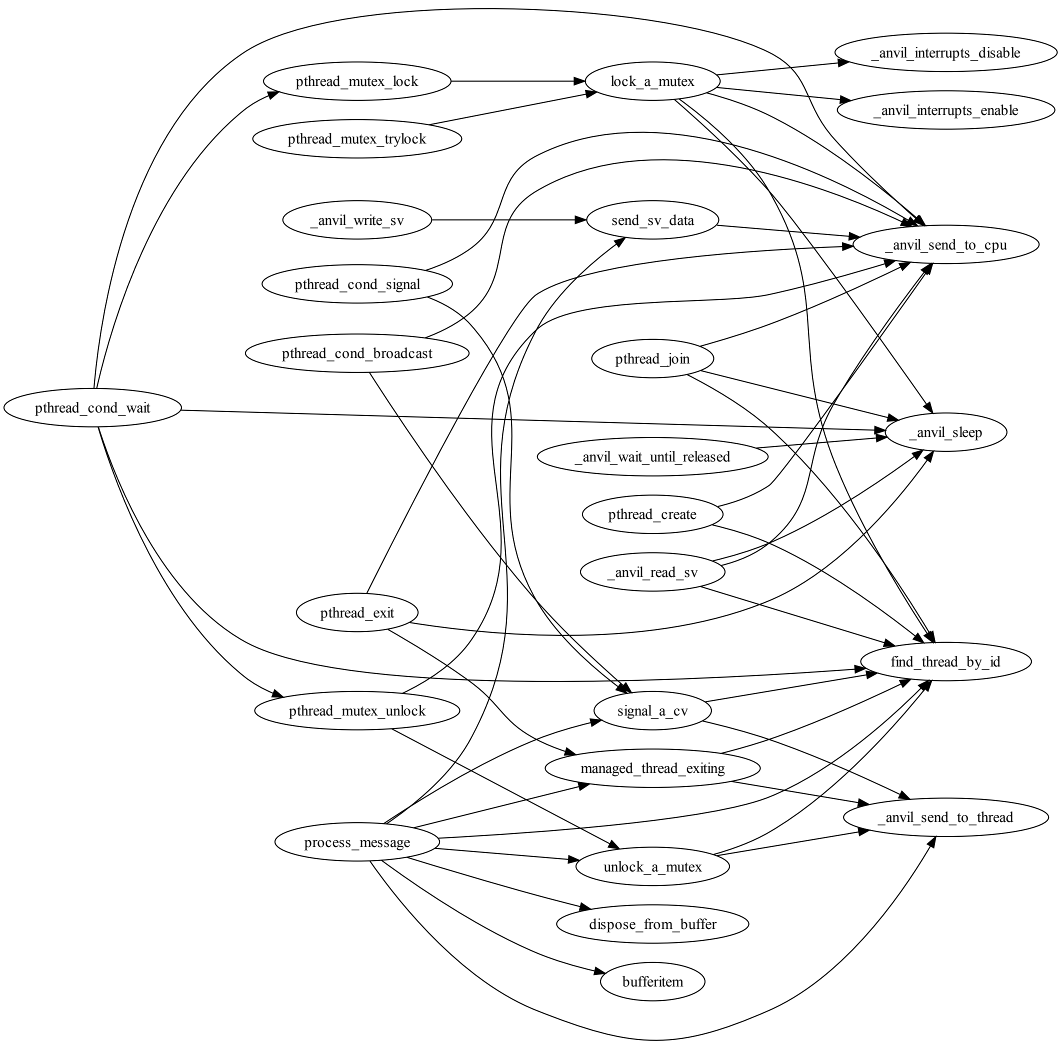

Figure 4.7: An example callgraph generated during the code splitting phase.

it can be followed through the source code using data flow analysis[1].

The following algorithm is used:

- The analysis routine searches for points in the code where pointer types are created that are initialised to point to variables identified as shared.
- For each shared data pointer p initialised to point to shared variable s:
  - Annotate p in the AST as a reference to potentially-remote data s.
  - For the scope of p, determine any other pointers to which p is assigned. This includes function calls into which p is passed as an argument.
  - For each pointer or argument, mark the assignment target as a reference to s. (Function arguments may reference more than one shared item after this algorithm has completed if the function is called from multiple places.)
- Keep recursively applying this algorithm until all pointer references have propagated throughout the application.

Clearly this algorithm will mark large amounts of the program if p is global and commonly-used, and in this case the input program will have to be restructured or the inefficiency accepted. More advanced analysis techniques such as context-sensitive pointer analysis can be used to reduce this problem. Pointer arithmetic is not allowed to change the pointer's target to a different object. For example, in the following code the pointer is assumed to still refer to a section of the shared data after the assignment.

```
int shareddata[50];

int *mypointer = shareddata;
//mypointer is annotated to be a reference to shareddata
mypointer = mypointer + 7;
//mypointer is still a reference to shareddata
```

Equally, the following is not allowed:

```
int shareddata1[50];
int shareddata2[50];

int *mypointer1 = shareddata1;
int *mypointer2 = shareddata2;

//This is not allowed as it changes mypointer2's target object
mypointer2 = mypointer1;
```

The complete set of restrictions imposed on input code by Anvil are listed in section 4.15.

[1]A similar restriction is used to allow pointer-based access to threads, mutexes and condition variables, as described in section 4.9.

4.4.4 AnvilADL consistency checks

After the above parsing stages have been completed, the AnvilADL must also be parsed. As AnvilADL is a much simpler language than C, the parser only has to extract from the source code the set of declared features and the attribute assignments associated with each one. Once complete, the following consistency checks are performed against the AnvilADL and the input source:

- Assignments must be correctly typed. If an assignment refers to a feature then that feature must already have been defined and be of the appropriate type. For example, in the assignment `cpu1^memory = mem1`, the object `mem1` must be have been defined and of type `memory`.
- Any source language items that are referred to in the ADL must exist and be of the correct type. For example in the assignment `cpu1^threads = ["thread1"]`, there must be at global scope an instance of the `pthread_t` type called `thread1`.
- Each thread, mutex, condition variable, and shared data item from the source language must be assigned to exactly one OM. An OM may manage multiple items. (From the OM model of section 3.7)
- Each OM must be assigned to at least one processor. (From the OM model of section 3.7)
- Each processor must have its `memory` attribute set to an appropriate memory space.
- Each thread must be assigned to either exactly one processor using the `processor`'s `threads` attribute, or to a cluster using the `cluster`'s `contains` attribute. (From the definition of clusters in section 3.4.3.)
- Similarly, each shared data item must be assigned to either exactly one memory space using the `memory`'s `variables` attribute, or to a cluster using the `cluster`'s `contains` attribute. (From the definition of clusters in section 3.4.3.)
- A shared variable's OM must be assigned to a processor that has access to the memory space that contains the variable.
- Each cluster must be assigned (by setting its `targets` attribute) to at least one cluster target. (From the logical layer definition of a cluster, section 3.4.3.)
- Each cluster target must contain at least one processor and at least one memory space. (From the logical layer definition of a cluster, section 3.4.3.)
- Each channel must have at least two defined endpoints. (From the target layer model of section 3.4.4.)
- Each hardware item must have at least one defined port. (From the target layer model of section 3.4.4.)
- For each feature, any required type attributes must be set.

- If a shared variable can be migrated (it is assigned to a cluster with multiple cluster targets) then it must also have an associated mutex (see section 4.8.3) or data may be lost during a migration. (This restriction is due to the implementation of migration and is discussed in section 4.13.)
- By assigning an OM to a cluster *and* setting its `executes_on` attribute, the OM is declared as migratory, but the processors from `executes_on` are used as its target processors. In this case, `executes_on` should contain at least one processor from each cluster target.

If these checks all pass then parsing is complete and the refactoring process can begin.

4.5 Refactoring: Code splitting

The code splitting stage is the first phase of the refactoring process that actually manipulates the input code. At this stage, the application is expressed as single input program. It is necessary to split this program into a set of output programs, one for each target processor of the system. The AnvilADL description states which processors the threads of the system should be mapped to, and to which memory spaces items of shared data should be mapped.

To create the processor-specific programs, Anvil performs the following actions:

- The processor-specific programs needs to contain the source code for all the threads that may be scheduled on the processor. To determine this, for each processor p defined in the AnvilADL read the `p^threads` attribute. Then add to this list the threads that may migrate to p. This is the set of threads in cluster c, where c is assigned to cluster target t and p is in t. The union of these two thread sets is called the processor's *thread set*.
- Create an empty source file that will hold the code for this processor.
- For each thread in the processor's thread set, traverse its callgraph and add all encountered functions into the source file. This step must also include prototypes of functions that are prototyped and copy any required typedefs. The result of this stage is effectively the same as dead-code removal [4] and could be similarly achieved by copying across a custom-generated main function that calls the thread bodies of the processor's thread set. Any other unused functions would be removed.
- If the main thread is in the processor's thread set then copy across the existing `main` function.
- If the main thread is not in the processor's thread set then create a new, custom `main` function that initially sleeps, waiting for a message to wake it up. This wake up message is described in appendix A and is send by another thread in the system calling `pthread_create` with the appropriate arguments. It then calls the specified body function. For threads that have a set of possible body functions, the function to call is specified by the `pthread_create` message.

If more than one thread is scheduled for execution on the processor, a microkernel is linked in to provide minimal scheduling and localised OS services. Currently, Xilinx's xilkernel [256] is used as it is very lightweight and can be configured to provide the absolute minimum of required features.

4.6 Refactoring: Build architecture-specific libraries

The output of the code splitting stage is a set of programs, trimmed so that they contain only the code that is required for the threads of each target processor. These programs will not execute correctly because they may reference non-local shared memory or use the POSIX pthreads library which does not support distribution over complex architectures. To solve these problems, four architecture support libraries must be built that handle these problems, and then the code of each program refactored to make appropriate use of the newly-created libraries. The libraries are:

- An OM-aware communication library (section 4.7) – Allows the transfer of messages between the OMs and processors of the system. All other libraries use this layer as a base to implement their higher-level algorithms.
- An OM-aware shared memory system (section 4.8) – Handles remote data access, cache coherency and data marshalling, using the OMs to provide a distributed solution that avoids a single point of contention (as is experienced when using a standard OS).
- An embedded pthreads library (section 4.9) – Issues pthreads calls across a non-uniform architecture for manipulating threads, mutexes and condition variables using the OMs of the system.
- Driver code for custom hardware elements (section 4.10) – Allows the manipulation of custom hardware elements from within the C programming model without the need for low-level programming.

These libraries are processor-specific. For every processor in the system the libraries are generated and the processor's code is refactored to make use of its specially-generated libraries. This allows the code throughout the system to be optimised to use the memory, communications and hardware that is available to each specific processor. The following sections detail the specification and generation of these libraries.

4.7 Communications layer

Section 3.6.2 discussed the need for the OMs to implement a universal communications layer that allows all threads to communicate transparently. This is done by implementing a compile-time multistage permutation routing system [216] (see section 3.6.2). Each processor of the system (that is involved in a communication) has a small communications kernel added to its interrupt handler to allow the processor to fetch and forward messages. Some processors will be used as routers, forwarding messages between otherwise separate areas of the architecture.

In the Anvil system, routes are calculated offline. Because of this, it is necessary to create routes for all the communications that are required by the system. At this level, a communication is considered to occur between *processors* rather than between higher-level software entities such as threads or shared objects. When a request is encountered to send a message to a given thread, the thread is first resolved into the ID of its host processor. Doing this has a number of advantages:

- By routing communications between processors, the problem of routing communications remains purely an architectural concern and the software does not have to be considered.
- When multiple software elements are situated on the same host processor, it is wasteful to store identical routing information for each element.
- If software elements can migrate between processors, this occurs at an abstraction level above that of processor-to-processor communications. The problem of handling this migration is neatly confined to the problem of mapping a software element to its host processor. Once this is done, the rest of the layer can implement the communication. This separation allows the communications layer to be built in a modular fashion in which one module implements the physical communications and the other keeps track of where each software element is located throughout the system.

These two modules will be discussed in the following sections.

4.7.1 Implementing inter-processor communications

The set of all possible communicating processor pairs in a system is the cross product $c \times o$, where c is the set of all processors in the system and o is the set of all processors that have at least one OM mapped to them. C's programming model, and CTVs use of thread helpers, means that communications are always between a processor and an OM (which is hosted by a processor). This observation reduces the number of routes that it is necessary to calculate.

The current Anvil implementation enumerates routes for all the above pairs, but as Anvil is used to target larger architectures this will become infeasible. However, two methods exist to further limit the routes that need to be calculated. First, it is often possible to determine statically exactly which processors a given processor will communicate with and only calculate routes for those. A communication is initiated between two processors a and b exactly when:

- A thread mapped to a accesses a variable that is managed by an OM mapped to b.
- A thread mapped to a manipulates a mutex, condition variable or thread helper object that is managed by an OM mapped to b.

Much of this information is extracted during the initial parsing stage, and by techniques detailed in the discussion of the shared memory system (section 4.8) that deal with pointers. This information can be used to trim the calculation of routes that are never used.

Secondly, as architectures grow their communications requirements are increasingly handled by on-chip networks or multi-drop buses. For such channels, the individual nodes on the channel can be viewed as equivalent. Consider an ethernet network of four processors. There are 16 possible routes in this system (4 nodes × 4 nodes) but because they are all on a shared communications medium only one route needs be to calculated. The destination address for each packet will differ, but the route is exactly the same (send the packet to the network interface, it will appear at the correct node). Communication equivalence drastically reduces the required routing calculations.

Calculating an appropriate route is performed using an implementation of Dijkstra's algorithm [62]. Weights are assigned to the channels of the system based on their average throughput, and the algorithm is implemented to find a route which maximises the cost (meaning that it uses channels with the largest throughput). The programmer can manually change these routes after the code generation phase if necessary.

Once a route is calculated, it must be stored in the system in a way that makes it available at run-time. Broadly, there are three ways of doing this:

- Centralise routing information in a global 'route manager'. Flexible and saves memory throughout the system, but causes a large bottleneck so is not considered.
- Add routing information to the sender. Senders have to store entire routes for every processor that they may communicate with, and they have to embed that information in the message, but intermediary nodes do not need to store this information.
- Distribute routing information throughout the nodes of the system. The code for each processor is augmented with routing tables that are consulted whenever a message has to be transmitted forwarded. This means that messages can be small as they do not have to carry routing information, but each processor must store the first stage of a route to every other processor.

Anvil currently implements the third option listed above. Each processor is required to store a routing table that contains entries for all other processors in the system. The entry for processor p describes the channel that is used to send a message to p. This is similar to the routing tables that are stored by standard Ethernet routers. The same techniques described above to reduce the number of routes that must be calculated can be used to reduce the size of the routing tables. The table for processor a only needs to contain an entry for processor b if either:

- a communicates directly with b. The actions that cause a communication are enumerated above.
- There exists a route in the system in which b directly follows a. In this case, a will be required to forward a message to b, which is equivalent to a direct communication.

Anvil does not implement a fault-tolerant communications layer so the records in the routing tables are currently constant. A system that can cope with individual processor failures could be implemented on top of this layer but this is not considered in this thesis.

These tables are small. Each table entry is only a single byte, resulting in a tiny overhead for current embedded systems of only tens of cores. The footprint of this routing information is much smaller than the memory footprint of an equivalent middleware solution such as a CORBA Orb. Still, as architectures grow larger, CTV's system model allows the communications layer to maintain scalability through the information provided by the clustering model. Each cluster can be assigned a 'gateway' processor that is used to perform inter-cluster communications. Then, processors need only store routing information for the other processors in their cluster. Inter-cluster communications are passed to the gateway processor which can communicate with other gateways. This is identical to the hierarchical routing model used by the internet. The only requirement is that all processors have a globally-unique identifier (equivalent to the internet's IP addresses). This routing model is not implemented in the current version of Anvil, but can be once the sizes of embedded systems have grown enough to warrant its implementation.

With the routing tables created, Anvil builds the actual code that implements the communications layer. It exposes the following functions:

- `_anvil_send_to_thread(int targetthreadid, int *packet, int len)`:
  - `targetthreadid`: the global ID of the target thread
  - `packet`: a pointer to a character buffer containing the packet data
  - `len`: the length in bytes of the packet data

  Used to send a message to a thread of the system. The thread must be first resolved to a processor ID, and the communication is then implemented using a call to `_anvil_send_to_cpu`. The resolution step is discussed in the following section.

- `_anvil_send_to_cpu(int cpuid, int *packet, int len)`:
  - `cpuid`: the global ID of the target processor
  - `packet`: a pointer to a character buffer containing the packet data
  - `len`: the length in bytes of the packet data

  Sends a message to a given processor of the system. Processors are given globally-unique identifiers so that they can be identified at run-time. The implementation of this function is simply a large case statement that calls the appropriate channel driver to send the message.

An example of the generation of `_anvil_send_to_cpu` is below. The target architecture is described using this AnvilADL description:

```
cpu0, cpu1, cpu2, cpu3, cpu4 : processor;

c0to1 : channel mbox;
c0to1^ports = [cpu0(0x80000000), cpu1(0x80000000)];

c0to2 : channel mbox;
c0to2^ports = [cpu0(0x80001000), cpu2(0x80001000)];

c0to3 : channel can;
c0to3^ports = [cpu0(0x80002000, 1), cpu3(0x80002000, 2)]; //Second argument is priority

c0to4 : channel uart(115200, 8, False, 1); //115200 baud, 8N1.
c0to4^ports = [cpu0(0x80004000), cpu1(0x80004000)];
```

Below is the version of `_anvil_send_to_cpu` compiled for processor `cpu0`. (We are assuming that `cpu0` is assigned ID 0, `cpu1` is assigned ID 1 etc.) The function does nothing if called to send a message to processor 0. This is because to prevent unnecessary buffer copying this situation is handled at higher levels of the library stack. For the other processor IDs, an appropriate driver function is called to send the message over the channel selected by the routing algorithm (which is trivial on this architecture). Channel drivers are stored in an external archive in a uniform format that makes them easy to be used from generated code. Note that the parameters for the driver functions are given in the port definitions of the AnvilADL.

```
void _anvil_send_to_cpu(int targetcpuid, int *packet, int len) {
 int x;
 for (x = 0; x < len; x++) {
  switch(targetcpuid) {
   case 0:
    break;
   case 1:
    mbox_write(0x80000000, packet[x]);
    break;
   case 2:
    mbox_write(0x80001000, packet[x]);
    break;
   case 3:
    can_write(0x80002000, 1, 2, packet[x]);
    break;
   case 4:
    uart_write(0x80004000, packet[x]);
    break;
   default:
    break;
  }
 }
}
```

The other processors have to use multistage routing to communicate with any processor other than `cpu0`. `cpu0` is used to forward messages between the other processors. As a result, `cpu1`'s version of `_anvil_send_to_cpu` is as follows:

```
void _anvil_send_to_cpu(int targetcpuid, int *packet, int len) {
   int x;
   for (x = 0; x < len; x++) {
      switch(targetcpuid) {
         case 0:
         case 2:
         case 3:
         case 4:
            mbox_write(0x80000000, packet[x]);
            break;
         default:
            break;
      }
   }
}
```

4.7.2 Resolving software elements to host processors

It is necessary for the communications layer to be able to resolve a given software element (thread, mutex, condition variable etc.) to the processor upon which they are implemented. Commonly this information is known at compile-time because embedded systems are largely static systems. In this case, this resolution is trivially implemented during code refactoring. However, when migration of software elements is permitted the resolution becomes more complex and must include run-time support.

As observed in the previous section, the CTV system model ensures that all communications are between a thread and an OM. Therefore there are two situations that must be accounted for:

1. When a thread communicates with a (non-static) OM it must be able to resolve the current location of the OM.
2. An OM must be able to communicate with the items that it manages.

Note that this means that threads do not need to be able to resolve the location of other threads. The responsibility for doing this falls to their OMs.

Considering the first problem, algorithms for solving this have already been discussed in section 3.7.1. Anvil currently implements the *location propagation* algorithm, in which the position of each OM is stored in all processor nodes of the system. When an OM migrates, it must update all processors with its new position. This algorithm means that migrations are expensive, but that resolutions are very fast.

Only the position of migratable objects needs to be stored, and only on processors that access that object. Still, as systems grow in size a more scalable approach must be used. As with the routing tables of the previous section, the clustering information of the CTV system model can be used to good effect. A single processor in each cluster can be assigned the role of tracking migrations and resolving program elements to their host processors. The rest of the cluster

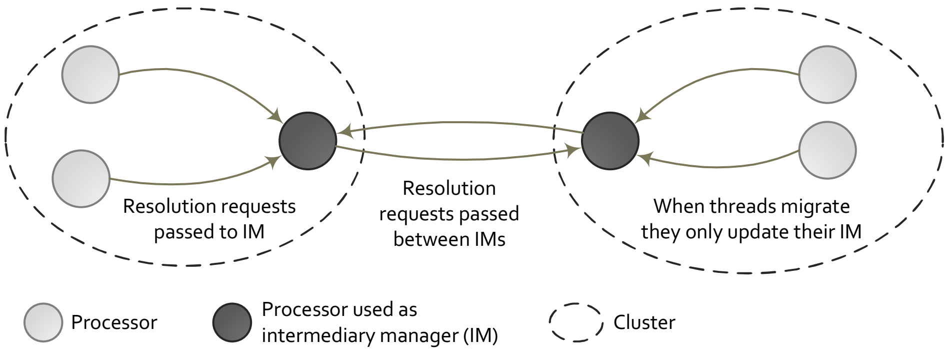
Resolution requests passed to IM
Resolution requests passed between IMs
When threads migrate they only update their IM
Processor
Processor used as intermediary manager (IM)
Cluster

Figure 4.8: Using clustering to reduce resolution requirements when threads can migrate.

can statically-route to this resolution service, and inter-cluster resolutions can therefore be performed hierarchically according to the clustering structure of the application. This reduces the memory overhead considerably (only one core is affected) and resolution time is kept small because most resolution requests are intra-cluster communications, which are fast. This is essentially the *intermediary manager* algorithm from section 3.7.1, but applied hierarchically and with clustering knowledge to ensure efficient mapping. This is illustrated in figure 4.8.

The second problem is simpler and is solved by ensuring that an OM tracks the current location of the items that it manages. As described in section 4.13 (which discusses the way that the actual migration of threads and data items is implemented in Anvil) it is an object's manager that performs its migration. Therefore an OM will always be able to keep track of the current positions of its managed items, and therefore it will always be able to communicate with them.

4.8 Shared memory system

The primary goal of the shared memory system is to allow the VP to present a single logical address space to the source language because this is the memory model assumed by C. As detailed in section 3.6.2, this requires the implementation of an object-based distributed shared memory system [211, 174] that can allow the threads of the system to share data efficiently and coherently.

This library is required to ensure correct behaviour in the following two situations. In all cases it is assumed that the program already exhibits correct concurrent behaviour without race conditions.

1. A thread is accessing data that is not locally-addressable (i.e. it is not directly connected to the memory bus of the processor) and may be shared between multiple threads. In this case, the OMs of the system must cooperate to pass data between the processors of the system to give the illusion that all data is available to each thread.
2. A thread is accessing data which is locally-addressable, but is shared between multiple processors. In this case, data does not have to be moved by the OMs, but if the processors have caches then coherency must be considered.

In the case where the thread is accessing data that is locally-addressable and not shared between multiple threads the existing language implementation will operate as expected.

The shared memory system implemented by Anvil uses migration semantics. Migration semantics state that when a thread is accessing an item of shared data the canonical version of the data is migrated to it for the duration of the access. In practice, this means that the data (or part of it) is logically transferred to the memory space of the accessor thread, which is free to operate on the data in any way before releasing it. Upon release, the data is logically migrated back to its original location. Logical migration means that the data does not strictly have to be moved. Only *control* of the data is moved.

These semantics were chosen because they are compatible with the mutex-based coordination of pthreads-based programs. Migration semantics are not specified by the CTV model and other schemes may equally have be implemented. For example, for languages with coordination semantics that encourage more finely-grained parallelism (such as Occam [120]) migration may be too coarsely-grained.

4.8.1 Library interface

The shared memory library presents two functions for reading and writing shared memory items:

- `int _anvil_read_sv(int svid, int offset, int size, int managercpu)`
  - `svid`: the global ID of the shared variable to read
  - `offset`: offset in bytes to start reading from
  - `size`: the number of bytes to read
  - `managercpu`: the global ID of the processor which hosts the OM of this shared variable
  - Returns the number of bytes read.
- `int _anvil_write_sv(int svid, int offset, int size, int managercpu)`
  - `svid`: the global ID of the shared variable to write
  - `offset`: offset in bytes to start writing to
  - `size`: the number of bytes to write
  - `managercpu`: the global ID of the processor which hosts the OM of this shared variable
  - Returns the number of bytes written.

Recall that each shared variable is managed by an OM, which is implemented as an interrupt handler on a processor which has direct access to the shared variable. Therefore, these functions are essentially implemented as remote DMA requests. When the target processor (that hosts the variable's OM) receives the request they will either write into or read from their local memory as instructed.

Communications are passed over the OM communications layer (using `_anvil_send_to_cpu`) so details such as routing and packaging of messages are all handled. As a result, the implementation of these functions is very simple. `_anvil_read_sv` sends a message to the variable's handler, and waits for a set of reply messages containing the data it requested. `_anvil_write_sv` sends the written data back to the OM, which stores it.

The first argument to each function is the `svid`, or shared variable ID. This is a globally-unique identifier which is assigned to every shared variable in the system and is passed between OMs to identify the variable being accessed. At compile-time, each OM is statically-built with a list of the variables they manage with automatically-generated code. For example:

```
#define _ANVIL_TOTAL_MANAGED_VARS 3
#define _ANVIL_TOTAL_ACCESSED_VARS 2
#define _ANVIL_MANAGED_VARS_INITIALISER {0, 0}, {1, 0}, {3, 0}
#define _ANVIL_ACCESSED_VARS_INITIALISER {2, 0}

typedef struct {
 int id;
 unsigned char * data;
} _anvil_var_t;

_anvil_var_t _anvil_managedvars[_ANVIL_TOTAL_MANAGED_VARS] =
   {_ANVIL_MANAGED_VARS_INITIALISER};
_anvil_var_t _anvil_accessedvars[_ANVIL_TOTAL_ACCESSED_VARS] =
   {_ANVIL_ACCESSED_VARS_INITIALISER};
```

The library-generation routines build the `#define` lines at the top of this example to parameterise the general code below. In this example, the OM mapped to the processor that this library is being built for manages shared variables 0, 1 and 3. The library also needs a data structure that describes the shared variables that code on this processors accesses. In this case, only one shared variable (ID 2) is accessed.

The `_anvil_var_t` struct is used for both managed and accessed variables and stores the ID of the variable and a pointer to where the data is stored. For managed variables this is a pointer to the actual storage location of the variable. In an accessed variable, this is a pointer to an equally-sized area of memory that is available for use, into which the shared data may be migrated.

In this implementation, all remotely accessed shared data has an area of local reserved memory into which it may be copied when the thread is working on it. This system is used because it has more predictable real-time performance and the system can guarantee that there will be available memory to perform the copy. For systems which infrequently access a large number of shared data items this can waste a lot of memory, so an alternative system can be implemented which instead dynamically allocates local space upon request using `malloc` and `free`.

In the example above, the pointers are all initialised to 0. This is because they are set by code injected through refactoring as shown in examples in the following section.

Note that both `_anvil_read_sv` and `_anvil_write_sv` alow the operation to begin from a byte offset within the shared variable. This is used to implement pointer-based access to shared variables, as discussed later in section 4.8.4.

4.8.2 Refactoring for remote shared data

The refactoring engine must analyse the input code, determine which statements potentially access shared data, and inject code to invoke the shared memory system correctly. Section 4.4.3 showed how the refactoring engine determines which variables in the program are shared data items. The refactoring engine scans the AST of the input code to locate expressions of the program in which remote shared variables are used as L- or R-values. R-values correspond to reads and L-values correspond to writes. Calls that remotely fetch and update the shared data from its source location are then injected before and after these accesses. The engine assumes that the program is well-behaved and only accesses shared data under mutual exclusion. This is a common assumption in such systems as it is impossible to ensure correct operation if the input program is scheduling-dependent or otherwise contains race conditions. Consider the following code:

```
void main(void) {
   printf("%d\n", x);
   x = x + 1;
}
```

Here, `x` is a shared variable which is read and written by the code in `main`. Anvil transforms the code into the following:

```
1  extern _anvil_a_var_t _anvil_accessedvars[];
2  extern _anvil_var_t _anvil_managedvars[];
3
4  int x; //Local space for remote data
5
6  void main(void) {
7     _anvil_accessedvars[0].data = (unsigned char *) x;
8
9     _anvil_read_sv(0, 0, 0, 4); //id, offset, len, bytes
10    printf("%d\n", x);
11    _anvil_read_sv(0, 0, 0, 4); //id, offset, len, bytes
12    x = x + 1;
13    _anvil_write_sv(0, 0, 0, 4);
14 }
```

Lines 1 and 2 declare external references to the auto-generated data structures of the shared memory system. Lines 4 and 7 show how these structures are populated with suitable pointers. After declaring local space (line 4), the address of the local space is stored into the structure. This system is used because it does not require the refactoring engine to implement a linker and perform manual address space allocation. The refactoring engine defers to the standard compiler chain.

Lines 9, 11 and 13 are the injected read and write calls. In this simple example, no mutexes are present so the system simply places a read call before each read statement and a write call after the write statement. This results in correct functional behaviour but clearly can lead to poor performance if data is frequently accessed.

4.8.3 Associating mutexes with shared data

The reason that Anvil has not been able to ameliorate the two read calls in the previous example is that the input program is poorly behaved and it is not accessing its shared data (variable x) under mutual exclusion. In Anvil, it is possible to associate each item of shared data with a given mutex. This allows the system to only read once when the mutex is locked, to internally cache all writes to that data, and to only update the final value when the associated mutex is released. This is the technique used to good effect by the Rthreads system [65] and is not novel to Anvil. In the example below, the read and write calls are only inserted once.

```
extern _anvil_a_var_t _anvil_accessedvars[];
extern _anvil_var_t _anvil_managedvars[];

int shdata; //Local space for remote data

void main(void) {
   _anvil_accessedvars[0].data = (unsigned char *) shdata;

   pthread_mutex_lock(&mux); //This lock was in the original code
   _anvil_read_sv(0, 0, 0, 4); //id, offset, len, bytes
   printf("%d\n", shdata);
   shdata = shdata + 1;
   _anvil_write_sv(0, 0, 0, 4);
   pthread_mutex_unlock(&mux);
}
```

Data items can be associated with mutexes by applying a `mutex` attribute to the shared data item in the Anvil ADL. For example, the code above has associated mutex `mux` with data item `shdata`. This can be achieved adding the following line to the VP description:

```
shdata^mutex = "mux";
```

If this is done, the shared memory system can rely on the guarantee of mutual exclusion to reduce the number of reads and writes that it performs. When a mutex is locked, read and write caches are maintained for each associated variable. When a read takes place it is only fetched if it is not already in the read cache. Similarly, when a variable is written it is only stored in the write cache. When the mutex is unlocked the write cache is flushed.

Anvil requires the programmer to tune appropriate read and write cache sizes. When the write cache fills up it is flushed. When the read cache fills up, items are discarded and so variables may be fetched multiple times. Although not implemented in Anvil due to time constraints, LRU and Pseudo-LRU are likely to be appropriate replacement policies. A study to determine the best policy is outside the scope of this thesis but is interesting further work.

The sizes of the caches for each item can be described using AnvilADL as follows:

```
shdata^readcache = 10;
shdata^writecache = 10;
```

An alternate approach is to not use caches and instead fetch the entire data item once when this mutex is locked. The entire item will be written back when the mutex is released. This works well for algorithms that read whole arrays of input data, or which require frequent random access to sections of the array, but demonstrates poor performance if only a small subset of the array is needed as a lot of data can be unnecessarily copied. This approach is entirely predictable, which is important in safety-critical systems, and it is amenable to DMA and burst transfers to improve throughput. This behaviour is specified by setting the `fetchall` attribute.

```
shdata^fetchall = True;
```

Anvil requires the programmer to manually associate mutexes with variables because when C is used as the source language this is difficult to automatically derive. If the program is static enough for an analysis engine to determine that the only situations where a given item of shared data is accessed lie between lock and unlock requests for the same mutex then it is safe to assume that the two should be associated. However this is frequently not the case and it is likely that for most programs many associations could not be statically determined.

However, this is a limitation of the source language rather than of CTV. If the language provides the programmer with a way to express this relationship from their source code then such inference is not required. Java, for example, allows the programmer to define a class with instance variables and synchronised methods. All other threads that call any of the class's synchronised methods must first lock an implied mutex, and this mutex can be automatically associated with the instance variables of the class.

4.8.4 Pointers

Recall that the refactoring stage described in section 4.4.3 can identify many forms of pointer-based access to remote variables. In order to implement this, the shared memory system uses offsets. Consider the following code:

```
int shareddata[50];
int *pointer = shareddata;

pointer[4] = pointer[5];
```

After it has been determined that `shareddata` is a shared variable, because `pointer` is initialised it will have been marked as accessing `shareddata` and therefore a target of the shared memory refactoring system.

The first step of the refactoring rewrites the pointer declaration so that it is initialised to zero. This allows it to be used as an offset by multiplying its current value by the size of the element it points to. The example above uses a pointer to an integer, so its address will be multiplied by four. From this point on, the refactoring proceeds exactly as before. Calls to `_anvil_read_sv` and `_anvil_write_sv` are injected before and after the pointer is dereferenced, with the calls ameliorated when the variable is associated with a mutex.

When using this technique, data marshalling must be considered when transferring data between processors of different endianness. Anvil does not yet support data marshalling, but it can be added as an extension to the implementation.

4.8.5 Link scripts

An important part of the shared memory system involves the use of link scripts to manipulate the positioning of shared data items. If a given processor has only one attached bank of addressable memory then the precise location of data items in memory is not important and the standard compiler toolchain can be allowed to position items automatically. However, if an item is to be placed in a specific memory bank (and therefore is tied to a specific address range) then the toolchain must be appropriately instructed. This is done through the creation of link scripts. To illustrate this, consider an architecture described by the following ADL:

```
cpu1 : processor Microblaze;
mem1 : memory BlockRAM;
mem2 : memory BlockRAM;
cpu1^memory = [mem1(0)];
cpu1^extramemory = [mem2(0x20000000)];
```

This ADL describes a processor with two memory banks, the second of which starts at address range 0x20000000. `gcc` and `ld` (the `gcc` linker) will treat the entire address range as homogeneous and may place variables in any location. However, the ADL specifies that a given variable, `shareddata`, should appear in the second memory bank. To do this, Anvil locates `shareddata` in the source code by searching the symbol tables and tags it with an *attribute* called `section`. This is a `gcc`-specific extension that informs the linker that the given declaration should be placed in a certain section. The refactored declaration looks like this:

```
int shareddata __attribute__((section ("mem2shareddata")));
```

The resulting code is `gcc`-specific and cannot be compiled on other compilers that do not support this extension. With this attribute, the linker will be instructed to place `shareddata` in a section named `mem2shareddata` if possible. It therefore must be given a link script which defines this section. The entire link script used is sizable, but the part of interest is as follows:

```
/*...*/
SECTIONS
{
   /*...Other section declarations...*/
   . = 0x20000000;
   mem2shareddata : {}
}
```

This sets the current address to 0x20000000 and declares a section called `mem2shareddata`. When given this link script `ld` will attempt to place variables as specified by the programmer. This script is generated automatically by the Anvil refactoring process.

The problem with this approach is that Anvil must ensure that each processor has a consistent view of the layout of shared memory. Consider the situation where two processors share a block of shared memory to which two variables are mapped. Whilst the two processors may place the shared memory block at different base addresses in their memory map, they must internally place the two variables at the same offset, or else they will be accessing different bytes when referring to the same variable. `ld` is a single memory space linker and does not provide support for doing this automatically, so the easiest way to ensure that this requirement is met is for Anvil to place shared data items manually.

It does this by creating a new section for each item of shared memory and locating those sections to the same offsets in in the memory map of the processor which accesses it. This adds a small amount of complexity to Anvil, but because `ld` is still used to perform the actual linking it is not too onerous. Anvil has to decide on a mapping order for the variables, determine the size of each variable, and place them consecutively.

Determining the size of a shared data item is difficult when it is a dynamic data structure like a linked list. For discussion on how dynamic data objects are distributed throughout the Anvil system see section 4.12.

4.8.6 Other comments on shared memory

Anvil has enough compile-time information to exploit burst-mode and DMA transfers to speed data transmission. The analysis described in this section allows the compiler to determine at which points large data transfers are likely to take place. This can be paired with architectural information provided by the programmer in the VP description to enable the automatic use of on-chip high-bandwidth communication modes, such as burst transfers and DMA. If a DMA engine is present in the system, processors can be instructed to use it automatically. Whilst not yet implemented in Anvil (DMA use currently requires manual intervention), these features could be integrated into the current system.

Also to reiterate, the details of the specific algorithms described in this section are not specified by CTV. As new, more effective shared memory techniques are developed they can be used by CTV implementations like Anvil to increase efficiency or decrease the run-time restrictions placed on the code. Also, whilst CTV's compile-time nature does place some limits on the run-time variability of the programmer's source code, it offers as a tradeoff that it can leverage structural information from the source language in a way that run-time systems cannot. Essentially, CTV has access to the high-level source code whereas a run-time system can only work from the stream of opcodes of individual threads.

4.8.7 Cache coherency

The second situation that the shared memory library must consider is when threads are accessing shared memory and they have caches that must be kept coherent. Cache coherency is a well-studied problem and the results from existing work [55, 45] can be directly implemented in CTV. As coherency is not the focus of this work, Anvil currently only implements a simple algorithm that ensures correctness. A more complex algorithm, such as those mentioned previously, could increase efficiency but are not yet implemented.

The algorithm implemented is as follows:

- Threads read data from a shared variables via interactions with the variable's OM. The implementation requires this to occur even when accessing shared variables that are local to the current processor.
- This imposes low overhead because the OM for the accessed variable will be either mapped to the current processor (so it will require only a function call) or a processor in

```
CPU1 : processor;
CPU2 : processor;
coherentcores : target;
coherentcores^coherent = True;
```

Figure 4.9: AST informing Anvil that coherency is handled between two CPU cores.

the same SMP node (so a single inter-core message). If the variable is associated with a mutex it is not necessary to inform the OM upon every write and read. Accessors need only inform the OM about a read once when the associated mutex is locked, and about writes (if they are made) only once the mutex is released. The provides a form of weak consistency [174], in which the programmer enforces consistency though correct use of synchronisation primitives.

- Each OM maintains a list of the threads that have accessed the variable since the last time it was written.
- When one of the threads writes a value back to the shared data, the caches of all the other threads may now be invalid.
- The next time a thread on the invalid list contacts the OM to begin a read, it replies informing the processor that it must flush its cache lines that cover the shared variable.

Some architectures support cache coherency natively (e.g. the ARM Cortex [83]), so when targeting these architectures Anvil does not need to consider coherency and assumes it will be provided automatically. The programmer can inform Anvil that coherency is automatically handled between a set of processors by adding them to a single cluster target and setting the `coherent` attribute of the cluster target in the AnvilADL. An example description is shown in figure 4.9.

Anvil's simple cache coherency can be illustrated as follows. Consider this code segment:

```
int shareddata[50];

void main(void) {
   printf("%d\n", shareddata[0]);
}
```

As discussed in section 4.8, the above code will be refactored as follows:

```
extern _anvil_a_var_t _anvil_accessedvars[];

int shareddata[50];

void main(void) {
   //Link the shared item to OM
   _anvil_accessedvars[0].data = (unsigned char *) shareddata;

   //Inform the OM that we are reading the shared data
   _anvil_read_sv(0, 0, 0, 4); //id, offset, len, bytes
   printf("%d\n", shareddata);
}
```

The call to `_anvil_read_sv` causes the transmission of a `M_SVREAD` message to the OM of the shared data item. The OM will then respond either with one of three messages:

- `M_SVREPLY` if the requesting thread does not have direct access to the requested data (such as in a non-uniform memory architecture with separate memory spaces).
- `M_SVREPLYSMP` if the processor hosting the requesting thread does have direct access to the shared data item (such as in an SMP architecture) and the data has not changed since it was last fetched by a thread on that processor.
- `M_SVREPLYCLEARCACHE` if the thread does have access (such as in an SMP architecture) and the data has changed since it was last fetched by a thread on that processor so caches should be flushed by the requesting processor.

`M_SVREPLY` is a long, potentially multi-part message which contains the requested data. `M_SVREPLYSMP` and `M_SVREPLYCLEARCACHE` are short acknowledgement messages which inform the requesting thread that it is OK to proceed. However, if an `M_SVREPLYCLEARCACHE` is received then the OM has detected that there is a potential for cache inconsistency so the requesting thread must clear the corresponding data cache lines, assuming caches are used. If the processor is not using cache then `M_SVREPLYSMP` and `M_SVREPLYCLEARCACHE` can be treated as equivalent.

Clearing a cache line varies depending on the target processor. On the Microblaze processor it is not possible to clear individual lines, so the entire cache must be flushed. This is done using the instructions in figure 4.10.

There is potential for an optimisation for SMP-style architectures in which no processor uses cache. In these architectures, the `M_SVREPLY`, `M_SVREPLYSMP`, and `M_SVREPLYCLEARCACHE` become redundant and the entire coherency system can be removed. The system will rely on mutual exclusion to ensure consistency.

4.9 Embedded pthreads library

In order to support pthreads-style constructs, Anvil provides an OM-aware implementation of the base pthreads services that is suitable for distributing over complex embedded architectures. Operations pertaining to mutexes, condition variables, and threads are provided as:

- A set of normal C functions that replace the supported pthreads functions. Rather than making calls to an embedded operating system as the original pthreads functions do, these use the underlying communications layer to send messages to the OMs of the system, requesting the appropriate features. The functions have different arguments to the original pthreads functions, so the refactoring engine is required to refactor the original calls.
- A set of interrupt handling routines that implement the OM-based functionality of mutexes, condition variables and threads. These are installed as interrupt handlers on the host processors of the system's OMs, and they respond to the messages sent by the replacement pthreads functions.

```
#Make space on stack for a temporary and save r12
addi r1, r1, -4
swi r12, r1, 0

#Disable data cache
#(Clear the dcache enable bit in MSR)
mfs r12, rmsr
andi r12, r12, ~128
mts rmsr, r12

#Re-enable data cache
#(Set the dcache enable bit in MSR)
mfs r12, rmsr
ori r12, r12, 128
mts rmsr, r12

#Load r12 and return
lwi r12, r1, 0
rtsd r15, 8
addi r1, r1, 4
```

Figure 4.10: Instructions to clear the Microblaze data cache

Because the library is automatically-generated for each target processor its functionality for both the replacement functions and interrupt handlers is split into two levels, an architecturally-neutral service level and an architecturally-specific architecture level.

Service level: Architecturally-neutral. Implements the functionality of the threads, mutexes, condition variables and shared data items as a set of OMs that are to be distributed amongst the processors of the system. Uses stub functions for all architecture-specific operations, such as ‘send to thread’ or ‘enable interrupts’. The code for this level is shared between all processors.

Architecture level: Architecture-specific. Implements all the stub functions of the service level. This level is processor-specific and is generated at compile-time by the refactoring engine.

4.9.1 Service level functions

The service level implements the following pthreads functions which operate as defined by the POSIX standard [115]:

- pthread_create
- pthread_exit
- pthread_join
- pthread_mutex_lock

- pthread_mutex_trylock
- pthread_mutex_unlock
- pthread_cond_wait
- pthread_cond_signal
- pthread_cond_broadcast

This is not the full set of services defined in the POSIX standard, as a complete implementation is outside the scope of this work. Instead, the above functions were identified as an interesting subset of services to work with. The pthreads functions provide the same functionality that the standard pthreads calls do, but their arguments are slightly altered in the following way:

- `pthread_*` functions: The first argument (normally a pointer to a `pthread_t` instance) becomes the global ID of the thread. Also, the global ID of the processor which hosts the thread's OM is required.
- `pthread_mutex_*` functions: The first argument (normally a pointer to a `pthread_mutex_t` instance) becomes the global ID of the mutex. Also, the global ID of the processor which hosts the mutex's OM is required.
- `pthread_cond_*` functions: The first argument (normally a pointer to a `pthread_cond_t` instance) becomes the global ID of the CV. Also, the global ID of the processor which hosts the CV's OM is required.

The implementation of these functions sends the appropriate request to the OM's processor and waits for the reply. Because the actual act of sending and receiving messages is architecture-specific, the library declares a set of stub functions that are later implemented by the architecture level.

The following Anvil-specific functions are also introduced by the service level:

`void _anvil_wait_until_released(void)`
Wait until this thread is awoken by a `pthread_create` call from elsewhere in the system. Used by the refactoring process when splitting the input program into a set of processor-specific programs.

`int _anvil_read_sv(int svid, int offset, int size, int managercpu)`	
`svid`	The global ID of the shared variable to read
`offset`	Offset in bytes to start reading from
`size`	The number of bytes to read
`managercpu`	The global ID of the processor which hosts the OM of this shared variable
Return value	Number of bytes read.
Read the specified number of bytes from a shared data item. Injected by the refactoring process.	

`int _anvil_write_sv(int svid, int offset, int size, int managercpu)`	
`svid`	The global ID of the shared variable to write
`offset`	Offset in bytes to start writing to
`size`	The number of bytes to write
`managercpu`	The global ID of the processor which hosts the OM of this shared variable
Return value	Number of bytes written.
Write the specified number of bytes back to a shared data item. Injected by the refactoring process.	

`int _anvil_find_thread_by_id(int threadid)`	
`threadid`	The global ID of the thread
Return value	The index of the thread in the OM's internal data structures
Search the threads hosted by the current processor for the given thread ID.	

`void _anvil_send_buffer(int initword, int word, int byteno, int offset, unsigned char *data, int size, int cpuid)`	
`initword`	First word of the buffer
`word, byteno, offset`	Used to implement unaligned transfers
`data`	Data buffer to send
`size`	Number of bytes to send
`cpuid`	Global ID of destination processor
Send a buffer to another processor using the communications layer.	

`void _anvil_wake_thread(int index)`	
`index`	Index of thread in OM's internal data structures. Found with `_anvil_find_thread_by_id`
Called by the Anvil messaging layer to wake the given thread. If the thread was not asleep this function does nothing.	

`void _anvil_set_current_thread_to_wait(void)`
Called by the message handling routines to place the currently executing thread in a sleep state, later to be woken by `_anvil_wake_thread`.

4.9.2 Architecture level functions

The architecture level implements the following functions, which are declared as stubs at the service level:

`void _anvil_send_to_thread(int targetthreadid, int *packet, int len)`	
`targetthreadid`	Global ID of the target thread
`packet`	Data buffer to send
`len`	Number of bytes to send
Send a message to the named thread, uses the underlying channels of the target architecture.	

`void _anvil_send_to_cpu(int cpuid, int *packet, int len)`	
`cpuid`	Global ID of the target processor
`packet`	Data buffer to send
`len`	Number of bytes to send
Send a message to a processor, uses the underlying channels of the target architecture.	

`void _anvil_interrupts_enable(void)`
Enable interrupts on the current processor. Often this must be implemented using inline assembly code.

`void _anvil_interrupts_disable(void)`
Disable interrupts on the current processor.

`void _anvil_sleep(void)`
Place the processor into a low-power state. Not all processors support low-power modes, in which case this function should simply perform no operation and the library will automatically spin-lock as appropriate.

`int _anvil_read_from_interrupt(int vec)`	
`vec`	Interrupt vector to read from
Return value	Index of created message buffer
Read a received message into an internal message buffer from a channel attached to a given interrupt vector.	

`int _anvil_interrupt_ready(int vec)`	
`vec`	Interrupt vector to check
Return value	Zero if interrupt vector is deasserted
Check whether a given interrupt vector has received a message.	

`int _anvil_get_interrupt_vector(void)`	
Return value	Interrupt vector of the currently active interrupt
Called inside an interrupt handler to determine the interrupt vector of the currently active interrupt.	

`void _anvil_acknowledge_interrupt(int vec)`	
`vec`	Interrupt vector to acknowledge
Acknowledge and clear the currently active interrupt. If the processor is using an interrupt controller then driver code is inserted here to manipulate the controller appropriately.	

These functions are assembled from a set of hardware libraries that the Anvil refactoring engine is provided with that include code fragments that describe how to, for example, send a byte over a UART channel or to put an OpenRISC processor into sleep mode. Then, from the ADL provided by the programmer, Anvil can select appropriate drivers and code.

For example, `_anvil_sleep` on an Arm Cortex-M3 processor can be expanded to the following:

```
void _anvil_sleep(void) {
  asm("wfi"); //Execute the WFI instruction
}
```

A Microblaze processor (as it is an FPGA softcore) does not support any form of low-power mode. Consequentially, `_anvil_sleep` on an Microblaze expands to:

```
void _anvil_sleep(void) {
  //NOP
}
```

To support such processors, the Anvil runtime uses a spin lock that calls `_anvil_sleep` each time it spins. On a processor that supports sleep it will only spin once.

4.9.3 Code refactoring for embedded pthreads

Once the pthreads functions have been built, the original pthreads calls must be refactored to use the replacement functions. This requires converting the arguments of the original calls from pthreads' internal representations of threads, mutexes and condition variables into globally-unique IDs that can be used to send messages to the appropriate OM.

In the simple case, the arguments are passed directly so they can simply be fetched from the symbol table. For example:

```
pthread_mutex_t mux;

void main(void) {
   int pthread_mutex_lock(pthread_mutex_t *mut);
}
```

It is trivial to resolve `mux` to uniquely identify the mutex that is being manipulated and convert the pointer into `mux`'s unique identifier. When pointers are used, however, this becomes more difficult. The embedded pthreads library solves this problem by converting the pointer from a local memory address into the ID of the mutex it references directly. As noted in section 4.8, the shared memory system only allows pointers to shared variables if the pointer is initialised to point to the shared variable. The same rule is applied to pointers of the following types:

- `*pthread_t`
- `*pthread_mutex_t`
- `*pthread_cond_t`

The refactoring engine scans the AST for all pointer declarations of the above types, checks which item they are initialised to, and converts the initialisation so that the pointer stores the ID of the referenced item instead of its local memory address.

```
pthread_t mythread;

pthread_t *pointer; //Not allowed!
pthread_t *pointer = &mythread; //OK, refactored to...
pthread_t *pointer = (pthread_t *) 7; //(assuming the global ID of mythread is 7)
```

After refactoring, a call to `pthread_mutex_lock` has the following prototype:

```
int pthread_mutex_lock(short om_cpu_id, short mutex_id);
```

The two arguments are now the ID of the processor which is hosting the mutex's OM, and the ID of the mutex itself. The processor ID is used by the communications layer to statically route messages and make appropriate use of the underlying hardware channels. The mutex ID is read at run-time by the OM.

4.10 Custom hardware drivers

Anvil allows the programmer to use elements of custom hardware from within the programming model of C. This is implemented in two main sections:

1. Code that handles incoming interrupts, thereby allowing hardware elements to trigger the execution of user-provided code on the target processor.
2. Code that manipulates a given hardware element using pre-written driver code as appropriate.

The first section concerns interrupt handling. When building the main Anvil interrupt handler for the current processor, code is inserted to execute a user-provided interrupt service routine (ISR) whenever an interrupt occurs that has the interrupt vector of a known hardware element. The pertinent sections of the main Anvil interrupt handler look like this:

```
void main_anvil_handler()
{
   int vec, currentirq;

   //Get the vector of the current interrupt
   vec = _anvil_get_interrupt_vector();

   //...snip...
   // Main interrupt message handling and OM implementation
   //...snip...

   //Check if it is an accelerator that is interrupting us
   for(currentirq = 0; currentirq < _ANVIL_ACCS; currentirq++)
      if(_anvil_acc_irqs[currentirq].id == vec) _anvil_acc_irqs[currentirq].callback();

   //Acknowledge the interrupt
   _anvil_acknowledge_interrupt(vec);
}
```

The functions `_anvil_get_interrupt_vector` and `_anvil_acknowledge_interrupt` are architecture-specific stub functions that fetch and acknowledge the current interrupt vectors respectively. Because this depends on both the type of the processor and the target architecture, the implementations of these functions are fetched from a bank of hardware libraries. The function is written once for each processor type that Anvil supports and archived for use by the library generation routines. For example, the implementation of `_anvil_acknowledge_interrupt` for a Microblaze processor with an interrupt controller is:

```
void _anvil_acknowledge_interrupt(volatile int *intc, int num)
{
 int i, mask;
 mask = 0x01;
 for(i = 0; i < num; i++) mask = mask << 1;
 (*(intc + 3)) = mask;
}
```

This manipulates an attached interrupt controller that is located at base address `intc`.

The `for` loop of the main Anvil interrupt handler runs once for each interrupt and searches an internal data structure that contains definitions of all hardware accelerators to see if the current interrupt originated from a hardware element that must be handled. This data structure is defined using the following code:

```
#define _ANVIL_ACCS 2
#define _ANVIL_ACCS_INITIALISER {1, 0}, {2, 0}

typedef struct {
   char id;
   void (*fnpointer)(void); //The function to call when this interrupt fires
} _anvil_acc_irq_t;

_anvil_acc_irq_t _anvil_acc_irqs[_ANVIL_ACCS] = {_ANVIL_ACCS_INITIALISER};
```

This defines an array of structs of type `_anvil_acc_irq_t`, one for each attached hardware interrupt. These structs contain two elements, the interrupt vector to which they relate and the address of an ISR function to call when the interrupt fires. As with the shared memory library, the definitions of `_ANVIL_ACCS` and `_ANVIL_ACCS_INITIALISER` are generated by the Anvil refactoring engine at compile-time from the programmer's AnvilADL description.

The second section of this library allows the programmer to interact seamlessly with custom hardware elements. All supported peripherals have a library header file that defines the functions that are supported by that piece of hardware. For example, a UART transceiver's driver includes functions to send a byte, receive a byte and to check the status of the input and output buffers. This code is retrieved from Anvil's external hardware archives and added to the output code as appropriate.

All Anvil driver functions take as their first parameter a C struct that describes the target peripheral. As required by the CTV system model's target layer (section 3.4.4), Anvil must be provided with the base address and interrupt vector for each hardware element. For example, the driver struct for a callback peripheral (a programmable stopwatch that can be set to interrupt the processor after a user-provided amount of time) is:

```
typedef struct
{
   volatile int *addr;
} callback_t;
```

Its driver consists of a single function:

```
void callback_schedule(callback_t callback, int t);
```

To allow the programmer to use this peripheral, when the current processor has access to this peripheral Anvil inserts an instance of the `callback_t` struct with the same name as was provided in the AnvilADL. This allows the programmer to describe in the ADL that a peripheral of a given name exists, and then to use that name directly in their code as the argument to driver functions. The programmer does not have to resort to adding architectural details such as addresses or interrupt vectors to the application. A complete example using the callback hardware follows.

In the following AnvilADL, the programmer has described a single instance of the callback hardware, called `cbhw`. `cbhw` is connected via its only port to the processor `cpu1`. The parameters of this port describe it as being memory-mapped at base address `0x8000A000` and on interrupt vector 0. The programmer has specified that a function called `mycallback` should be fired when the hardware reports back.

```
cpu1 : processor Microblaze;
cbhw : hardware Callback;
cbhw^ports = [cpu1(0x8000A000, "mem", 0, "mycallback")];
```

This hardware can be used transparently by the programmer by referencing `cbhw` and writing `mycallback` as follows.

```
#include <callback.h>

void mycallback(void) {
   //Do something
}

void main(void) {
   callback_schedule(cbhw, 2000); //Callback in 2000 clock cycles
}
```

The refactoring engine amends the programmer's code to include a definition for the driver struct, which it calls `cbhw`. The following line is inserted in the programmer's code:

```
callback_t cbhw = {mycallback};
```

This allows the driver routines to operate correctly. More complex hardware elements will pass more arguments into the driver struct. The address and interrupt vector that the programmer provided in the architecture description are folded into the main Anvil interrupt handler when it is auto-generated. This links the receipt of an interrupt at run-time to the callback routine.

The callback hardware is an example of a user-provided interrupt handler, but more complex hardware drivers can provide their own handlers in exactly the same way.

4.11 Compilation

The output of the Anvil refactoring process is as follows:

- One set of source files per target processor of the original application. Generated by the code splitting phase and refactored to call functions from the Anvil libraries.
- A precompiled object file that implements the shared parts of the embedded pthreads library.
- One set of source files per target processor that implement the architecturally-specific Anvil libraries. Includes the communications library, shared memory system, parts of the embedded pthreads library, and any hardware drivers that are used by the application.

- A custom linker script per processor, generated by the shared memory system.

Due to the fact that Anvil uses unmodified ANSI C, the output code can be compiled with a standard C compiler such as the `gcc` compiler suite. Because the target architecture may contain many different types of processor, an appropriate compiler must be selected for each processor-specific program and library. Currently this process is not automated.

The refactored processor-specific programs and each processor-specific library are all separately compiled to object files using an appropriate compiler. Then, for each target processor its object code is linked against its specific libraries using any custom link scripts generated by the shared memory library. The result of this is a single executable for each target processor. The executables can then be used in a variety of ways depending on the target application. They might be programmed directly into on-chip flash-based storage, stored on disk, transmitted into dynamic RAM during a bootstrap phase, or in the case of FPGAs, merged into the FPGA's configuration bitstream. This is not considered part of the Anvil build process.

4.12 Distributed dynamic memory allocation

In order to aid verification and offline analysis, hard real-time and safety-critical software tends to only make use of fixed-size data structures. Variable-size structures can grow and shrink according to program flow and input data, meaning that it can be difficult to make guarantees about their maximum size or timing properties. Nonetheless, dynamic structures are frequently used in soft real-time and general-purpose software for their flexibility and ease-of-use.

Dynamic structures require the use of a memory management system which maintains a heap structure and provides methods to allocate and deallocate chunks of storage. In C, these are provided by the `malloc` and `free` functions as part of its standard library. Unfortunately, C assumes that it is executing within a single logical memory space so it does not provide a method for distributing `malloc` calls over the class of complex embedded architectures that are targeted by systems like Anvil and CTV.

Consider the following pseudocode:

```
//Shared data item, implemented as a linked list
linked_list shareddata;

Producer thread {
   D := malloc() //Dynamically create a new data item
   Call produce(D)
   Add D to 'shareddata'
   Repeat as necessary
}

Consumer thread {
  D := the next data item from 'shareddata'
  Call consume(D)
  Call free(D) //Dynamically free D
  Repeat as necessary
}
```

This example uses a C-style programming model. Like C, it implicitly assumes that both threads and the shared data item all reside within the same memory space. Therefore, the calls to `malloc` and `free` will refer to the same heap and the code will operate as expected. However, if the consumer thread and the producer thread are mapped to different processors that reside within different memory spaces then the code will fail because `malloc` and `free` will operate on different heaps.

In general, the problems encountered are as follows:

- C's runtime only maintains a single heap, so `malloc` will only allocate memory from that heap. Other memory spaces in the system will remain unused.
- C does not allow the programmer to reason about the processors or memory spaces in the mapped system. As a result, each `malloc` call will simply attempt to allocate memory from the local memory of whichever processor is executing the call. This is acceptable when the processor merely requires temporary local storage, but it is not suitable for use in implementing shared dynamic data structures. Such data structures require memory to be allocated from a specific memory space - the space in which the data structure is located.
- C does not support true abstract data types. These are implemented as a set of library functions that (by convention) are passed a reference to a structure which contains the internal state of the data type. There is no high-level semantic link between these functions and programmers must rely on conventions, such as including all functions of a data type in a single header file or naming them with a common prefix.
- Consequentially, these functions must always execute on the processor of the thread that is manipulating the data, rather than the processor of the shared object's OM (which is closer to the data and therefore more efficient). In a distributed system it is more efficient to pass high-level operations (i.e. 'insert item', 'delete item', 'sort list') to a closely-situated OM that performs the operation rather than attempting to perform the operations remotely.

To solve these problems, Anvil provides the programmer with a way of encapsulating state and the operations that can be performed on that state into a user-defined shared object. The limitations of C mean that the only shared objects that are normally available are those which represent the threads, mutexes and condition variables of the pthreads API. Anvil allows the user to add extra directives to the AnvilADL to define new shared objects, which can then be mapped to the target architecture. User-defined shared objects are implemented solely to improve the mapping of software to the target architecture - they do not change the semantics of the code in any way.

4.12.1 User-defined shared objects (UDSOs)

User-defined shared objects (UDSOs) are give the programmer the ability to define their own limited form of shared objects (from the CTV system model). The advantages of this are as follows:

Support for dynamic structures: As discussed above, C (and therefore Anvil) cannot support dynamic data structures within a complex non-uniform architectures because it assumes a single logical address space. Shared objects have access to a stack and heap (in Anvil this is the stack and heap of their OM), so encapsulating data structures along with the functions that manipulate them inside a shared object solves this problem.

Reduces communication overhead: The shared object is mapped as a single entity, ensuring that its functions remain physically close to the data they manipulate. Threads only need call the high-level interface of the shared object rather than being forced to manipulate the data remotely themselves, thereby reducing communication and coherency requirements.

No change to the semantics of C: UDSOs are a mapping feature only, they do not change the functional semantics of the input program in any way.

Reduces code storage overheads: Functions only need to be compiled and stored once for each UDSO, rather than by every client which accesses the object.

The shared objects (from the CTV system model) that can be defined using UDSOs are limited. Due to the fact that such features cannot be expressed by C, it is not possible to use guards on shared object function calls. Also, in order to ensure that the semantics of C are not affected, the UDSOs does not use its read or write locks. This means that mutual exclusion is not guaranteed by the UDSO. A UDSO has the following features:

- A UDSO comprises a set of C functions and a set of C variables. They are represented as a shared object in the CTV system model.
- The functions make up the interface of this shared object. The variables make up its state.
- Because shared objects have their own memory space, dynamic memory allocation is possible within a UDSO (but confined to the memory space of the UDSO), thereby allowing the efficient implementation and manipulation of dynamic data structures. Extreme care should be taken if pointers are passed out of a UDSO as they will become invalid if the UDSO is migrated.
- Like any other shared object, each UDSO must have exactly one OM.
- The code for shared functions is compiled only for the processor upon which its manager is mapped. Other threads in the system call this OM to execute the protected functions.
- The UDSO's variables are mapped to the address space of its OM.

4.12.2 Describing UDSOs using AnvilADL

When describing UDSOs in C, there are two situations that need to be accounted for:

1. The first situation is when the programmer wishes to merely group a unique set of functions and variables together as a UDSO and map them as a single unit. Each function

is unique to the UDSO being defined. To do this, the programmer needs to be able to state which functions should be exposed as the interface of the UDSO, and which items of shared data should become its internal state.

2. A more complex situation is encountered when functions are shared between UDSOs. For example, a queue datatype may be instantiated many times, but the `get` and `put` functions are shared between all instances. In standard C programming, these `get` and `put` functions will take a pointer to a structure which contains the state of the queue and therefore allows them to work on many queues without duplicating code. For each call of these shared functions, the compiler needs to be able to determine which instance of a UDSO is being affected. Because CTV is a compile-time technique, this must be possible at compile-time.

The first situation can be solved easily using AnvilADL. AnvilADL allows the programmer to declare UDSOs using instances of the `protected` feature category. Functions and shared variables are then added to the UDSO by assigning to the `protected`'s `functions` and `state` attributes respectively. This is shown in the following example:

```
stack_t thestack;

void push_to_stack(int item) {
   //Add an item on to 'thestack'
   ...
}

int pop_from_stack(void) {
   //Remove an item from 'thestack'
   ...
}

void main() {
   push_to_stack(4);
   x = pop_from_stack();
}
```

```
stack : protected;
stack^functions = ["push_to_stack", "pop_from_stack"];
stack^state = ["thestack"];

cpu1 : processor;
mem1 : memory;
cpu1^memory = [mem1];
cpu1^manages = [stack];
```

In this example, the functions `push_to_stack` and `pop_from_stack` affect the variable `thestack` only. Only one stack is maintained by the code and the AnvilADL can wrap up these functions and the stack's state inside a single UDSO called `stack`. As the UDSO is represented as a shared object in the CTV system model it is required to have an OM, which is set to `cpu1` in the above AnvilADL. The code for `push_to_stack` and `pop_from_stack` are compiled for processor `cpu1` and stored in its code memory, which in this example is `mem1`. `thestack` is stored in the memory of `cpu1`, which in this case is `mem1`.

The second situation is more complex and arises from the fact that the code presented in the previous example is slightly atypical. More commonly, the code for `push_to_stack` and `pop_from_stack` would be parameterised to allow them to operate on more than one stack. Consequentially, the compiler needs to be able to determine for each call of these functions which UDSO is being affected, as this will be translated into a procedure call to the affected shared object. To do this, Anvil introduces the concept of an *instance identifier*. An instance identifier is a C variable that is passed as an argument into functions that are shared between UDSOs. It can be used by the compiler to identify at compile-time which UDSO is being called. Consider the following code:

```
void queue_put(queue_t thequeue, int item) {
   //Add 'item' to 'thequeue'
}

int queue_get(queue_t thequeue) {
   //return the top item from the queue
}

void main() {
   queue_t q1, q2; //Declare two queues

   queue_put(&q1, 4); //Affects the first queue (q1)
   queue_put(&q2, 7); //Affects the second queue (q2)
   x = queue_get(&q1); //Affects the first queue (q1)
}
```

This code is more typical of the way that data structures are defined in C. Here, the instances of the `queue_t` datatype (`q1` and `q2`) can be considered instance identifiers because they determine which queue is affected by the otherwise identical calls to `queue_put` and `queue_get`. In an object-oriented language this might be explicitly expressed as:

```
q1.queue_put(4);
q2.queue_put(7);
x = q1.queue_get();
```

AnvilADL allows the programmer to describe the instance identifiers that are present in their program. Then, when the compiler encounters a shared function call it can use the instance identifier to determine which UDSO the call is referring to. Instance identifiers therefore cannot be created dynamically and must be either a global variable or a static function variable.

Instance identifiers and their UDSOs are provided as part of an AnvilADL description. After declaring a `protected` feature, if it requires an instance identifier this can be set by assigning to its `identifier` attribute. Then, when providing the function names that form the UDSO (with the `functions` attribute) the programmer must specify which argument is the instance identifier. The queue example above can be expressed in AnvilADL as follows:

```
q1, q2 : protected;
q1^identifier = "main.q1";
q2^identifier = "main.q2";
q1^functions = ["queue_put"(1), "queue_get"(1)];
q2^functions = q1^functions;
```

This defines two UDSOs that share the queue_put and queue_get functions in their interfaces. The functions assigned to the functions attribute are followed by integers in parentheses which state which argument corresponds to the instance identifier. This value is 1-based, so '2' corresponds to the second argument. The functions of the UDSO may have their instance identifiers in different positions, although it is common C programming convention to keep this the same. Note that in the example above the only state encapsulated within the UDSO is the instance identifier itself. The instance identifier is automatically assumed to be part of the UDSO so it is not necessary to set the state attribute.

Due to the fact that UDSOs are distributed across non-uniform memory architectures, not all functions can be used as shared functions. In both UDSOs with and without instance identifiers the following two restrictions apply:

- Arguments (except the instance identifier) must be passed by value, not by reference. Because the target and source processors may not share any memory, passing by reference is frequently not possible.
- Argument lists must be of fixed length. Variable argument functions are not supported in Anvil, although future work could introduce this if required.

4.12.3 UDSO example

This section describes the complete implementation of an example UDSO, from specification down to implementation code running on the target architecture, and details the refactoring stages required. For clarity, this section only presented an annotated summary of the source code. Appendix E contains the full source.

- In the first stage of refactoring, the AST of the program is searched to find all locations where the functions of UDSOs are called (the functions assigned to the functions attribute of the protected in the AnvilAST). For every call the refactoring engine has to determine which UDSO the call refers to. If the function is not shared between UDSOs (it does not need an instance identifier) then this is trivial. If it is shared, then the refactoring engine ensures that the argument which corresponds to the instance identifier can be statically-traced back to an instance identifier.
- After this analysis, the OM of each UDSO can be augmented to perform the functions assigned to its functions attribute. Each function is assigned a unique ID which is used by the internal messaging protocol. The messages M_UDSOCALL and M_UDSOREPLY are used to implement the actual shared function calls, as described in appendix A. The OM's message handler is extended to handle these messages.
- Because the OM now calls the shared functions, they will be compiled to object code for the host processor of the UDSO's OM. If these functions use dynamic memory allocation (malloc and free) then either an embedded implementation of these functions will be required or the processor must be running a kernel that provides this functionality. An embedded microkernel such as Xilinx's xilkernel [256] is ideal for this purpose. Dynamic memory will be allocated from the target processor's heap only. The programmer is required to ensure that the dynamic memory requirements of the OM does not exceed the memory that is available to it.

- Finally, Anvil's refactoring phase (section 4.3) refactors every function call to send the appropriate message to the UDSO's OM and then wait for any response.

For simplicity the queue presented here stores integers. A more general queue can be built, but care must be taken to ensure that its interface uses pass-by-value. The queue presents the following interface:

```
\\C code
struct{} queue_t; //Implementation omitted
void queue_put(queue_t *q, int item);
int queue_get(queue_t *q);
int queue_empty(queue_t *q);
```

In this example, a single instance of the queue type is manipulated by two threads in the following program:

```
\\C code

#include "queues.h"
#include <pthreads.h>

queue thequeue;
pthread_t producer_thread, consumer_thread;

void *producer_thread_body()
{
   int item;
   while(running)
   {
      item = produce_next_item(); //Produce an item
      queue_put(thequeue, item); //Insert the item into the queue
      //(Omitted) Signal a condition variable to notify the consumer
   }
   pthread_exit();
}

void *consumer_thread_body()
{
   int item;
   while(running)
   {
      //(Omitted) Wait on a condition variable for a signal
      while(!queue_empty(&thequeue)) { //Empty the queue
         item = queue_get(thequeue); //Get an item from the queue
         consume_item(item) //Consume the item
      }
   }
   pthread_exit();
}

void main(void)
{
   //(Omitted) Initialise the queue
   //Set up both threads
   pthread_create(&producer_thread, NULL, producer_thread_body, NULL);
   pthread_create(&consumer_thread, NULL, consumer_thread_body, NULL);
   //(Omitted) Perform extra initialisation, join on the above threads
}
```

For this queue, the struct `queue_t` is used as the instance identifier and internal state of the UDSO. The two threads `producer_thread` and `consumer_thread` are intended to be mapped to different processors. Because the internal implementation of the queue (can be found in appendix E) uses a linked list and dynamic memory, it would not be possible to distribute it over Anvil's shared memory system without the use of a UDSO.

The following ADL is used to define the queue as a UDSO and map it appropriately. The producer thread is mapped to the same processor as the queue (`cpu1`), the consumer thread is mapped to another processor (`cpu2`).

```
# AnvilADL code
cpu1, cpu2 : processor Microblaze;
queue : protected;
queue^identifier = "thequeue";
queue^functions = ["queue_put"(1), "queue_get"(1), "queue_empty"(1)];
cpu1^manages = [queue];
cpu1^threads = ["producer_thread", "main"];
cpu2^threads = ["consumer_thread"];
```

The functions of the shared object are assigned unique IDs so that the `M_UDSOCALL` and `M_UDSOREPLY` messages can refer to them correctly. The following IDs are used:

Function	**ID**
`queue_put`	0
`queue_get`	1
`queue_empty`	2

In the original code the threads call functions to directly manipulate the queue. The refactoring changes this so that instead of calling the function the threads send a `M_UDSOCALL` message to the OM that hosts the target UDSO. The thread then waits for a `M_UDSOREPLY` message to receive the return value of the function. In this example the calls to `queue_empty` and `queue_get` in `consumer_thread_body`, and the call to `queue_put` in `producer_thread_body` must be refactored. The routines to send and receive `M_UDSOCALL` and `M_UDSOREPLY` messages are part of the communications layer, the messages are described in appendix A. After refactoring, `consumer_thread_body` looks like this:

```
\\C code
void consumer_thread_body(void)
{
   int item;
   while(running)
   {
      //(Omitted) Wait on a condition variable for a signal
      while(!_anvil_udso(0, 2)) { //(Refactored) Empty the queue
         item = _anvil_udso(0, 1); //(Refactored) Get an item from the queue
         consume_item(item) //Consume the item
      }
   }
   pthread_exit();
}
```

`_anvil_udso` is a variable argument function defined as part of the communications layer. Its parameters are the ID of the target UDSO, the ID of the shared function, and the arguments of the shared function. Its behaviour is to send a `M_UDSOCALL` message to the manager of the target UDSO and then wait for the `M_UDSOREPLY` message that contains the return value, if any. Note that in this example there is only one UDSO so it has been assigned ID 0.

During the Anvil refactoring stage (section 4.3), the code is split (see section 4.5) as follows:

- **`cpu1`:**
  - Existing `main` function.
  - `producer_thread_body`
  - `queues.h` and its associated `queues.c`.
  - The Anvil communications and pthreads libraries.
  - This processor requires a kernel, as it uses dynamic memory (`malloc` and `free`).
- **`cpu2`:**
  - Autogenerated main function which waits for `consumer_thread` to be created by the `main` thread.
  - `consumer_thread_body`
  - The Anvil communications and pthreads libraries.
  - This processor does not require a kernel and does not include the queue implementation.

Threads mapped to `cpu2` are no longer dependent on the implementation of the queue datatype because they send UDSO messages instead of attempting to directly manipulate the datatype. The queue is localised entirely within the memory space of `cpu1` and its implementation only has to be compiled for a single processor. Note that the use of the UDSO in this example has not changed the semantics of the code, it has instead added extra mapping information that allows the refactoring engine to better distribute dynamic data structures to a non-uniform memory architecture.

4.13 Implementing migration

In order to implement the logical layer, Anvil supports the migration of data and OMs throughout the system. The migration of threads and Anvil POs are supported by the system model but not yet implemented as to do so is outside the scope of this thesis. Section 4.7.2 discussed how the communication layer accounts for migration to ensure that the rest of the system can continue to communicate with migrating objects. This section explains how migration is supported by the OM model and how it is implemented.

Migration is always performed by the manager of the migrating object. The OM will prepare the object for migration, send it to the target processor, restart it, and inform the rest of the system that the migration has taken place. In some situations, the OM will also need to migrate itself at this time. This section will detail how the various items of the system are migrated and the limitations that are encountered when doing so.

4.13.1 Migrating OMs

The first migration type that must be considered is the migration of an OM. In Anvil, OMs migration is heterogeneous, meaning that they can migrate between all processor types.

As detailed in section 4.2.8, AnvilADL marks OMs as migratable by assigning them to the `contains` attribute of a cluster c. Anvil will then automatically select a single processor from each cluster target to which c is mapped to form a set of the OM's *target processors*. This automatic selection can be overridden by also assigning to the manager's `executes_on` attribute.

The model, therefore, is that OMs may migrate between a set of target processors in a way that is consistent with the cluster-based migration model. However, in order to maintain implementation efficiency this is not the way that it is actually implemented. It is not necessary to perform true code migration to move the OM. As highlighted later in section 4.13.3 this is very difficult in an unconstrained heterogeneous environment. Instead, OMs are placed on each target processor and control over the managed items (threads, shared objects, shared data items) is migrated between the OMs, rather than the OMs themselves.

When an OM migrates, it passes the state of its managed objects to the target instance and updates the communications layer so that the rest of the system sends their requests to the new manager. OM migration is implemented using the following algorithm:

- All managed items must be shut down.
  - For variables and shared objects (mutexes, CVs, Anvil POs) the migrating OM completes any requests currently underway and stops responding to further requests. Any requests that arrive from this point on are queued.
  - If a request takes too long to complete it may be aborted (for example, if a thread is requesting a lock that another thread has claimed and then forgotten to unlock). The current implementation assumes that all requests will complete in finite time.
  - For threads, the thread is removed from that processor's run queue. If the thread is in a critical region the OM must wait until the thread leaves the region and can be unscheduled.

- The OM transfers its current state to the target OM instance. The state of an OM is defined in section 3.4.6 but can be summarised as follows:
  - For all OMs, the current request queue.
  - For mutex OMs, the current owner of the muetex and the queue of all waiting threads.
  - For CVs OMs, the queue of pthreads currently waiting on the condition.
  - Thread helper OMs, the current state of the thread and the queue of all threads waiting to join it.

  This state transfer is easy to implement, because the OM is not currently executing. It transfers its state using messages defined in the OM message protocol which is detailed in appendix A.
- The OM updates all existing processors with its new location. This is detailed in section 4.7.2.
- The OM messages the target OM instance to begin executing.
- All managed items are then restarted by the new instance and the system resumes. All queued requests can be answered.

It is possible when using this algorithm for the source OM to have transferred its request queue and completed migration, but due to network delays a request still arrives at the source OM. This is not a problem because the source OM is still operational, even though it has transferred its management duties to another instance. In this case the request is forwarded on to the migration target.

Note that if the source and target processors use different endian representations then the migrating OM must convert the transferred bytes appropriately. To implement this, an OM can query the endianness of the target processor with the `M_GETENDIANNESS` message.

Migration of a managed object is performed by the transmission of a `M_MIGRATE` message from the source to the target. This is a multi-part message which contains the ID and state of the migrating object. This message is defined in appendix A.

If a shared variable is being migrated between memory spaces that can both be accessed by the source processor, it is not necessary to involve the target OM throughout the data transfer. Instead, the source processor transfers the shared variable itself (which could involve the use of DMA), and then sends a `M_MIGRATE` with an empty state. The receiving OM understands that the data has been moved accordingly. This can be done because in Anvil space for shared variables is statically allocated at compile-time. A system which used dynamic memory allocation would have to include a negotiation phase in which the target OM replies to the source with a suitable address to copy the data to.

4.13.2 Migrating shared data

The migration of shared data elements is an extension of OM migration. As detailed previously, the OM of a shared variable must be assigned to a processor that has access to the memory that contains the shared variable. This affects the migration behaviour. There are two situations encountered when migrating shared data:

- The variable migrates to a memory space that is still accessible to the processor hosting its OM.
- The variable migrates to a memory space that is inaccessible to its OM.

In the first situation, the OM has to perform a bulk memory copy from the old location to the new location. It must ensure that no threads are currently accessing the data by requesting the lock of the shared variable's associated mutex (section 4.8.3). Anvil does not support migrating shared variables that do not have an associated mutex. The associated mutex is released once the data copy is complete.

The second situation requires that the OM migrate itself to a suitable location, along with the value of the shared variable. The OM migration proceeds exactly as detailed above in section 4.13.1. When the OM transfers its state to the target processor it copies across the shared data. As with the previous situation, the migration ensures that it holds the shared variable's associated mutex before proceeding. The current implementation of Anvil pre-allocates space for the shared data to migrate into to provide greater reliability. Each OM contains duplicate definitions of the shared variable and the data is copied between these using a raw pointer. A more flexible but less predictable approach could instead attempt to allocate space dynamically just before migration, failing if there is not enough spare memory.

4.13.3 Migrating threads

Migrating a thread is much more challenging than migrating the OMs or shared data items of the system. These items are both passive objects with a fixed functionality so it is easy to implement migration which takes place only when they are in a known state. Also, their internal state is well-defined by the Anvil specification so it is clear what data must be migrated. Threads, on the other hand, are active objects that are defined by the programmer with an unknown state and memory usage. Making this even more challenging is that in an embedded context the source and target processors might be of different architectures. This requires the implementation of a *heterogeneous process migration* framework.

The process of transferring a running thread from one processor to another requires the following steps:

- First the thread must be stopped so that its execution does not interfere with the migration. It is not possible to halt execution at any arbitrary point because the in-memory source and target executables may be different, meaning the current program counter value will have no meaning in the target system.
- This problem is solved using a checkpointing system in which checkpoints are added at frequent points in either the source code or opcodes of the thread. Threads may only be migrated once they reach a checkpoint. The checkpoints are identical in the source and target executables, thereby allowing a program counter correspondence across different ISAs and executable formats.

- The execution context of the thread must be extracted. This consists of the state of the processor's registers and the thread's stack. This can be obtained from the kernel on the source processor, normally named a process control block (PCB).
- The context must be translated so that it fits the architecture of the target processor. This requires converting register contents appropriately and marshalling between different data representations. This translation is not always possible so it imposes restrictions on the migrations that are supported.
- The translated context is transferred to the target processor and a new thread created from it. The thread is now executing on the target processor and kernel.

There are a number of existing systems which implement heterogeneous process migration systems. Tui [205], Dome [8], Gantel et. al. [85], and Ramkumar et. al. [185] all use a version of the algorithm above to migrate tasks between processors with differing instruction sets and architectures. These approaches differ by the manner and frequency at which checkpoints are added, displaying different run-time properties accordingly.

Systems based on Java or other interpreted languages have the advantage that threads are compiled to bytecodes so portability is guaranteed. Examples of such systems are JESSICA2 [269], the work by Sakamoto et. al. [193] and the work by Truyen et. al [225].

Anvil does not currently implement a thread migration mechanism because the required amount of implementation effort places it outside the scope of this thesis. It is further work to take a system such as Tui and implement it as part of Anvil. This section instead argues that CTV provides support to a thread migration system and does not interfere with their implementation.

The CTV system model of Anvil assists thread migration systems in the following ways:

- As detailed in sections 4.7 and 4.13, the communications layer of Anvil supports migration by ensuring that the rest of the system can transparently communicate with migrating items. Messages are always routed to the current location of the object without requiring any extra effort from the sender. Existing thread migration systems tend to be implemented either on SMP-style architectures or over large-scale networked architectures, and so rely on network routing to provide this. CTV and Anvil supports routing over complex non-standard architectures.
- Similar to the previous point, the shared memory system of Anvil supports thread migration by ensuring that the migrated thread can still access shared data items.
- CTV promotes a programming model in which dynamic behaviour is explicitly enumerated at compile-time. The programmer states exactly which processors a given thread may migrate over. As a result, if the thread migration system cannot support a given type of processor it can still be used in the static parts of the system. This is in contrast to existing systems, such as Tui, which restrict the entire system to migration-amenable processors.

Due to the fact that Anvil deals with inter-thread communication, shared memory and hardware access, it places only one restriction on the implementation of a thread migration system – that it is unsafe to migrate a thread whilst it is executing within the Anvil libraries. If the thread is currently updating the communications layer or interacting with an OM it must complete that operation before it is migrated. When using the checkpointing system described above that is used by many heterogeneous thread migration systems, checkpoints should only be placed in the user's code, and not in the code that is generated by the Anvil refactoring engine. If this is done then Anvil will not interfere with existing thread migration systems.

4.13.4 Causes of migration

The previous sections have detailed the way that a CTV-based system implements migration of threads and data. The implementation uses the clustering information provided by the CTV system model, alongside the OM model, to provide migration services in a scalable way that does not introduce unnecessary bottlenecks into the system. However, the implementation presented so far assumes the presence of a higher-level algorithm that decides when migration should take place, which items should migrate, and to where.

The use of migration to improve the performance of a system is well-studied. Migration has been used to improve systems according to a wide range of metrics such as throughput in a distributed environment [119, 44], communication minimisation [222], fault tolerance [37, 2] or power scaling [166].

Anvil does not yet implement any form of automatic migration and assumes the presence of a user library to implement a suitable migration policy. There are two reasons for this. Firstly, Anvil should not re-implement the existing work from sources such as those described in the previous paragraph. Secondly, because of the wide range of reasons for implementing migration in an embedded system it would not be possible to create a 'one size fits all' policy that works well for all target systems. A migration policy to assist the shutdown of processors for power saving is unlikely to work well in a system which uses migration for application flexibility and maximising throughput. Future work will extend Anvil to include a wide range of migration algorithms and thereby avoid the need for the programmer to work at this level, but this is outside the scope of this thesis.

Currently, migration is triggered by calling the following functions:

- _anvil_migrate_om(int sourcecpu, int omid, int targetcpu);
- _anvil_migrate_thread(int sourcecpu, int threadid, int targetcpu);
- _anvil_migrate_sv(int sourcecpu, int svid, int targetaddress);

These functions initiate the transmission of a `M_MIGRATE` message to the appropriate OM. The OM then begins the migration process as detailed in the previous sections.

4.14 Traceability

The refactoring stage adds a layer of transformation to the programmer's source code, and so CTV's impact on traceability must be considered. Clearly, the code that is executing on the target platform is not the code that the programmer wrote, and this can cause problems for interactive debuggers. However, as Anvil's aim is to take a single input program and split it into multiple programs that are distributed over a multi-core architecture, this impact is unavoidable.

Due to the fact that Anvil performs all of its refactoring at compile-time in an entirely predictable manner, it is still possible to refer any given line of output code back to the source code line that it came from, as long as the VP mappings are known. Anvil's refactoring stages can be thought of as similar to standard compiler optimisations. They can make drastic changes to the ordering and function of the actual final instructions, but because these are predictable it is possible to carry standard debugging information into the object file and not lose visibility.

The following features must be accounted for in any debugging environment:

- The most significant changes made during refactoring are by the code splitting phase, which splits the single input program into a set of output programs, one for each target processor. Each output program is a subset of the input program (as determined by code reachability analysis). Consequentially, the vast majority of these output programs remain identical and easily traceable back to the input code, although the debugging environment should account for the fact that code is frequently duplicated over multiple processors, especially in the case of common libraries. All output programs, apart from the one derived from the initial `main` function, will have a custom entry point which waits until other threads in the system wake it with a `pthread_create` call.
- Extra code is injected into the output programs to implement the distributed shared memory system (section 4.8). This code consists of shared variable read and write calls, and chance management functions. These changes are only minor and can easily be traced in exactly the same way as if the programmer had added them manually. However, such refactoring can affect timing and memory use so in areas where such features are important the code analysis tools must operate on the refactored version of the program.
- pthread functions are refactored from the POSIX standard form into the Anvil form. This means that they have different arguments, but their traceability remains otherwise the same. This could affect data-flow analysis and data-dependency graphs.
- The communications and embedded pthreads libraries are dynamically-generated and added to the output source code. This means that during debugging, the dynamic nature of the libraries must be considered and the programmer must ensure that the result of the refactoring stage is taken into account by their debugging environment. However, whilst the libraries are dynamic, they only change in small, well-defined areas. The entire of the service level (see section 4.9) is static and does not change. At the architecture level only function implementations change, functions are not created or removed. This means that call chains and traces are unaffected.
- Interrupt handlers are added to implement sections of the communication library. If an embedded kernel is being used, this interrupt handler is attached to the kernel's existing interrupt handling scheme. The code for the handler is static, although unused sections are optimised out in a similar way to standard dead code removal that is implemented by many compilers and linkers. When debugging, the programmer should be aware that such a handler has been added.
- If an object manager is mapped to a target processor, the code for that processor will be augmented with the OM's code. This code is largely static, but as with the interrupt

handler for the communications layer, sections that are not required will be removed. The main difference implied by the presence of an OM is the extra memory required for OM data structures (request queues, mutex and condition variable state, etc.).

- The use of hardware methods results in some functions of the source language calling driver code to manipulate external devices. The code for these drivers comes from an external support library and should be included in the debugging environment.

One problem caused by the presence of the refactoring stage is that if the compiler reports an error during development it will report the error location according to its position in the refactored code rather than the original source. Due to the fact that all of the refactoring steps mentioned above are entirely predictable, it is possible to filter the reported error locations before presenting them to the user. This requires either modifying the compiler or piping its output through a VP-aware filter. However, for many smaller edits it is sufficient to use line annotations.

Line annotations exist in many toolchains to allow the input code to manipulate the compiler's internal line counting and thereby produce better error messages. CTV can use these facilities so that inserted code does not affect line numbering of any errors in the programmer's original source code. The following example shows how this is done with `gcc`. The original code, shown below, reads from a shared variable and so will have extra lines injected into it by the refactoring engine.

```
1  void main(void)
2  {
3     int x;
4
5     x = shareddata; //Shared variable read
6
7     printf(%d\n, x); //Syntax error
8  }
```

As can be seen, there is a syntax error on line 7 (the programmer has forgotten to enclose the format string passed to `printf` in quotation marks). If this code is passed to `gcc` it will report line 7 as the source of the error. However, the refactoring phase adds a number of lines to the program to ensure that the remote variable access operates as expected, resulting in the following:

```
1  extern _anvil_a_var_t _anvil_accessedvars[];
2  int shareddata; //Local space for remote data
3  void main(void)
4  {
5     int x;
6
7     _anvil_read_sv(0, 0, 0, 1); //id, offset, len, bytes
8     x = shareddata; //Shared variable read
9
10    printf(%d\n, x); //Syntax error
11 }
```

The compiler now reports that the error is on line 10, which is unhelpful to the programmer. To avoid this, the preprocessor directive `#line` can be used as follows.

```
extern _anvil_a_var_t _anvil_accessedvars[];
int shareddata; //Local space for remote data
#line 1
void main(void)
{
   int x;

   _anvil_read_sv(0, 0, 0, 1); //id, offset, len, bytes
   #line 5
   x = shareddata; //Shared variable read

   printf(%d\n, x); //Syntax error
}
```

Error messages will now correctly correspond to the programmer's original source code.

4.15 Restrictions on input code

In summary, the following restrictions are placed on the input code that Anvil can accept:

- The language must conform to the ANSI C standard. `gcc` attributes and other extensions are also supported for compatibility with standard libraries.
- Instances of `pthread_t`, `pthread_mutex_t` and `pthread_cond_t` must be declared either global or static. They may be placed in arrays, but the array size must be compile-time static.
- Dynamically allocated memory (using `malloc` and `free`) cannot be shared between threads, unless enclosed in an Anvil PO (section 4.12).
- Pointers used to access shared data must be initialised to point at the shared data item.
- Pointers used to refer to threads, mutexes or condition variables must be initialised to point at the item.
- Pointers, once initialised to an object, can not be assigned to point to a different object.
- For every `pthead_create` call, the first argument must be a direct reference to a pthread_t instance. Multiple `pthread_create` calls can be used to describe threads with multiple thread bodies.
- Non-constant function pointers cannot be used in Anvil.

In addition to this the programmer should be aware that pointers passed between threads are not guaranteed to work. If at compile-time the threads are mapped to processors with different memory maps then problems can arise. This is not checked. Programmers are instead encouraged to pass an ID number or similar which can be resolved at the target thread in order to refer to shared objects.

4.16 Conclusion

This chapter has introduced Anvil, an implementation of CTV that supports the clustering system model described in the previous chapter. Anvil's input language is ANSI C (with use of the POSIX pthreads library assumed to provide concurrency and coordination) and it distributes input code over FPGA-based, heterogenous, non-uniform memory architecture systems. This chapter also defines a simple architecture description language called AnvilADL (in section 4.2) which is used to describe the target architecture of a system and the way in which the input program should be mapped to it.

Due to the compile-time nature of CTV, Anvil is based around a source-to-source refactoring engine, described in section 4.3. The user's input code is split from a single input program into a set of subprograms, one for each target processor of the system. Then, a set of processor-specific libraries are constructed that implement the communications layer, a distributed shared memory system, an implementation of pthreads for embedded systems, and include all required hardware drivers. The processor-specific subprograms are then refactored to use the generated libraries. The result of the refactoring process is that the user's architecturally-neutral code is mapped according to their AnvilADL description and will execute correctly.

A requirement of the CTV system model is that the application's structure is fixed at compile-time. Accordingly, Anvil places restrictions on the run-time variability of the input code, which are summarised in section 4.15. These restrictions allow the implementation of the Anvil libraries to be only as large as is required by the current application. Only features that are actually used by threads on a given processor need to be included in its support libraries. For example, if a processor does not migrate then it does not need to store code to perform a migration, even if other processors in the system host migrating objects. The overheads associated with a feature are limited to the processors that use it.

Anvil provides support for run-time migration, with its communication library ensuring that migrating objects can always be communicated with. Anvil does not yet implement migration policies, or techniques for heterogeneous thread migration as these are studied extensively in existing work.

Chapter 5

Evaluation

Chapter 3 introduced Compile-Time Virtualisation (CTV), a technique that can be used to facilitate the mapping of embedded software to complex, non-standard architectures. This was exemplified in chapter 4 which introduced Anvil, an implementation of CTV based on the C programming language. Anvil implements CTV's system model and virtualisation techniques to allow applications written in standard C to be mapped to heterogeneous FPGA-based architectures.

In this section, CTV and Anvil are evaluated against their stated goals. Section 5.1 examines the expressibility that CTV and Anvil affords the programmer when targeting code at non-standard architectures. Section 5.2 evaluates the overheads in a CTV-based system and in Anvil specifically, and section 5.3 evaluates the clustering model and how it can be useful for targeting current and future embedded architectures.

5.1 Targeting architectures with CTV and Anvil

This section evaluates whether the CTV system model, and the CTV implementation Anvil, are sufficiently expressive to be used for the development of software for current embedded architectures. The model's descriptive capabilities are explored in terms of both the representation of the target architecture and the representation of software to hardware mappings.

To do this, a range of architectures are shown along with corresponding AnvilADL descriptions and notes about software mapping. The architectures of real systems are described to show applicability to real-world problems, and invented architectures are used to argue about future systems and specific situations. Where appropriate, example applications are mapped to these architectures.

By using the refactoring techniques detailed in section 4.3, the Anvil refactoring engine can target all the architectures shown. Small code examples are shown to demonstrate the way in which software can be distributed over these architectures. Note that because this section is evaluating the expressibility of CTV and Anvil, the performance of the final mapped system is not considered here. Section 5.2 presents an in-depth analysis of overheads and efficiency.

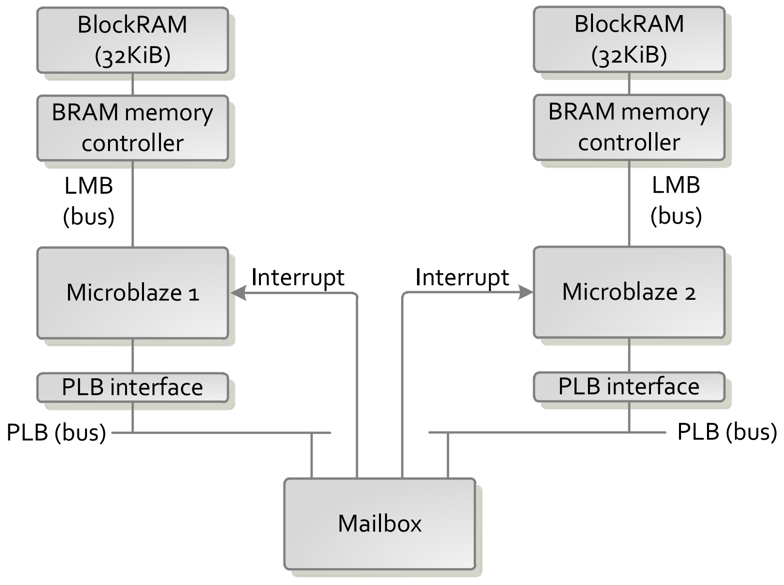

Figure 5.1: Simple dual-core architecture as it is implemented on an FPGA.

The CTV system model has already been shown to be suitable for describing the programming models of existing embedded development languages. Section 3.4.5 showed how programs written in C, Ada, and Java can all be described by the model, and chapter 4 contains many examples of how C code is represented by AnvilADL.

5.1.1 Capability-centric descriptions

The CTV system model does not fully describe all details of the target architecture. Implementation details that are transparent to the programmer are omitted because their transparency means that they do not have an effect on the way in which software is mapped. In essence, the CTV system model (and therefore AnvilADL) is only concerned with the *capabilities* of the architecture from a processor-centric viewpoint [1].

The capability-centric view of CTV is illustrated by the architecture shown in figure 5.1. This architecture is a simple FPGA-based dual-core system comprised of two Xilinx Microblaze processors [258], each with their own separate memory and connected via a mailbox communication channel [262]. Note that implementation requirements of the processors require the inclusion of details such as peripheral buses, memory buses, interrupt lines and memory controllers.

Rather than express all these details, figure 5.2 shows that way that CTV models the architecture. The differences are:

[1]Note that the omitted information does affect the non-functional properties of the architecture, and so affects mapping decisions. Because Anvil does not perform automatic mapping it does not need this information, but an automatic system would require extra attributes in the AnvilADL.

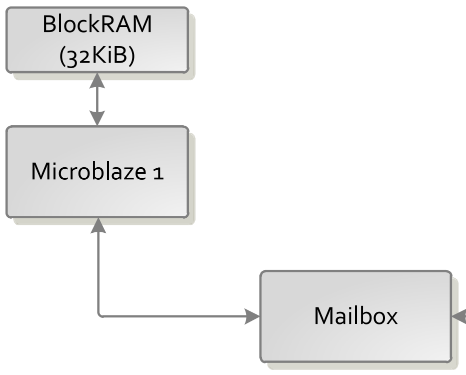

Figure 5.2: The simple dual-core architecture of figure 5.1 as it is modelled.

- Memory is expressed simply as the abstract concept of a 'memory space' rather than including details of the memory controller and how it connects to the actual memory and to the processor. Caches are expressed as a property of the memory → processor connection, rather than by including the cache, cache controller, and associated buses.

- Interprocessor communications are expressed as abstract 'communication channels'. This carries all required semantic information without detailing channel hardware and the way it is connected to the target processors.

These two changes have the advantage that they provide extra semantic information. It is not possible to tell from the architecture of figure 5.1 that the mailbox hardware is used for interprocessor communications, without detailed knowledge of the PLB interfaces and the behaviour of the mailbox hardware itself. However, in the simplified architecture of figure 5.2, this is all replaced with a 'channel' object that is defined to provide interprocessor communications. Similarly for memory, without detailed knowledge of the memory controllers the system cannot automatically determine their purpose. However once simplified to a 'memory space' object their use is clearly defined.

The AnvilADL description of this architecture is as follows:

```
cpu1, cpu2 : processor Microblaze;
mem1, mem2 : memory BlockRAM;
cpu1^memory = [mem1(0x0, 0x7FFF)];
cpu2^memory = [mem2(0x0, 0x7FFF)];
mem1^size = 8192;
mem2^size = 8192;
mem1^width = 32;
mem2^width = 32;

mbox : channel Mailbox;
mbox^endpoints = [cpu1(0x80000000, 0), cpu2(0x80000000, 0)];
```

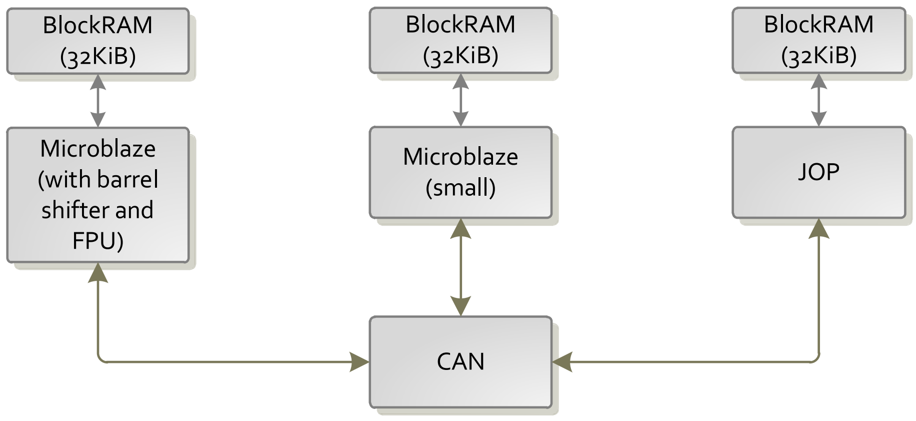

Figure 5.3: An example of a heterogeneous embedded architecture.

FPU	Barrel shifter	Area (LUTs)	Max frequency (MHz)
None	None	1421	92.621
None	Yes	1647	92.615
Basic	Yes	2660	89.946
Extended	Yes	3134	89.594

Figure 5.4: Different configurations of Microblaze soft processor on the Xilinx XC3S500e FPGA. [260]

5.1.2 Heterogeneous architectures

A common distinction between embedded architectures and general-purpose or high-performance architectures is that embedded systems tend to contain heterogeneous processing elements with greatly differing capabilities. Consider the architecture in figure 5.3. It contains two Microblaze processors that are broadly similar but configured differently so that they contain different functional units. The larger Microblaze contains a barrel shifter unit and floating point unit. This enables it to perform many more operations, but increases its size accordingly, as shown in figure 5.4.

The third processor is a different processor type, the JOP Java processor [196]. JOP (Java Optimised Processor) is a hardware-based implementation of the Java Virtual Machine that allows direct execution of Java bytecodes. JOP has much more predictable execution time bounds than a Microblaze executing the same bytecodes through a software-based JVM, making it suitable for real-time systems.

The memories used in the architecture are the same as in the previous example and communications are provided by an on-chip CAN bus using a Xilinx CAN interface [259]. The hardware is described as follows:

```
bigmb, smallmb : processor Microblaze;

bigmb^barrelshifter = True;
bigmb^fpu = True;
```

```
jop : processor JOP;

mem1, mem2, memjop : memory BlockRAM;
bigmb^memory = [mem1(0x0, 0x7FFF)];
smallmb^memory = [mem2(0x0, 0x7FFF)];
jop^memory = [memjop(0x0, 0x7FFF)];

mem1^size = 8192;
mem2^size = 8192;
mem3^size = 8192;
mem1^width = 32;
mem2^width = 32;
mem3^width = 32;

can : channel CAN;
can^endpoints = [bigmb(0x80000000, 0, 0),
   smallmb(0x80000000, 0, 1), jop(0x80000000, 0, 2)];
```

Due to the large amount of heterogeneity in this system, the placement of code throughout the architecture is particularly important. Consider the following program fragment:

```
pthread_t calcthread;

void vector_mult(float *data, int len, float mult) {
   int i;

   for(i = 0; i < len; i++)
      data[i] = data[i] * mult;
}

void *calculation_thread() {
   float data[50];
   float multfactor;

   ...prepare data...

   vector_mult(data, 50, multfactor);
}

void main(void) {
   ...
   pthread_create(&calcthread, 0, calculation_thread, 0);
   ...
}
```

This code makes use of floating point arithmetic to perform a vector multiplication, so if implemented on a processor without a floating point unit the compiler must emulate the floating point operations with integer arithmetic. Mapping the code fragment is performed simply by adding one of the following lines to the AnvilADL:

Processor	Execution time (cycles)	
	1 iteration	100 iterations
smallmb	15,034	1,503,734
bigmb	631	63,426
jop	51,002	114,428

Figure 5.5: The performance of the same code fragment on different target processors.

```
bigmb^threads = ["calcthread"];
//OR
smallmb^threads = ["calcthread"];
```

Note that Anvil's source language is C rather than Java so it cannot normally use the JOP processor. For illustrative purposes, mapping to JOP was performed manually by wrapping the `calculation_thread` function up as a Java program and compiling it using the JOP toolchain. No other refactoring is needed in this example. The compiler should be told to use the FPU in the Microblaze that supports it by adding the `-mhard-float` compiler switch.

Figure 5.5 shows the effect that different mappings can have on system performance. In each case exactly the same results are obtained, but the amount of time taken to perform the vector multiplication varied considerably. Clearly, when mapped to `bigmb` the hardware floating point unit can be used and the code executes quickly. `smallmb`'s software emulation is over 23 times slower. JOP's result is different again. It takes much longer to execute, but this is because the measured time taken includes the overhead of setting up the processor's microcode stores. As this test shows a short program that is only executed once, this overhead dominates. Longer programs increase JOP's efficiency relative to the other two processors.

This example shows two important points about CTV and Anvil. The first is that CTV's main contributions are in providing seamless architectural mapping and exposing a high-level hardware interface to allow efficient exploitation. The architecture is exposed in a high-level manner that allows quick and easy exploration of the solution space. However, the second point is that CTV still relies on the presence of a suitable source programming model and compiler to make use of the underlying hardware. Because Anvil's source language is C and not Java, it cannot make effective use of the JOP processor and so manual intervention is required. Section 3.4.8 showed how the system model can be applied to Java, so this is a limitation of Anvil rather than CTV.

5.1.3 Non-uniform memory

The architecture in figure 5.6 shows a non-uniform memory architecture. The architecture contains two processors that each have their own block of memory and have access to a single block of shared memory. The shared memory is a block of off-chip DDR2 memory controlled by a multi-port DDR2 memory controller (not shown). In the implementation this controller is the Xilinx Multi-port Memory Controller (MPMC) [265]. The architecture is represented in AnvilADL as follows:

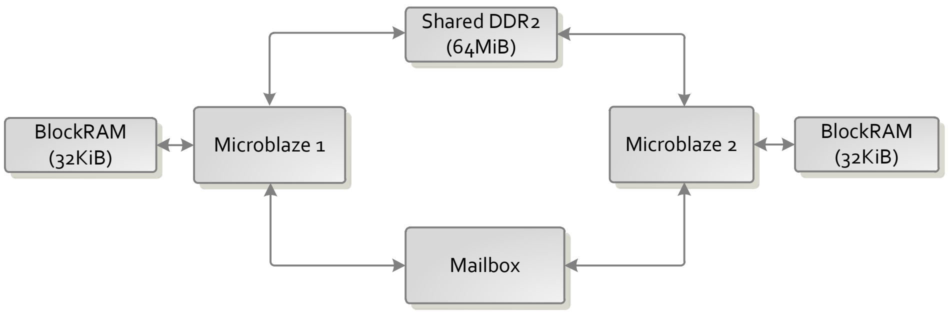

Figure 5.6: An example architecture with a non-uniform memory architecture.

Address	**Contents**
`0x00000000 - 0x00007FFF`	BlockRAM
`0x00008000 - 0x0FFFFFFF`	*Unassigned*
`0x10000000 - 0x10FFFFFF`	Shared DDR2 memory

Figure 5.7: The address map of both processors in the NUMA example.

```
cpu1, cpu2 : processor Microblaze;
mem1, mem2 : memory BlockRAM;
sharedmem : memory MPMC_DDR2;

cpu1^memory = [mem1(0x0, 0x7FFF), sharedmem(0x10000000, 0x10FFFFFF)];
cpu2^memory = [mem2(0x0, 0x7FFF), sharedmem(0x10000000, 0x10FFFFFF)];

mem1^size = 8192;
mem2^size = 8192;
mem1^width = 32;
mem2^width = 32;

sharedmem^size = 16777216;
sharedmem^width = 32;
//64 MiB

can : channel Mailbox;
mbox^endpoints = [cpu1(0x80000000, 0), cpu2(0x80000000, 0)];
```

The resulting address map of both processors is shown in figure 5.7.

As with the previous example, this sort of architectural feature highlights the importance of code mapping. Anvil's communications layer means that shared data can be placed in any memory space and the application will still execute correctly, but the difference in performance is significant. Figure 5.8 shows the effect that memory mapping decisions have on a set of benchmark programs executing on CPU1. If the program's primary input data is local (assigned to memory `mem1`) then the execution is in some cases two orders of magnitude faster than if the data is mapped to a remote memory location (`mem2` in this architecture).

Benchmark	Execution time (cycles) Input data stored locally	Execution time (cycles) Input data stored remotely
fdct	9908	216350
quicksort	26157	320142
binarysearch	1114	9502
bsort	78044	1075473
sobel	574097	21811154

Figure 5.8: The effect of mapping data items to different memory spaces in the architecture of figure 5.6. Adding caches would reduce the discrepancy for programs that frequently access the same data (sobel) but have little effect on programs that do not (binarysearch).

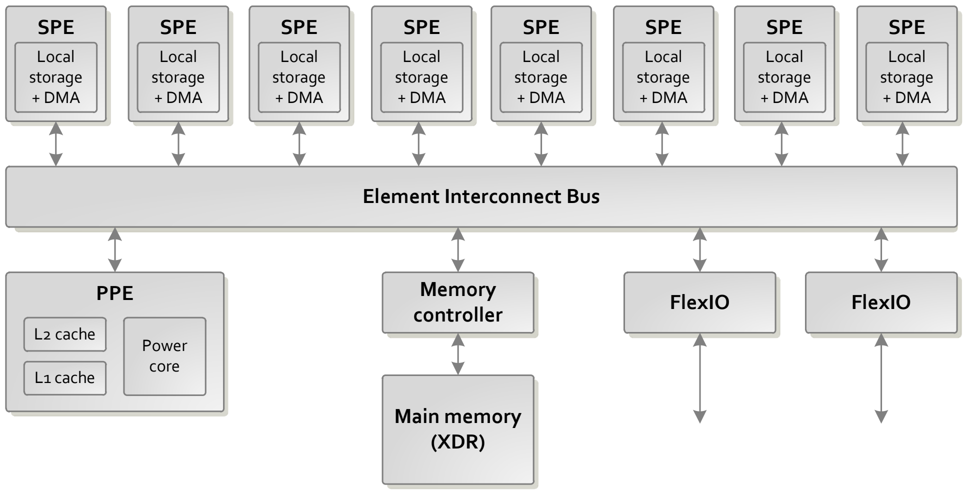

Figure 5.9: Block diagram of the Cell processor architecture [180]

5.1.4 IBM Cell

Cell [122] is a microprocessor architecture that consists of a general-purpose RISC-based core alongside a set of coprocessors designed to accelerate multimedia and vector processing applications. The central RISC core is called the *Power Processing Element* (PPE) (sometimes simply Processing Element) and the surrounding auxiliary cores are called *Synergistic Processing Elements* (SPEs). The cores are connected using an internal bus called *Element Interconnect Bus* (EIB). The memory hierarchy of the Cell makes extensive use of DMA. Connected to the EIB is a memory controller which allows access to external memory. Each SPE, however, is equipped with its own DMA engine that can be scheduled to fetch data into their own local memory. The PPE has two levels of standard cache. A block diagram of the architecture is shown in figure 5.9.

The Cell can be described in AnvilADL using the following code. Note that most Cell configurations contain eight SPEs - the repeat declarations are omitted from the description for brevity.

```
spe0 : processor SPE;
//and declarations for spe1, spe2 etc.

eib : channel EIB;
eib^endpoints = [ppe, spe0, spe1...];

mic : memory XDR;
mic^width = 64;
//mic^size set to the system memory capacity

ppe^memory = mic;

spe0local : memory SPE_LOCAL; //SPE local store
spe0^memory = spe0local;
//and declarations for spe1, spe2 etc.

//Note, it is tempting to add the following
//spe0^extramemory = mic;
//but this is incorrect because SPEs do not have direct access
//to system memory. Instead, the programmer references shared
//data assigned to system memory and the communication layer
//provides the associated DMA to pre-load local storage.

//FlexIO access
flexio0, flexio1 : hardware FlexIO;
flexio0^ports = [ppe, spe0, spe1...];
flexio1^ports = [ppe, spe0, spe1...];

//DMA controllers
spe0dma : DMA SPE_DMA;
spe0dma^memory = [spe0local, mic];
spe0dma^ports = [spe0, ppe];
```

When using CTV-based techniques to target the Cell the following points should be considered:

- The complexity of the EIB can be transparently handled by the EIB channel, associated drivers, and the refactoring engine. The EIB's efficiency is increased by using messages that are closer in length to eight bus cycles. Such information allows the refactoring engine to automatically optimise the size of multi-part messages.
- Mapping the input program to the PPE and SPEs can be handled transparently by the refactoring engine. This allows the programmer to place their code appropriately throughout the architecture. The EIB is organised as a ring so the mapping of pipeline stages to SPEs can have a significant effect on application throughput. CTV's transparent mapping capabilities can aid this considerably.
- Access to the Cell's IO is provided by wrapping the FlexIO IO controller as a custom hardware element. The programmer must access the FlexIO manually, but CTV hides the low-level code required to access it over the EIB.
- The PPE's cache does not need to be explicitly modelled because its operation is transparent to the programmer.

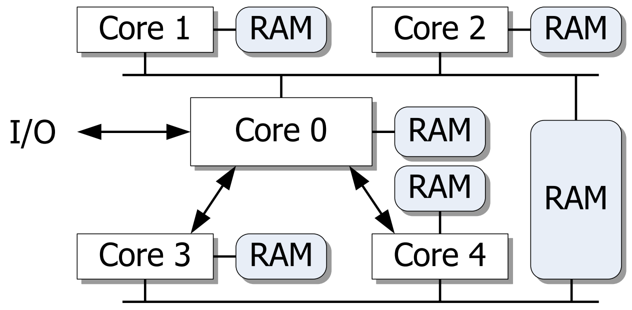

Figure 5.10: 5-core Cell-like system with custom interconnect

- The SPEs are SIMD processors, capable of large, single-cycle vector operations. CTV assumes that a suitable programming model and compiler are used to effectively exploit this capability. Unextended ANSI C is unlikely to stretch the capabilities of the SPEs, making Anvil a poor choice for high-performance Cell programming due to its lack of fine-grained parallelism. This is not a limitation of CTV, however, as a version of C with support for loop-parallel operations could have been used (or an auto-parallelising compiler as discussed in section 2.1.1).
- Note that many Cell programming models also use mailboxes for communication between SPEs. These can be modelled as channels, as shown in the examples of section 5.1.1.

The key to achieving high throughput in Cell programming is effective use of the SPE's memory engines. The SPEs cannot directly access system memory. Instead, accesses are passed through a *Memory Flow Controller* inside the SPE that uses its DMA engine to fetch data from system memory or pass data to another SPE. This can be modelled by CTV by modelling the SPE's local memory and data in system memory as shared data items. Then, when accessed by SPE code the communications layer automatically handles the required DMA accesses. This has to be normally done with manual programmer intervention through the use of custom libraries, or middleware. More information on memory transfers and analysis of efficiency can be found in section 5.2.4.

CTV has an interesting advantage over purely run-time systems because it has access to the entire source code of the system, during the refactoring stage. This allows CTV to automatically schedule memory transfers offline to maximise EIB use. A full investigation into the benefits of this is outside the scope of this thesis however, as this section simply aims to show that the CTV system model can suitably target the Cell processor.

5.1.5 FPGA-based Cell-like system

Figure 5.10 shows a five-core FPGA-based system that is based on features of the Cell and is considered to help show the scalability of CTV and the OM model. In this architecture, cores 1-4 are small Microblaze cores with a 3-stage pipeline (representing the SPEs) whilst core 0 has a floating point unit, 5-stage pipeline, and instruction and data caches. Like the Cell, each core

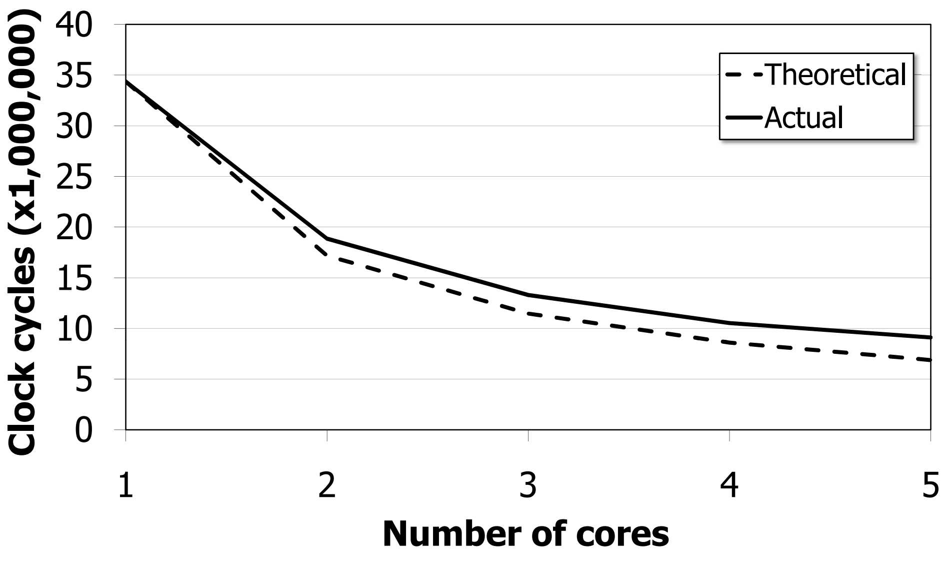

Figure 5.11: Block encryption time for 3-DES compared against the theoretical best-case speedup.

has its own local memory and can access main system memory over the interconnect. The only significant differences from the programmer's perspective is that cores 1-4 are not vector processors and the different bus topology, which has been selected to increase the complexity of the architecture. The architecture consumes 91% of the target FPGA's logic resources and has a maximum throughput of 2160 Dhrystone MIPS at 360MHz.

To test the ability of the communications layer and shared memory system to operate over such a complex architecture, an implementation of the 3-DES encryption algorithm [13] was developed that can scale to a variable number of worker threads. 3-DES was chosen because it is a block cypher that, once the encryption key is distributed to all workers, can be easily parallelised by encrypting multiple blocks concurrently.

The structure of the implementation is shown in figure 5.12. It uses a single controller thread to allocate blocks to worker threads located elsewhere in the system. Each block is modelled as a shared data variable, and access to them protected by associated mutexes (section 4.8.3) to improve the efficiency of the shared memory system. The worker threads are signalled to start by the controller thread which passes the block number to encrypt and the key. Once signalled they begin encrypting their block and the controller thread waits for them to complete. Note that due to the requirement that threads are compile-time static, the worker threads are not being created and destroyed for each block, they are instead passed a list of blocks to encrypt. An implementation could also use a pthreads thread pool.

The efficiency of the shared memory system for this implementation is very good because its memory access pattern is simple and predictable. The entire block is fetched, encrypted, and transmitted back, so no unnecessary copying takes place. Section 5.2.4 evaluates this issue, and compares against existing shared memory systems where appropriate.

Figure 5.11 shows time taken to encrypt fifty blocks of data using for varying numbers of active

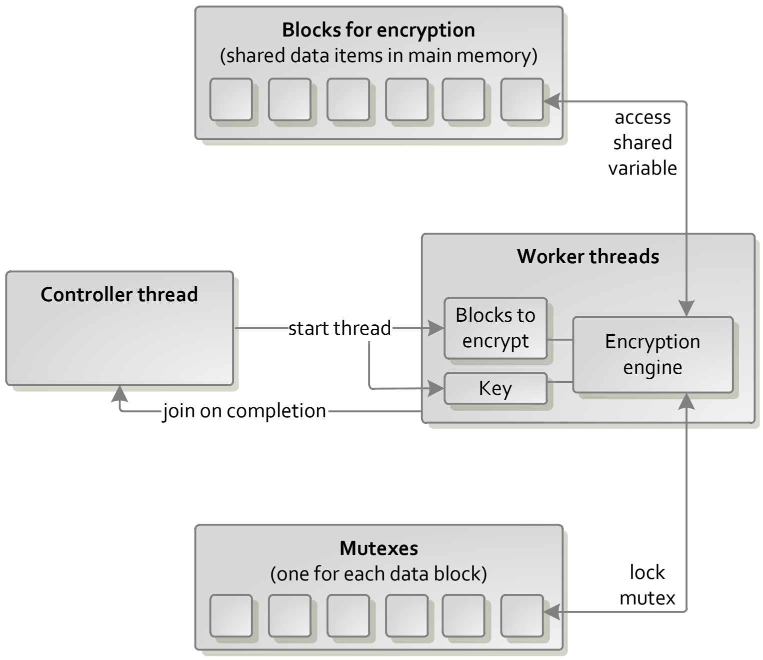

Figure 5.12: 3-DES implementation block diagram.

cores when compared against the theoretical best-case speed up (200% for 2 cores, 300% for 3 cores etc.). The best-case is unattainable because it disregards time taken moving data and for inter-thread communication, but it serves to provide scale and a point of comparison to the results. The results show that the system scales well to five cores and that the distributed nature of the OMs allows the system to execute without the presence of a sole bottleneck. When used with associated mutexes the shared memory system can move the blocks of data efficiently, as evaluated later in this chapter.

5.1.6 IO and custom hardware

As detailed in section 4.10, Anvil allows the programmer to make use of function accelerators and I/O channels that are present in the implementation architecture. The way in which this is achieved uses standard techniques for accessing external devices, but the use of CTV allows Anvil to reduce the burden that this places on the programmer, creating code which is more portable and easier to maintain.

Figure 5.13 shows an architecture built to demonstrate this. The target system is a single-core Microblaze system containing a custom hardware element which is designed to evaluate the quadratic polynomial $y = ax^2 + bx + c$ in IEEE floating point arithmetic. The coefficients a, b, and c are presented to the hardware unit and on the next clock cycle y is available. The Microblaze core used in this system does not have a floating point unit, so any floating point operations specified by the input source code have to be implemented in software by the compiler.

The I/O in the architecture is a standard UART and a set of LEDs connected via a general-

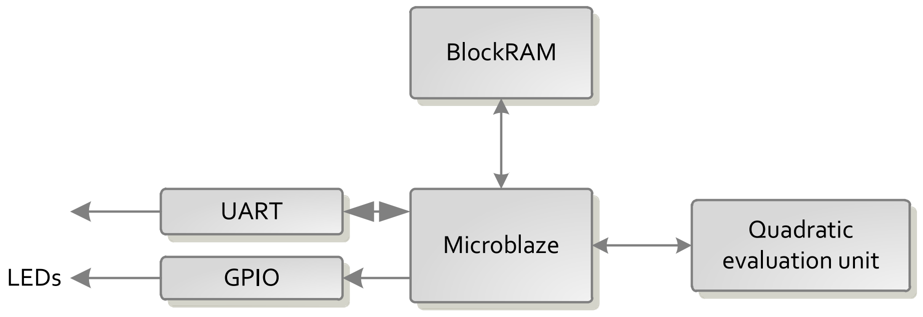

Figure 5.13: Example architecture containing custom hardware elements.

purpose I/O (GPIO) core.

The quadratic evaluation unit, UART, and GPIO are accessed through synchronous ports and therefore the only parameter they require is the memory address at which they are located. The system is described in AnvilADL as:

```
cpu0 : processor Microblaze;
mem0 : memory BlockRAM;
cpu0^memory = [mem0];

quadeval : hardware Quadratic;
quadeval^ports = [cpu0(0x86000000)];

extuart : hardware UART_IO;
extuart^baud = 9600;
extuart^uartsettings = "8N1";
extuart^ports = [cpu0(0x88000000)];

leds : GPIO(8, "O");
leds^ports = [cpu0(0x8A000000)];
```

The Anvil hardware libraries define the interface to these cores through driver functions:

```
//Quadratic evaluation unit
typedef struct {
   volatile int *addr;
} quad_t;

int evaluate_quadratic
      (quad_t *quad, char a, char b, char c){
   *(quad.addr) = (a << 16)|(b << 8)|c;
   return (*(quad.addr + 1));
}

//UART
typedef struct {
   volatile int *addr;
   char recv_buffer[UART_BUFSIZE];
   int recv_start = 0;
   int recv_count = 0;
} uart_t;

void uart_send_char(uart_t *uart, char c) {
  while (*(uart->addr + 2) & 0x08);
  *(uart->addr + 1) = c;
}

//Receive function not shown.
//Adds the received byte to the circular buffer recv_buffer

//LEDs
typedef struct {
   volatile int *addr;
} gpio_t;

void gpio_output(gpio_t *gpio, int val) {
   *(gpio->addr) = val;
}
```

The various structure types hold the port's parameters and any internal values required by the driver. The declaration of these structs is inserted into the source code during refactoring, one for each instance declared in the AnvilADL description. For the above description, the following is inserted.

```
quad_t quadeval = {(int *)0x86000000};
uart_t extuart = {(int *)0x88000000};
gpio_t leds = {(int *)0x8A000000};
```

Once this is done, to access the quadratic evaluation unit the user can write code to use the `evaluate_quadratic` function normally and the refactoring will ensure that it is always compiled to access the correct memory locations. No run-time overhead is associated with this because the code is generated statically at compile-time. Example application code is shown below:

```
int main(void) {
   char c;

   c = uart_get_char(uart);
   gpio_output(leds, c);
}
```

This way of handling external devices is not new, it is the accepted way of doing this in C-based languages. However, the interesting points are that CTV allows the user to define the presence of their hardware in the Anvil ADL and then simply use those devices without any extra structure definitions. The user does not have to copy addresses or interrupt vectors into their code, and interrupt handlers can be automatically applied to the correct devices. This code is portable to any architecture with similar devices present, whereas the traditional method may not be.

Figure 5.14 shows the number of clock cycles taken to evaluate a single equation using software emulation and the hardware accelerator, over a range of input values. The hardware

Software	With h/w acceleration
3808 - 3910 cycles	48 cycles

Figure 5.14: Quadratic evaluation times

version is two orders of magnitude faster than software emulation, only requiring two bus transactions to evaluate an expression.

5.1.7 Texas Instruments OMAP family

The Texas Instruments OMAP (Open Multimedia Application Platform) [58] is a range of System-on-Chip devices developed for use in portable embedded devices. They are frequently deployed in mobile phones or similar devices and feature multi-core architectures with a wide range of integrated peripherals. Programming for the OMAP platform requires being able to make effective use of the following architectural features:

- The OMAP contains a range of memory technologies arranged in a non-uniform memory architecture. The architecture contains caches and DMA engines.
- All devices in the OMAP family contain a wide range of peripherals and I/O devices that can be used to perform specialist tasks.
- Devices in the OMAP range contain an embedded ARM core, a programmable DSP core, and some also contain a GPU. Therefore code must be split across multiple heterogeneous processing elements with different programming paradigms.

CTV is well-suited to targeting architectures with complex memory hierarchies and can effectively describe the different memory types present in the architecture and their locations. For example, The following AnvilADL fragment describes the memory topology of the OMAP3530 [219].

```
//These two lines inform Anvil that coherency must be
//considered, and which CPU driver to use to manipulate
//the cache lines.
main : processor CortexA8;
main^cache = [instruction, data];

//112k on-chip Boot ROM
bootrom : memory ROM;
bootrom^width = 32;
bootrom^size = 28672;

//64k on-chip RAM
intram : memory;
intram^width = 32;
intram^depth = 16384;

//Off-chip RAM
```

```
gpmc : memory GPMC;
ddrsd : memory DDR_SDRAM;
intram^width = 32;
ddrsd^width = 32;
//^depth depends on the attached RAM device
main^memory = [bootrom, intram, gpmc];

//DMA controller
dma : DMA OMAP_DMA;
dma^memory = [bootrom, intram, gpmc];
dma^ports = [main];
```

The GPMC memory controller of the OMAP can control multiple external memory devices. If this is the case in the target architecture then the designer can choose to model the whole GPMC as one large memory space or model the attached memories separately if they have differing characteristics.

Note that the description above does not specify the address ranges for the internal memory devices. On boot, the memory devices are configured and mapped by the OMAP firmware's first stage boot loader. As a result, the above description should be populated with the set up values that are used so that the VP correctly reflects the underlying hardware.

Anvil's shared memory system can use the declared memory spaces to allocate storage. The OMAP already has a unified address space (from the perspective of the ARM core), but Anvil can facilitate the use of DMA and shared memory mapping. Examples of this are explored later in section 5.2.4.

The peripherals that are part of the OMAP can be represented as custom hardware components due to the fact that they are all memory-mapped and can be effectively wrapped in driver code. Examples of how to do this are presented in sections 4.10 and 5.1.6. The peripherals on the OMAP family include the following:

- Serial UARTs
- I2C bus interfaces
- General-purpose IO controllers
- General-purpose timers and watchdog timers
- A2D and image capture chips

These peripherals are all passive, responding to accesses from the main ARM processor, so they are well described by the CTV target layer (section 3.4.4).

The main difficulty with targeting the OMAP is that it contains heterogeneous processors – an ARM9 core, a VLIW DSP core, and a GPU in many cases. Whilst such systems are supported by the CTV system model, Anvil cannot target these processors effectively because DSP programming is unsupported in ANSI C and `gcc`. This is a limitation that results from Anvil's source language. CTV is language-agnostic, so an implementation could be developed that is based on another language or language extension that supports vector programming more easily, such as OpenMP (see section 2.1.1). In this case, CTV would be able to map vector programming constructs to be executed on the DSP or GPU. For example, in OpenMP a vector multiplication can be expressed as follows:

```
int main(int argc, char *argv[]) {
   const int N = 100000;
   int i, a[N];

   #pragma omp parallel for
   for (i = 0; i < N; i++)
      a[i] = 2 * i;

   return 0;
}
```

The `omp parallel for` pragma identifies that the operation should be sent to one of the other processors for execution.

Because of C's unsuitability for DSP programming, Anvil cannot fully support the OMAP architecture. An Anvil extension that allowed vector processing is possible but outside the scope of this thesis.

5.1.8 Summary

This section has demonstrated that both CTV's system model and Anvil's AnvilADL are sufficiently expressive to allow the targeting of modern embedded systems. A caveat is identified that, because C does not support vector-based processors, Anvil also cannot support such processors without the implementation of a language extension. The area of GPU programming (section 2.1.2) is still developing and stabilising so it was not considered by this thesis, but implementation of languages such as OpenCL [169] or OpenMP [39] to support vector processors is interesting further work and fully-supported by the CTV system model.

An observation from this section is that AnvilADL descriptions can be reused in the same way that code libraries can. Section 5.1.4 contains an AnvilADL description for the Cell processor which once created can be reused in multiple projects that wish to target the same hardware. Only the statements which perform architectural mapping would need to change. Accordingly, future work could consider coding styles for AnvilADL descriptions which split the description more cleanly into three sections:

- **Hardware description:** Description of the target architecture in terms of its processors, memory spaces, channels, and hardware elements. Definition of cluster targets. Identification of DMA engines. Specification of interrupts, address spaces, etc.
- **Software description:** Cluster definitions. Mutex associations. Anvil POs. Instances of OMs and the items they manage.
- **Mappings:** Mappings of threads to processors and data to memories. Assignment of clusters to cluster targets. Mapping of OMs to processors.

The hardware description section need only be updated if the hardware is revised, and can be reused by other projects that are to be executed on the same architecture. The software description section need only be updated if the software is changed and can be reused when the same application is to be ported to a new target architecture. The mapping section is unique to the combination of application and target hardware and must be updated when either change.

In this section the range of supported processors is limited to those which can be easily implemented on the experimental hardware, a set of Xilinx FPGAs. It should be noted however that support for additional processor architectures can be easily included into Anvil. Support for a processor requires a compatible compiler and toolchain, and that processor drivers are written to implement the architecture-level functions of the embedded pthreads library (defined in section 4.9.2).

5.2 Overheads

Sections 3 and 4 claimed that the compile-time nature of CTV allows it to introduce a much smaller amount of overhead into the final system than equivalent run-time techniques. This section investigates this statement.

Section 5.2.1 enumerates the potential areas from which overhead may originate in an embedded system. Sections 5.2.2 and 5.2.3 then go on to discus how the compile-time nature of CTV can be leveraged to reduce these. It demonstrates that in the worst-case, CTV can choose to 'fall back' to a purely run-time system and therefore does not increase overall system overheads. Sections 5.2.4 and 5.2.5 then examine in detail the overheads of the shared memory system. Finally, section 5.2.6 looks at the compilation overheads introduced by CTV.

5.2.1 Sources of overhead

To analyse CTV's effect on run-time overheads it is necessary to first consider the potential sources of overhead in existing systems. When developing systems in a high-level programming language, overhead is used here as an informal measure of the difference in performance between:

- the best possible series of machine instructions to achieve a given task,
- and the best high-level program that achieves the same task, and that can be written without leaving the high-level programming model.

The qualification about not leaving the high-level programming model is because it is possible to write abstraction-breaking programs such as a C program that is just a wrapper for inline assembly code. Overhead is not codified as a quantitative measure because 'performance' can refer to different design goals such as memory usage or data throughput.

Modern programming languages may introduce overhead in any of the following areas:

- **Interpretation:** Some systems, such as standard Java or Smalltalk, do not compile to native opcodes and must be either interpreted or 'just-in-time' compiled using a JIT compiler [138, 266]. This allows for a very flexible and dynamic language but requires the presence of a virtual machine and compiler at run-time which imposes a large memory footprint. Also, interpreted languages are frequently slower than natively compiled ones.
- **Heavyweight communications:** Because languages are general-purpose and the run-time is not given architectural information, the communications layer implemented by standard languages is very general and lacks architectural specialisation. Custom architecture communications channels may be unused. Also, because the language run-time does not have any knowledge of the input program the communications layer inserts many unnecessary layers of indirection. Even when two static threads communicate, the layer will frequently require an OS call or similar to determine the threads' current locations, rather than refactoring the input code to simply make the communication directly.
- **Data copying:** When two threads share data but execute in physically separate areas of the architecture, copying and updating of shared data is inevitable. However, modern languages frequently do not contain a model of the target memory hierarchy and communications infrastructure, resulting in programs which are forced to communicate and share data inefficiently.
- **Operating system support:** Most modern languages cannot execute without some form of OS or rub-time support. The OS provides services, like threading, communications, coordination etc. and the various threads of the system must call in to the OS to use them. However, this creates a bottleneck, because almost always the OS only executes on a single processor of the system, forcing all other cores to call in to it frequently. Some languages (such as Ada) can replace the OS with a complex language run-time, but the bottleneck problem remains.
- **Memory management:** Most languages assume a single logical address space over the entire architecture to simplify programming. This is frequently an unworkable assumption with large NUMA systems, but on simpler architectures it can still place sizable demands on the hardware. Complex memory management units (MMUs) are required to maintain this abstraction. MMUs are very expensive in terms of silicon area and many embedded processors do not have them.
- **Cache coherency:** Languages tend to assume perfect cache coherency across the entire system. Programmers are not asked to consider that in some cases shared data might be incoherent. As a result, as systems move to hundreds of cores or more complex topologies coherency becomes increasingly unworkable. Rather than only maintaining coherency for data objects that require it, all memory is kept coherent across all caches in the system because that is what is required by the programming model. This leads to a large amount of overhead in terms of execution time (in the case of software cache coherency) of silicon area (for hardware cache coherency).
- **I/O and interrupts:** Low-level device driving requires efficient handling of interrupts and direct memory-mapped I/O. In higher-level languages that abstract away from hardware features (such as Java), inefficiency can be introduced, either because the language model does not consider such details, or because OS calls are required.

The following section considers the way in which CTV attempts to reduce these areas of inefficiency.

5.2.2 Reduction of overheads due to CTV

Many of the identified sources of inefficiency in section 5.2.1 originate from the fact that input code is written without architectural knowledge and the run-time system is executing without knowledge of the application. Accordingly, the two must interact using general-purpose APIs during execution and cannot specialise without losing generality. CTV instead provides architectural mapping information at compile-time, allowing a general programming model but one that can still specialise to reduce overheads.

This gives CTV the potential to reduce run-time overhead in the following areas:

- **Heavyweight communications:** Because the architectural mappings are provided at compile-time, CTV's communications layer can statically route communications between non-migrating elements. (Anvil's implementation of this is described in section 4.7.) The overheads associated with run-time dynamic routing are only incurred for situations where they are absolutely required. The programmer can also direct communication over specific communication channels to make efficient and predictable use of the underlying hardware.
- **Data copying:** Copying data cannot be avoided, but CTV's compile-time information allows the shared memory system of an implementation (Anvil's is described in section 4.8) to determine at compile-time when copying is required. This information can be leveraged to preload caches and schedule DMA transfers.
- **Operating system support:** CTV distributes OS services over the architecture in a scalable manner using the OM model, preventing system bottlenecks (section 3.7).
- **Memory management:** Most embedded systems only use statically-allocated data and do not load external code, so they do not require complex memory management. CTV can use code refactoring and the OM model to provide more complex memory management from software when it is required (section 4.8).
- **Cache coherency:** The clustering model (section 3.2) allows the programmer to limit the required coherency in their system, thereby allowing the system to relax coherency requirements and only maintain coherency between tightly coupled areas that require it.

Every system displays a certain amount of inescapable overhead that originates from the need to move data across non-uniform memory architectures or send messages over communication channels. Assuming efficiently-written input code, this overhead cannot be reduced. *Non-essential* overhead in addition to this is what CTV attempts to eliminate.

When implemented as a solely compile-time system, there are two possible areas in which a CTV-based system could perform worse than a middleware-based run-time system:

- CTV uses a general-purpose communications protocol for implementing architecture support libraries and the OM model, that is based on the programming model of the chosen language. (These messages are detailed in appendix A.) A programmer with application-specific knowledge could implement a custom communications system that performs better for the target application.

Benchmark	With CTV	Without CTV
quicksort-fpu	26,222	26,222
fast-dct	7,716	7,716
binarysearch	114	114
linkedlist	3453	3453

Figure 5.15: Evaluation time (clock cycles)

- Whilst the specifics of the shared memory, routing, and cache coherency algorithms are not specified by CTV's system model, they are applied at compile-time. It is not always possible for the compiler to statically determine exactly which elements of shared data are read in a given code block, so as a result a compile-time system may be forced to assume the worst and transfer more data than is actually required.

In general however, the extra overhead that the use of CTV can introduce is provably no worse than that of existing systems, because in the worst case, CTV could simply choose to implement existing run-time shared memory and communication systems. (For example, Adsmith or Rthreads, see section 3.6.2.) This would remove many of the areas in which CTV has the potential to leverage compile-time information to reduce overheads, but it demonstrates that the use of CTV is never required to make a system less efficient.

CTV's compile-time overhead reduction can be contrasted with the just-in-time (JIT) compilation employed by the JVM or languages like Haskell. JIT compilation allows run-time acceleration of code, but requires the presence of a full compiler and general-purpose VM. CTV's compile-time specialisation makes use of the extra information provided by the system model to reduce these requirements at the expense of increased compilation overheads.

CTV's potential to use compile-time information to reduce overheads is illustrated effectively when Anvil is used to target a single-processor system with a single block of memory - the standard Von Neumann-style architecture assumed by C's programming model. Because this architecture does not require the use of any of Anvil's libraries, no run-time overhead is introduced at all. The VP and the actual platform are the same, meaning that the virtualisation reduces to nothing. With a run-time system that makes use of translation or virtualisation layers (Java / Smalltalk) or heavyweight middleware (such as CORBA [183]) even when the benefits of these systems is not needed they still introduce significant run-time overheads.

Figure 5.15 shows the execution times of four benchmark programs when run with and without the aid of Anvil. It shows that because of the compile-time nature of Anvil, it does not need to alter the code so that it executes correctly on the target architecture and the resulting execution time overhead is therefore 0%.

Having demonstrated that CTV's worst-case performance is the same as existing run-time systems, the following sections detail aspects of Anvil, enumerate its performance, and where appropriate compare with existing systems.

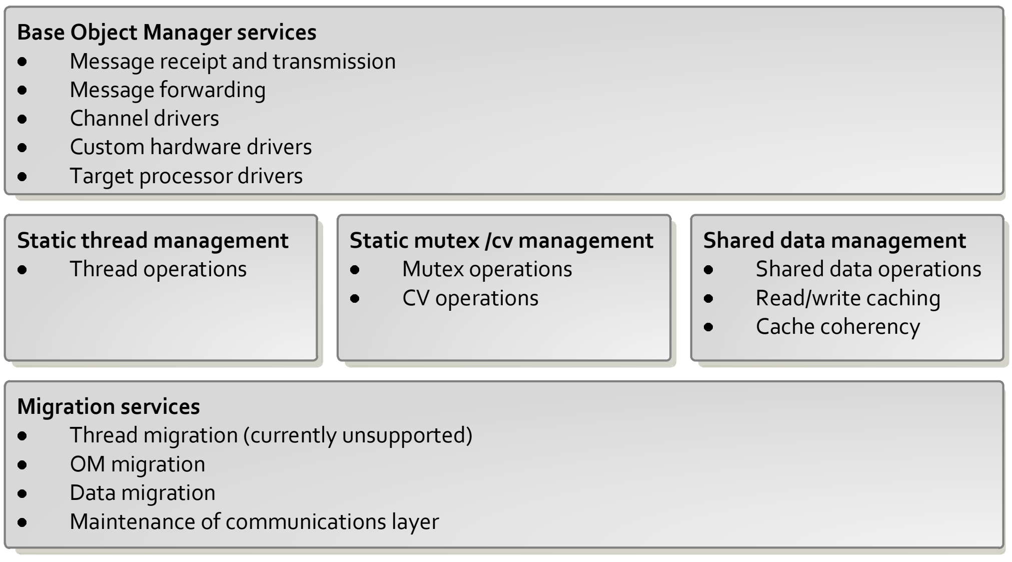

Figure 5.16: The service layers of Anvil's OMs

5.2.3 Object managers

Section 5.2.2 showed that CTV can reduce its impact to the minimum required by the input application. This section shows this effect on the OMs of the system, and how they may change in response to the source application.

OM features are added according to figure 5.16. Every OM provides a base set of features that allow it to form part of the communications layer and forward messages for parts of the system. If the OM also manages objects, the services for those objects are included in its compiled code base. Finally, if migration is supported then a further layer of code is required to provide migration services. Figure 5.17 shows how the code size of the OM changes as its supported features change. Note that this table shows indicated sizes only. Most features of the OM share functions and data structures so it is difficult to precisely apportion code to specific feature sets. Also, these numbers are for a Microblaze OM, other processors may be slightly different.

Even though the current Anvil code base is not optimised, it can be seen that OMs only impose a small code footprint on the embedded processor, up to approximately 7.6kB. For comparison, the Xilkernel microkernel (which is heavily optimised) requires 16kB of code for basic round-robin scheduling and limited interrupt handling, or 22kB for its full feature set [256].

The scale of these sizes is highlighted by considering the memory footprint of run-time middleware solutions such as CORBA ORBs. Although not directly comparable due to their differing feature sets, Real-Time CORBA [247] is aimed at deployment in real-time and embedded systems despite the TAO ORB measuring 2075kB in size, and the ZEN Real-Time ORB measuring approximately 2539kB [135].

The OM's throughput does not change by a significant amount as features are added. The

Configuration	**Code size (bytes)**
Base	5256
Manage mutexes	6180
as above plus threads	6656
as above plus CVs	7092
as above plus shared variables (manage only)	7200
as above also accessing shared variables	7390
as above plus POs	7490
Full	7790

Figure 5.17: The code footprint of an OM scales according to its use. Numbers given for the Microblaze soft processor.

reason for this is that OM features are triggered by the receipt of request messages. Therefore the main body of the OM, once it has assembled an incoming message simply tests the message's type and jumps to the appropriate handler routine. The code looks like this:

```
void process_message(int buffer) {
   switch(bufferitem(buffer, 2)) {
      #ifdef _ANVIL_OM_BASE
      case M_FORWARDMESSAGE:
         //Handle
         break;
      #endif

      #ifdef _ANVIL_OM_THREAD
      case M_THREADCREATE:
         //Handle
         break;

      case M_THREADEXIT:
         //Handle
         break;
      ...
```

OMs also require memory for their internal data structures. In Anvil this data footprint is predictable because these structures are statically allocated at compile-time. Other implementations may choose to dynamically allocate this data from a heap to increase flexibility at the loss of predictability. Because of this static allocation, each OM is parameterised by two values:

- `_ANVIL_REQUEST_QUEUE_SIZE`: Determines the maximum number of requests that can be queued on a managed object (mutex, shared variable, etc.) This is currently set to its worst-case value, which is the number of processors (including the current processor) that host threads that access this object.
- `_ANVIL_CIRC_BUFF_SIZE`: The number of concurrent unbuffered messages this processor can handle. This is currently set to its worst-case value, which is the number of channels and custom hardware items that the processor is connected to.

The required memory for this data is shown in the following table:

Item	Size
Message buffers	$3 + (34 \times$ _ANVIL_CIRC_BUFF_SIZE$)$ bytes
Hosted thread state	$4+(4*$_ANVIL_REQUEST_QUEUE_SIZE$)$ bytes per currently hosted thread
Managed thread state	$3 + (4 *$ _ANVIL_REQUEST_QUEUE_SIZE$)$ bytes per thread
Managed mutex state	$3 + (4 *$ _ANVIL_REQUEST_QUEUE_SIZE$)$ bytes per mutex
Managed CV state	$2 + (4 *$ _ANVIL_REQUEST_QUEUE_SIZE$)$ bytes per CV
Managed PO state	$3 + (4 *$ _ANVIL_REQUEST_QUEUE_SIZE$)$ bytes per PO
Managed shared variable state	5 bytes per shared variable
OM local variables and buffers	Approximately 56 bytes not including stack frame overhead. Varies depending on target architecture.

5.2.4 Shared memory

This section examines Anvil's shared memory system (section 4.8). This evaluation focusses on the following two features of shared memory systems:

Transfer selection An efficient shared memory system should only transfer bytes that are required by the application. For example, if a processor only accesses part of a remote array, only that section should be transferred to it before execution and transferred back afterwards. This section considers the volume of data transferred by Anvil's shared memory system and only analyses the input code and the way in which it is refactored.

Transfer efficiency Once the algorithm has decided which bytes to transfer, they should be moved efficiently without requiring undue processing overhead. This section uses actual implementation tests to measure execution and transfer times.

These are discussed in the next two sections.

Transfer selection

The shared memory system implemented by Anvil is functional and demonstrates optimal or close to optimal performance for some programs, but is less effective for others. Due to the fact that Anvil's base language, C, does not consider non-uniform memory architectures, the shared memory system cannot always extract enough information to achieve an optimal implementation. This section will show how different programming styles can affect efficiency.

Section 4.8 described the algorithms that are used by the shared memory system. Migration semantics are implemented, meaning that before an item of shared data is accessed it is transferred from the remote location to the memory space of the accessor node. If the data is stored in memory that is shared between the remote node and the accessor node then this transfer is not required, and only cache coherency is considered (section 5.2.5).

The first presented test illustrates the importance of using an associated mutex. Consider the following code in which a thread is created for performing a scalar vector addition. The created thread also sums the elements of the vector and outputs it to the user.

```
int shareddata[50];
pthread_t thread1;
pthread_mutex_t mux;

void *vectoradd(void *add) {
   int i; int sum;

   pthread_mutex_lock(&mux);
   for(i = 0; i < 50; i++) {
      shareddata[i] = shareddata[i] + *(int *)add;
      sum = sum + shareddata[i];
   }
   pthread_mutex_unlock(&mux);

   //Report sum to the user
   ...

   pthread_exit();
}

void main {
   int x = 20;

   //Fill shareddata with data
   ...

   //Spawn a thread to do the vector addition
   //Pass it the value of x
   pthread_create(&thread1, 0, vectoradd, (void *)&x);

   //Wait for thread to complete
   pthread_join(thread1, 0);
}
```

Anvil does not automatically associate mutexes with shared data items, so the connection between `mux` and `shareddata` must be made explicitly using AnvilADL. If it is not made, then Anvil inserts calls to read and write `shareddata` before each time it is read and before each time it is written respectively. Accordingly, two calls to `_anvil_read_sv` are added and the code of the `vectoradd` function is refactored as follows:

```
void *vectoradd(void *add) {
   int i; int sum;

   pthread_mutex_lock(&mux);
   for(i = 0; i < 50; i++) {
      _anvil_read_sv(0, i * sizeof(int), sizeof(int), 0);
      shareddata[i] = shareddata[i] + *(int *)add;
      _anvil_write_sv(0, i * sizeof(int), sizeof(int), 0);

      _anvil_read_sv(0, i * sizeof(int), sizeof(int), 0);
      sum = sum + shareddata[i];
   }
   pthread_mutex_unlock(&mux);
```

Situation	Items transferred
Without assoc. mutex	150 (100 read, 50 write)
With assoc. mutex	100 (50 read, 50 write)
Optimal performance	100 (50 read, 50 write)

Figure 5.18: Anvil's shared memory can attain optimality in some situations.

```
   //Report sum to the user
   ...

   pthread_exit();
}
```

Anvil has not associated the mutex with `shareddata` so it can not guarantee that it is operating under mutual exclusion and is forced to fetch the same data twice. Also, it is forced to fetch and write back the array in single item chunks, one per loop iteration, rather than copying the entire array at once. Anvil can request the required values from the array individually because the shared memory system supports pointer-based offset-driven access as described in section 4.8.4. This however means that the transfer cannot take advantage of burst-mode transfers or DMA.

If the programmer's AnvilADL associates the mutex with `shareddata` then the function is instead refactored as follows:

```
void *vectoradd(void *add) {
   int i; int sum;

   pthread_mutex_lock(&mux);
   _anvil_read_sv(0, 0, sizeof(shareddata), 0);

   for(i = 0; i < 50; i++) {
      shareddata[i] = shareddata[i] + *(int *)add;
      sum = sum + shareddata[i];
   }
   _anvil_write_sv(0, 0, sizeof(shareddata), 0);
   pthread_mutex_unlock(&mux);

   //Report sum to the user
   ...

   pthread_exit();
}
```

Figure 5.18 shows the amount of data that must be transferred in each situation. As can be seen, when Anvil uses an associated mutex optimality can be attained in this situation.

The above code is a good example of the kind of program that Anvil's shared memory system handles very well. The shared data is accessed in its entirety so it is not inefficient to transfer the entire array when its associated mutex is locked. Many embedded systems with high performance requirements access data in this way because they implement encoders, image processors or similar systems.

The next example shows the situation that that Anvil handles poorly. Consider the following code:

```
sample_t samples[];
pthread_mutex_t mux;

void sw_fft(int start, int end) {
   pthread_mutex_lock(&mux);

   //Process the specified portion of the sample array
   //Implementation omitted

   pthread_mutex_unlock(&mux);
}

void main(void) {
   int pos;

   while(processing) {
      //Process the sample window
      sw_fft(pos, pos + 64);

      //Slide the window
      pos++;
   }
}
```

This code implements a sliding window FFT algorithm, in which a window is slid over the sample array one element at a time. This is a fairly common paradigm in signal processing applications as it allows a circular sample buffer to be used indefinitely. Unfortunately because of the low-level nature of C, the code contains no higher-level information about the behaviour of this algorithm. The optimal behaviour is to fetch the first 64 samples on the first call to `sw_fft` as a bulk transfer and then on subsequent calls fetch only one additional sample, discarding the oldest sample resulting in $64 + i$ transfers, where i is the number of times the window is processed.

Anvil's read caches should be able to reach this optimal performance (if a perfect LRU replacement policy is implemented) but it cannot because of the way that the programmer has locked the associated mutex. Because the mutex is only held for the duration of the call to `sw_fft`, Anvil is forced to refetch the samples on each call because they may have changed between mutex locks. If the programmer moved the mutex lock to surround the main while loop then optimality would be obtained. This is shown in figure 5.19.

It should be noted that similar problems arise when considering cache coherency schemes, as the problem of maintaining a coherent cache is equivalent to maintaining a distributed shared memory system.

This starts to expose the main weakness with Anvil's shared memory system. Here, the programmer is forced to think about the way that data flows throughout their program and place the associated mutexes accordingly. Whilst this can be seen as a reasonable compromise and necessary, the C programming model which is exposed by CTV's VP promotes transparent

Situation	Items transferred	DMA possible
Mutex locked in `sw_fft`	$64 \times i$	No
Mutex locked in `main`	$64 + i$	No
Optimal performance	$64 + i$	Yes for initial transfer

Figure 5.19: The programmer is forced to consider placement of associated mutexes to get close to optimality. i is the number of iterations of the sliding window.

shared memory. The C programmer is not used to such concerns, and using mutexes to control data flow is unusual. Also, DMA is still not possible because there is no way to express that on the first call to `sw_fft` the read cache should be bulk filled, although this could be added in an extension to AnvilADL. Using CTV with other languages that are more explicit about data transfer (such as PGAS languages, section 2.1.4) would help with this. CTV, unlike run-time systems, can benefit greatly from increased expressibility in the source language.

The final example demonstrates another interesting situation that can arise from Anvil's shared memory system. The following code shows a binary search algorithm that executes over a shared array.

```
int shareddata[50];
pthread_t thread1;
pthread_mutex_t mux;
int result;

void *binary_search(void *t) {
 //Values hard-coded for clarity
 int low = 0;
 int high = 49;
 //Dereference the target value
 int target = *(int *)t;

  pthread_mutex_lock(&mux);

 //Perform the binary search
  while (low <= high) {
     int middle = low + (high - low) / 2;
     if (target < shareddata[middle])
        high = middle - 1;
     else if (target > shareddata[middle])
        low = middle + 1;
     else
  {
   //Return result to calling thread
   result = middle;
   break;
  }
  }

  pthread_mutex_unlock(&mux);
 pthread_exit();
}
```

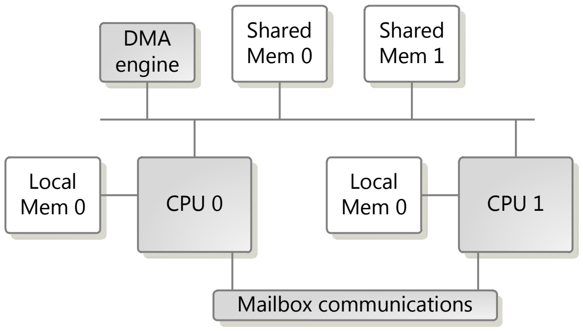

Figure 5.20: A non-uniform memory architecture to evaluate the shared memory system.

```
int main(void)
{
 int x = 23; //Value to search for
 pthread_create(&thread1, 0, bsearch, (void *)&x);
 pthread_join(thread1, 0);

 //Result can be read from shared variable 'result'
}
```

Because this algorithm relies on random access to the shared array, the programmer might specify that the shared memory system should bulk transfer the array when the associated mutex is locked using the `fetchall` attribute (see section 4.8.3). This allows Anvil to use burst transfers and DMA and is very efficient.

The problem with this is that the average case (and worst case) performance of this code is $O(log\ n)$, meaning that for the example above with 50 items in the array, only six reads are required in the worst case. As a result, fetching the entire array is very wasteful and it is actually faster in this case to not associate `mux` with the shared data and only fetch data when it is required. Anvil will fetch exactly the data required, and although these fetches will be slower because they are random and so cannot bulk copy, this is likely to be faster overall.

Transfer efficiency

Whilst the previous section concentrated on the efficiency of Anvil's shared memory system in terms of the data that it elected to transfer, this section analyses the overhead inherent in those transactions. All tests execute on the architecture shown in figure 5.20.

Two types of transfer are possible:

Memory-to-memory transfers When a single processor is moving data between two memory spaces that it can directly access. (Common in SMP architectures.)

Processor-to-processor transfers When the source and target memory spaces are not both

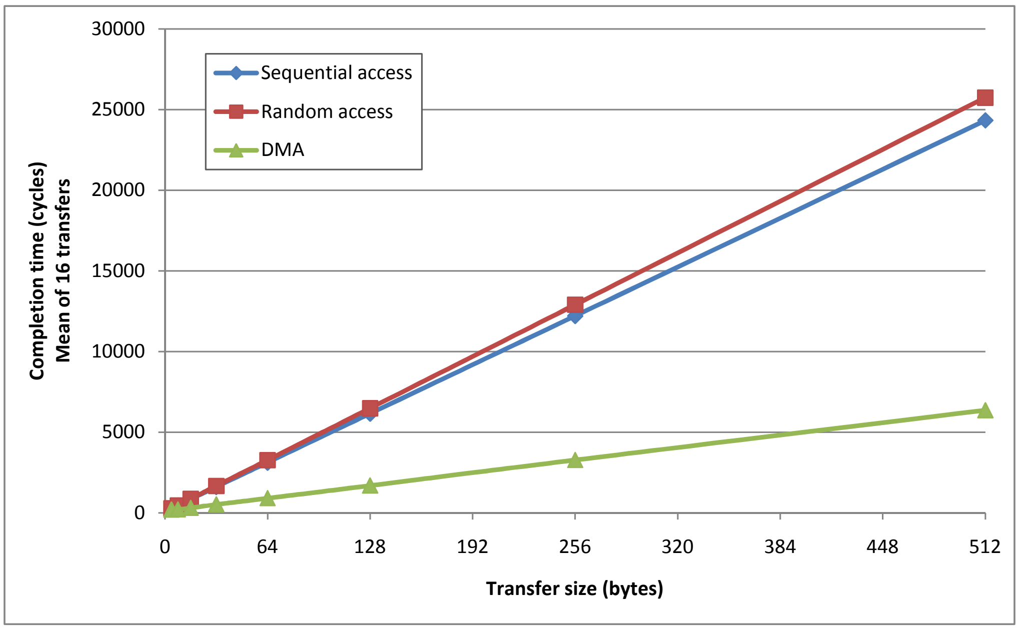

Figure 5.21: Transfer rates of shared data between the two shared memory blocks ('Shared Mem 0' and 'Shared Mem 1') using different transfer modes.

visible to any one processor, so two processors must cooperate to move the data. (Common in NUMA architectures.)

These are discussed in the following two sections.

Memory-to-memory transfer

The shared memory system transfers shared data by passing it from the OM that manages the shared data item to the processor which requested it. This requires processing time from both processors, but if DMA engines are available in the target architecture then this can be offloaded to allow the processor to execute other threads. The test architecture provides a DMA engine (the Xilinx-provided XPS Central DMA Controller [261]) that is accessible from processor 0 and can be used by Anvil.

In the first test, figure 5.21 shows the transfer times for various sizes of shared data item when transferring data between the two blocks of shared memory ('Shared Mem 0' and 'Shared Mem 1') using different transfer modes. Because this example is transferring between the two shared memory blocks, DMA can be used when the transfer is a sequential access.

The graph shows that the difference between random access and sequential access in this architecture is minimal, although sequential access is slightly faster in all cases. This is because the relatively slow speed of the soft processor masks the access time of DDR memory. This difference will become more apparent when using faster processors that can saturate the memory controller. DMA access is however much faster because the DMA engine is dedicated

Size (bytes)	Transfer time (cycles)	Time consumed by transmission delay
4	1212	28%
8	1392	37%
10	1340	39%
40	2760	69%
80	4512	80%
120	6178	86%
240	11368	92%

Figure 5.22: Time taken to transfer shared variable data in the architecture of figure 5.20

hardware that does not experience the bus transaction and execution fetch overhead that the processor does. Another advantage of using DMA is that the processor can execute other threads whilst waiting for the transfer to complete.

A measure of efficiency for these transfers can be obtained by comparing the amount of cycles that are spent waiting for the memory and bus transfer against the amount of cycles spent processing the transfer. This does not apply to DMA-based transfer. For the sequential access and random access results, on average 86% of the execution time involves initiating bus transfers or waiting for the memory controller. Therefore only 14% of the execution time (3646 cycles for the 512 byte transfer) is spent executing code from the Anvil shared memory libraries.

Processor-to-processor transfer

In the previous test, only one processor was involved in the transfer because it had direct access to both the source and destination memory spaces. This test demonstrates the other form of transfer that the shared memory system must account for, in which this is not the case. The transfers in this test are between the two local memory spaces ‘Local Mem 0’ and ‘Local Mem 1’. This requires processor 0 to request data from processor 1, and the data is transferred using the communications layer. Note that DMA cannot be used in this situation because the memory spaces are local to the processor.

The results of this test are shown in figure 5.22, and the transfer times can be broken down as follows:

- Approximately 760 cycles are required for the initial request packet sent to the processor hosting the shared data’s OM.
- The processor hosting the shared data’s OM suffers a base overhead of approximately 100 cycles responding to the request packet’s interrupt, parsing it, and creating a new transmit buffer to start the transfer.
- From that point, the buffer can be filled trivially with `memcpy`, taking around 10 cycles per word. If unaligned accesses are supported, or data marshalling is required, this buffer assembly time will take extra processing. Equally, if the memory is slower then fetching will consume more cycles.

- On the mailbox-based communication channel of the test architecture, transmission takes 162 cycles per four byte word, assuming no bus congestion.
- The receiving processor suffers approximately 180 cycles of overhead setting up a receive buffer.
- As above, copying from the receive buffer to normal memory is trivial, but extended if accesses are unaligned or marshalled.

Figure 5.22 also shows the percentage of the transfer time which is consumed waiting for the communications medium to transfer the raw data. As this trends towards 100% the transfer reaches maximum efficiency as no faster transfer is possible on the target architecture. As can be seen, for the smallest transfers the complete transfer time is dominated by the time taken to assemble, transmit, receive and parse the initial request packet. However, as the request grows in size the transmission time of the reply begins to dominate and therefore increase the overall efficiency. This highlights the importance of using bulk transfers rather than random shared variable access.

5.2.5 Cache coherency

The cache coherency algorithm implemented by Anvil is described in section 4.8.7. Anvil implements a weak coherency system that ensures correctness by emptying cache lines whenever coherency problems can arise (when a node writes to shared data that another node has previously read). Anvil refreshes the entire shared variable in these situations, which is inefficient if only a small part of a large array has changed. Coherency algorithms are well-studied and more efficient algorithms can be implemented as an extension to Anvil's OMs, but this is outside the focus of this work. As a result the current performance of Anvil's coherency system is not evaluated.

There are many areas in which the compile-time nature of CTV may be able to assist in the implementation of more efficient coherency systems. For example, clustering information can be leveraged to provide strict coherency inside a cluster and only weak consistency between clusters. Equally, nodes that do not access a given shared data items do not affect its coherency, resulting in a more scalable system that current solutions which attempt to keep the entire of memory coherency across all nodes. In the worst case, an existing run-time system can be implemented, so therefore the use of CTV does not result in a less efficient system than what is already possible with the state-of-the-art.

5.2.6 Compilation times

A CTV-based system trades run-time overheads for compile-time overheads. A lot more work has to be performed at compile-time, including parsing, static analysis, code splitting, code refactoring, and then traditional compilation of code for each target processor. As a result, compilation times in a CTV-based system are much greater than for a run-time only system. Furthermore, the Anvil parsing and refactoring engine (section 4.3) is written in the Python programming language. Because Python is interpreted, its execution speed is much slower than a native programming language.

Number of cores in target architecture	Average compilation time (seconds)
1	0.391
2	0.662
3	1.031
4	1.330
5	1.502

Figure 5.23: Compilation times of benchmark programs when using Anvil.

It is difficult to make meaningful comparisons of the time taken by a CTV-based compiler because it is automating much of the architectural-mapping work that has in other systems to be performed by the programmer (or by a run-time middleware layer every time the program is executed). In general, compilation takes much longer than normal but is not overly onerous.

The primary factor that determines compilation time for the current implementation is the number of target processors. Each processor requires a new refactoring pass and invocation of `gcc`. The benchmark programs used in section 5.1.3 were compiled for a range of architectures with differing numbers of cores and the results are shown in figure 5.23.

For all input programs, the actual code had only a small effect on the overall execution time as parsing time is dominated by the time taken to load the Python interpreter. The output stage is slow, taking around 300ms per target processor for these relatively simple examples. For comparison, standard compilation of programs this size takes around 0.1 seconds.

In general, CTV takes many times longer than traditional compilation. However, the slow implementation of the Anvil compiler, the requirement to load the Python interpreter, and the lack of optimisation are heavily contributory to this. Given equivalent levels of implementation quality, the refactoring engine should take no longer to parse the input code than the actual compiler [2]. The AnvilADL description then has to be parsed, but AnvilADL is a simple language and the descriptions are very much smaller than the application source code. Also the refactoring engine should take a comparable amount of time to the intermediary code generation phase of the compiler.

This section highlights that the major difference experienced is that, because libraries are generated and compiled for each target processor, CTV-based compilation times scale linearly with regards to the number of processors in the target system. Therefore it is expected that compilation time for a single processor system could be optimised to be of a similar order of magnitude to normal compilation, but each additional target processor will require a similar amount of compilation time again.

5.2.7 Summary

In summary, this section has demonstrated that the run-time overheads present in Anvil are very low when compared with similar run-time systems. This is attributed to the code special-

[2]Note that it is possible to hook into the compiler to use its parsing stage directly and avoid the multiple parses that are currently required, but this was avoided to demonstrate that CTV can operate with an unmodified compiler.

isation and architectural mapping made possible by the CTV methodology, and to CTV's use of the OM model.

Investigation has shown that Anvil's shared memory system, whilst providing transparency, still requires the programmer to consider the way that they write their code in order to ensure efficient operation of the distributed memory and cache coherency algorithms. In all examined cases, close to optimal behaviour can be obtained although it was identified that an extension to support DMA for sliding window algorithms might be beneficial. This highlights that there is only so much that can be done without any support from the language. However, CTV's unique ability to present a simplified programming model thorough the use of Virtual Platforms allows for architectural mapping of amenable code to be performed automatically, greatly assisting deployment on complex architectures. Also, unlike a post-partitioning approach in which the architecture of the software is heavily influenced by the implementation target, CTV and Anvil allow for simple solutions to be quickly developed and tested without any hardware knowledge. These might display inefficient memory usage patterns, but will operate correctly and provide useful testing and profiling information. Memory use can be corrected and optimised later, once hardware and software architectures are finalised.

CTV's VP exposes a single logical address space which can encourage programmers to forget about the way that they use shared variables. Whilst CTV guarantees that any code (subject to its restrictions) will execute correctly, if the shared memory use is not coded as discrete transfers then the code will be inefficient. Also, because of the VP's single address space abstraction, it might be slightly unclear to the programmer *why* their code is inefficient. This is not a problem for analysis or predictability as the refactoring stage can be as predictable as required, but IDE feedback to support the developers to make these decisions would be helpful in an industrial implementation of CTV.

Another avenue of research would be to implement an analysis phase which attempts to determine which policy is more appropriate, or to use a run-time 'tuning' algorithm which changes its behaviour based on previous executions.

5.3 Clustering

The clustering model (section 3.2) that is supported by CTV allows the system designer to guarantee that their applications will maintain locality of computation and data when executing on highly dynamic platforms. The two-level logical layer (section 3.4.3) allows greater expressibility than affinities or other forms of direct mapping that are usually the only available placement mechanism.

The argument behind focussing on the maintenance of locality is that it will result in a lower execution time for the running application due to decreased communication latencies. However, cluster-based migration requires extra overhead at each migration point because the entire cluster must be migrated rather than a single object. As a result, there are some situations where the use of clustering is less efficient than simply allowing unrestricted migration.

To further explore the circumstances that affect this, experiments have been performed (detailed in section 5.3.1) and a simulator developed (detailed in section 5.3.2).

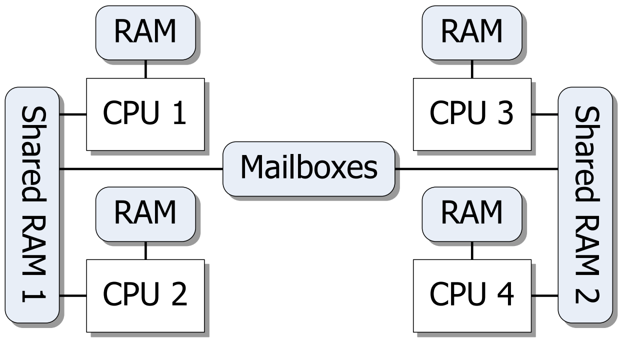

Figure 5.24: The experimental architecture

5.3.1 Clustering experiments

The experimental architecture (figure 5.24) is a non-standard four-core system implemented upon a Xilinx XC4VLX25 Virtex 4 FPGA on the Xilinx ML401 prototyping board and uses the Microblaze soft-processor. The cores are arranged as two pairs, and each pair has their own bank of shared memory. The pairs can communicate using a dedicated communications buffer.

Consider that in the presented system threads can freely migrate between all four cores. When this happens it is preferable for data to move along with the threads that use it, because the communication link between the two pairs is based on mailboxes and is relatively slow. Therefore, if a thread moves from CPU1 to CPU3 but its local data does not, the central link will become a bottleneck resulting in poor system performance. Without both the clustering model and the architectural model provided by CTV this requirement cannot be expressed in existing programming systems through the use of affinities alone.

The AnvilADL description of this architecture is as follows:

```
//Hardware elements
maincode : cluster;
cpu1, cpu2, cpu3, cpu4 : processor;
mem1, mem2, mem3, mem4 : memory;
leftshared, rightshared : memory;
cpu1^memory = [mem1, leftshared];
cpu2^memory = [mem2, leftshared];
cpu3^memory = [mem3, rightshared];
cpu4^memory = [mem4, rightshared];

//Communications
mbox : channel Mailbox;
m1to2, m3to4 : channel Mailbox;
mbox^endpoints = [cpu1(0x80000000), cpu2(0x80000000),
   cpu3(0x80000000), cpu4(0x80000000)];
m1to2^endpoints = [cpu1(0x80001000), cpu2(0x80001000)];
m3to4^endpoints = [cpu3(0x80001000), cpu4(0x80001000)];
//...and I/O channels for external comms not shown

//Logical layer
```

```
leftpair, rightpair : target;
leftpair^contains = [cpu1, cpu2, leftshared];
rightpair^contains = [cpu3, cpu4, rightshared];

anapplication : cluster;
//Add elements from the source to the cluster by setting
//anapplication^contains
anapplication^targets = [leftpair, rightpair];
```

In this description, each pair of processors is grouped together as a cluster target, and a single cluster is defined. Any source level items (threads, data, etc.) that are assigned to that cluster will be implemented either entirely on the left-hand pair or entirely on the right-hand pair. They will not be assigned separately.

A range of single and multi-threaded benchmarks were run on the target system. Each program was executed in three different modes and the resulting execution times are shown in figure 5.25. In this figure, all results are normalised against the *local* execution time, which is the time taken when running with its data in local shared memory (i.e. it has not migrated at all). *Migrated* shows the same program, but when one of its threads has been migrated to the other CPU pair and must fetch its data remotely. *Clusters* shows the cost of executing the application with cluster-aware migration. This includes the initial cost of migrating the application's data and then its subsequent execution time.

Note that for these tests, thread migration is simulated because Anvil does not yet implement it. The thread is compiled for each target processor that it may migrate to (according to the logical mappings in the AnvilADL) and during a migration the source thread is simply terminated and its clone created on the target processor.

In order to ensure that the simulation results are realistic, it is necessary to measure the amount of data required to describe the state of the current thread. Every thread migration, an amount of dummy data equal to this state is transferred from source to target.

Because this simulation does not use heap data (data that is allocated with `malloc` and `free`) thread state consists of the contents of the stack and the contents of the processor registers [3]. The Microblaze has 32 general-purpose registers and up to eighteen special registers (depending on configured options) resulting in 50 words of state. The size of the stack is determined by fetching the current stack pointer and comparing it to the base stack pointer. For applications that require Xilkernel, the `pthread_attr_getstack` function is used to obtain a worst-case guarantee of the current stack size.

The results show that clustering allows threads to migrate throughout a heterogeneous architecture without risking incurring sizable performance penalties. Most programs complete in between 1.1 and 2.8 times the execution time, as compared to between 8.5 and 37.9 times for systems in which the thread has migrated but not its associated data.

One outlier is the binary search benchmark. This system performs slightly worse when using full cluster than not. The reason for this is that the search only reads on average $log\ n$ data items (where n is the input array length) yet migrating the entire array requires n accesses. Whilst individual random accesses are much slower than the burst transfer used migrating a

[3]If heap data was supported it would not affect the results, as the same amount of state needs to be transferred but it is fetched from a different location.

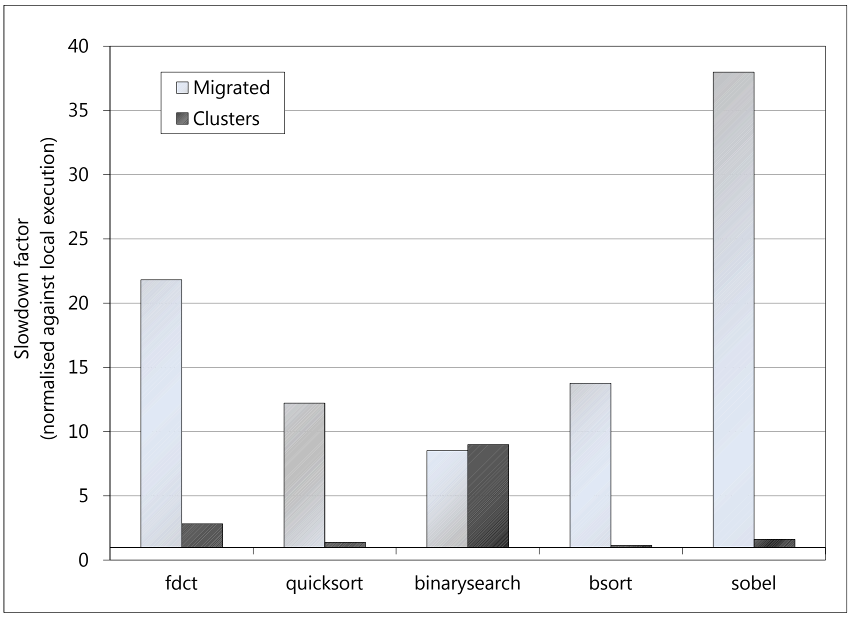

Figure 5.25: Resultant slowdowns experienced after a single migration. 'Migrated' shows results if only the thread is migrated. 'Clusters' shows results for migrating the thread and its data. All times include the time taken for one migration followed by one execution.

cluster, the execution time is slightly faster overall. This indicates that whilst cluster-based migration appears a good choice for most benchmark programs, care should be taken if the program only accesses a small number of random elements from a large data source. The converse of this is the Sobel filtering example. Because the Sobel algorithm frequents the same pixel values multiple times, full cluster migration is vastly more efficient.

5.3.2 Clustering simulations

This section presents results gained from a simulator that was developed to explore the effect of cluster-based migration on large grid-based systems. The architecture simulated is a regular grid of processing nodes, where each node has its own local memory. There is no global shared memory and caches are not modelled, so the simulation architecture is much closer to a large embedded architecture than the type of systems found in high-performance computing. The simulation framework creates a random set of tasks and assigns them to a variable number of clusters in the system. It also creates a random amount of shared data items and adds these to the clusters. The following task model is used:

The simulation (t, d, c, p) is composed of:

- Tasks
- Shared data items

- Clusters
- Processing nodes

A task t has:

- An execution time
- A memory access pattern that is a set of tuples (d, f) in which d is a shared data item and f is the probability that on any given clock cycle the task will access d.
- The current processing node to which it is assigned.
- A worst-case state size. The largest amount of state in bytes that must be transferred to migrate the state of this task.

A shared data item d has:

- A size
- The current processing node to which it is assigned.

Clusters (c) are represented as a set of tasks and shared data items. Each processing node (p) has a likelihood of being shut down at any given instruction. When a node is shut down, any tasks and data items that are on that node must be migrated elsewhere. The simulation assumes that all processing nodes share a common clock. Communications are modelled assuming a static routing system similar to the CTV communications layer (section 4.7). Accordingly, transfer costs are measured as a function of the number of hops they require. Link congestion is not considered in this model.

Task sets are generated as follows. In all cases unless otherwise stated, values were obtained by analysing the characteristics of the embedded benchmarks used in figure 5.25 and other example programs written for section 5.1. As a result, the values tend to represent smaller programs more than large monolithic systems (like a kernel or similar). However this section demonstrates that as programs grow in size and task length the importance of clustering increases, so these results are more likely to be conservative.

- The number of clusters is currently uniformly selected from 1 to 5. Due to the fact that the clustering model is not currently used for embedded development, it is not possible to determine suitable values from existing systems. Accordingly, these values are chosen as they are reasonable estimates of the number of ‘jobs’ performed by smaller embedded systems at any one time. Larger systems may perform more tasks at once, but as stated above, the focus of this simulation is on smaller-scale embedded systems.
- The number of tasks in a cluster is uniformly selected from 1 to 10, which was selected from experience as reasonable values for the amount of parallelism inherent in a given problem. Embarrassingly parallel problems can of course contain a much larger amount of tasks when implemented on general-purpose computers, but ten is reasonable for embedded implementations because each task imposes a state footprint in memory so the number will be kept rather low.

Item	**Time (cycles)**
Random memory access cost	31
Initiate burst transfer	31
Cost per word after initiation of burst transfer	1
Communication cost per hop	85

Figure 5.26: Timings used in the clustering simulation.

- The number of shared data items in a cluster is uniformly selected from 1 to 10.
- Shared data accesses are set at between every 10 operations to every 300 operations. This hides the fact that in reality memory access occurs in bursts, but because this simulation does not consider congestion this does not affect the results.
- Data items are sized from between 4 bytes and 1024 bytes, and are selected uniformly.
- Task state size and execution time is selected to be between 10,000 and 1,000,000 cycles and selected uniformly. The effect of varying task size is examined specifically later.
- Migration chance is more difficult to select representative values for, as such dynamic systems are currently rare in the embedded domain. Accordingly, the base migration rate is set so that on average ten tasks can complete their execution per single migration, which ensures that migration is of a realistic order of magnitude. The effect of varying migration chance is examined specifically later.

The simulation runs in two modes. In clustering mode the simulation uses the clustering information provided by the c set to migrate the entire cluster (tasks and data) when any of its constituent nodes are shut down. In undirected mode the simulation ignores the c set and only migrates tasks and data when their host node is shut down. Memory access times and inter-node communication times are taken from experimental results on real softcore-based FPGA systems to ensure that they are of appropriate relative sizes. These times are shown in figure 5.26. Access times on non-FPGA systems tend to be at least an order of magnitude greater due to the heightened discrepancy between processor clock speed and memory clock speed, so this would only increase the importance of fast memory accesses.

Figures 5.27 and 5.28 show the effect of varying task execution time on grids of different sizes. In these experiments the probability of migrating tasks from a processing node is fixed at on average one migration every 100,000 clock cycles. This is deliberately very frequent - real systems are likely to migrate much less and therefore demonstrate lower overheads. As can be seen, on the smallest architecture (2x2) migrating the entire cluster appears on balance less efficient as the costs of migrations are considerably higher and do not give any real benefit due to the small size of the system and rapid migrations. However, on all larger architectures migrating whole clusters results in systems that display lower memory access costs. The tests for the 4x4 architecture demonstrate clearly that for tasks with short execution times the tasks are only likely to be migrated a small number of times so the large transfer cost of migrating an entire island is not beneficial. Longer tasks, however, rapidly become less and less efficient as they migrate further from their data. This is demonstrated most noticeably by the results for the 8x8 and 16x16 architectures which are in some cases are 6.5 times slower without clustering.

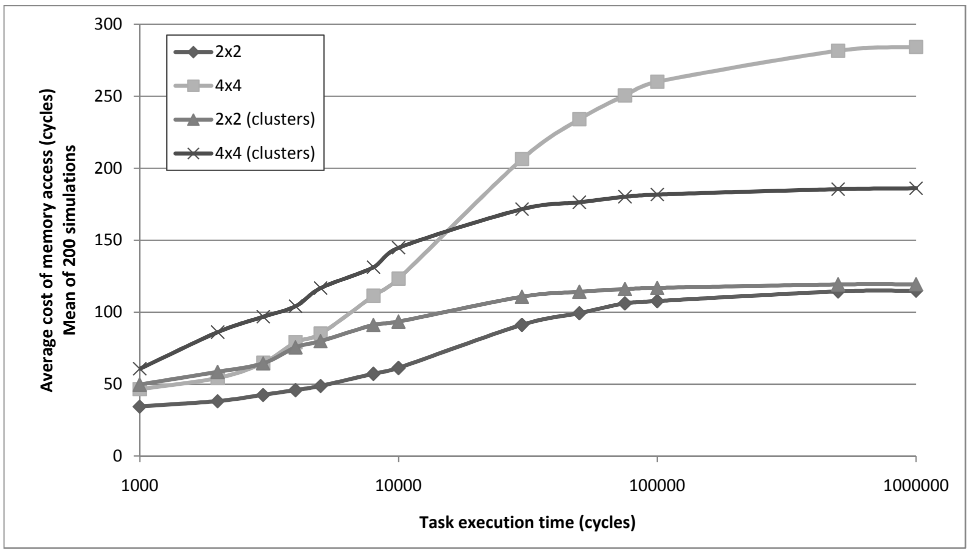

Figure 5.27: Varying task execution time with and without cluster-based migration on smaller architectures

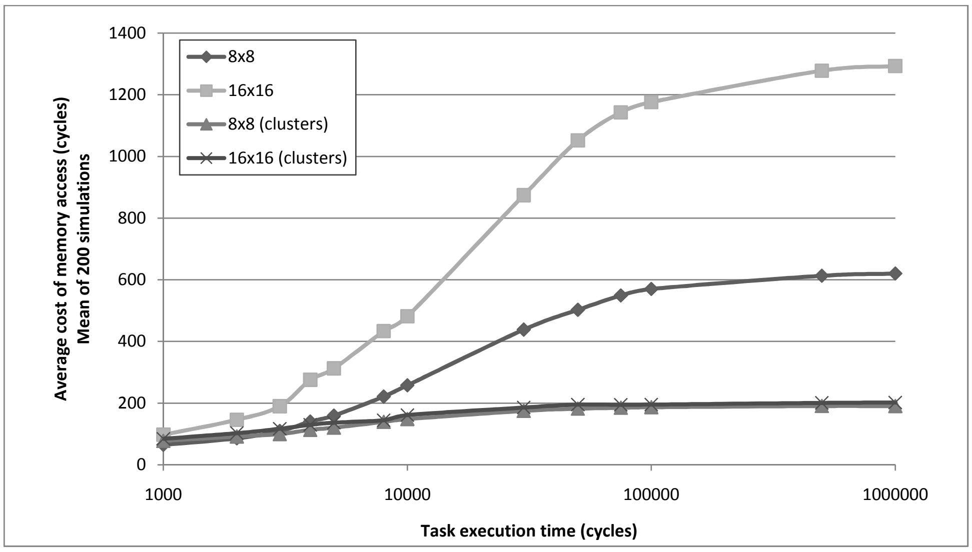

Figure 5.28: Varying task execution time with and without cluster-based migration on larger architectures

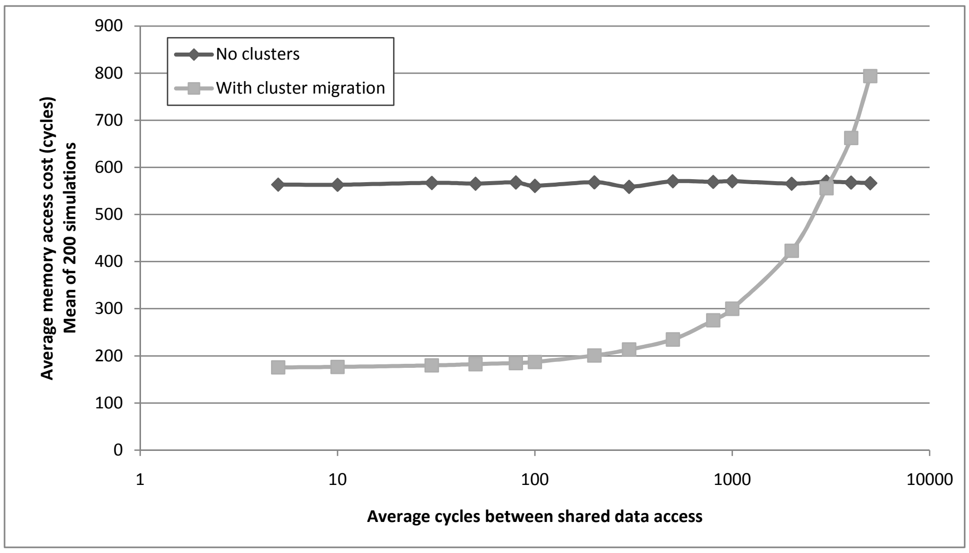

Figure 5.29: Varying task migration chance with and without cluster-based migration (8x8 grid)

Another interesting observation is that when using clustering, the results for the 8x8 architecture are almost identical to the results for the 16x16 architecture. The reason for this is that clustering guarantees that locality will be preserved. If the architecture grows in size it does not affect intra-cluster communications, and the coupling of the system is preserved. Communications between clusters will on average be slower, but these are much less common and involve smaller amounts of data. This demonstrates that a system that uses clustering and coupling displays greater scalability than one that does not.

Figure 5.29 shows the effect of varying the task migration chance. When migrations are unrealistically frequent (around every 200 cycles) the cost of full cluster migration is much higher than migrating single items. For more realistic systems, cluster-based migration results in lower average memory access times by a factor of 3 in some cases. As the chance of migration becomes smaller, access times for systems with clustering and systems without clustering converge. This is because it becomes increasingly unlikely that a task will experience a migration at all, thereby negating the difference between the two approaches.

Finally, figure 5.30 shows the effect of varying the frequency with which tasks access their shared data items. When cluster-based migration is not being used, this has no effect on overall system performance because the migration mechanism does not consider which data items are accessed by which tasks. As can be seen, if tasks only access their shared data infrequently (of a similar order of magnitude to the chance of migration) then migrating whole clusters displays no benefit whilst increasing the average cost. However, more frequent access results in much faster access times. This indicates that cluster-based migration is beneficial when shared data is accessed at a frequency which is at least an order of magnitude higher than the average inter-migration time. This is very likely to be the case for all realistic systems, but should nonetheless be considered.

These simulation results indicate that the use of clustering information in the CTV system model has the potential to greatly increase the efficiency of embedded systems that make

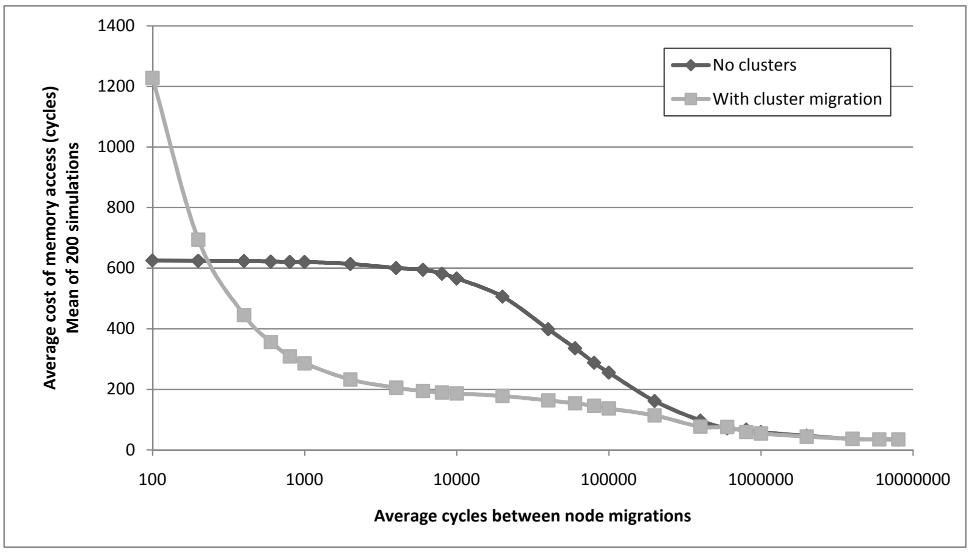

Figure 5.30: Varying how frequently tasks access shared data (8x8 grid)

use of thread migration. If shared data is accessed very infrequently, migrations are incredibly common, or the architecture is very small the costs can outweigh any benefits. However, on the majority of medium to large embedded systems with dynamism, the technique can be used to deliver sizable performance benefits.

5.3.3 Migration times

The previous section used simulation results to demonstrate the potential for cluster-based migration supported by the CTV system model to increase the overall efficiency of dynamic systems. This section focusses on Anvil's support for OM and shared data migration, and the efficiency of the underlying communications layer. As detailed in sections 4.13.1 and 4.13.2, Anvil currently supports the migration of OMs and of shared data items, whilst thread migration is supported by the CTV system model and Anvil communication layers but is not yet implemented.

Recall from section 4.13.1 that in the Anvil implementation a migrating OM migrates control of its managed items to a waiting OM on the target processor. This requires the transferal of the state of the object and any queued requests. Once migrated, the source OM must update the communications layer so that future requests are sent to the target OM. This section details the costs of the managed object migration whilst the following section considers the cost of updating the communications layer.

Section 5.2.3 detailed the size in bytes of the stored state for managed threads, mutexes, CVs and shared data items so these are not repeated here. The data that must be transferred during a migration is therefore:

Item	Accessors	State size	Comms. medium	Migration time (cycles)	Time (at 100Mhz)
Mutex	2	11 bytes	Mailbox	1094	11 μs
Mutex	2	11 bytes	UART	1304	13 μs
Mutex	4	19 bytes	Mailbox	1418	14 μs
CV	2	10 bytes	Mailbox	1110	11 μs
Thread	4	20 bytes	Mailbox	1436	14 μs
Shared variable	2	120 bytes	Mailbox	5610	56 μs
Shared variable	2	120 bytes	UART	6662	67 μs
Shared variable	2	240 bytes	UART	13104	131 μs

Figure 5.31: Time taken to migrate managed objects over a range of architectures. ‘Accessors’ is the number of other objects in the system that access the migrating object.

Item	Size
Managed thread	The thread’s state
Managed mutex	The mutex’s state
Managed CV	The CV’s state
Managed shared variable	The shared variable’s state (plus the size in bytes of the shared variable *if* the target processor does not have direct access to the memory space that currently contains the variable)
OM	The sum of all the objects that it manages

As detailed in section 4.13, a managed object is migrated with the `M_MIGRATE` message. The transmission time of such messages varies considerably depending on target architecture and the topology of the application. Handling these messages does not involve the manipulation of complex data structures so they can be dealt with very quickly. The Anvil OM manages an array of the items it is currently managing and simply adds the incoming migrating item to the end.

To give an indication of migration costs, figure 5.31 shows the migration times for a range of objects over different communications channels. These costs all assume that the migrating object’s state is in block RAM or cached memory so it can be accessed quickly. Given its small size and frequent access pattern this assumption is reasonable. If the state is in external RAM then access times will be accordingly slower.

Migration times are relatively small. At 100Mhz, migration takes between 11 and 15 microseconds for mutexes, threads and CVs, and scales linearly for larger shared variables. Note that the results in figure 5.31 do not include the time taken to update the communications layer, detailed in the following section.

Although not strictly comparable, these times are much lower than existing thread migration systems such as JESSICA2 [269], a Java-based system in which the presented migrations take between 3838 microseconds and 242012 microseconds at 540MHz. At similar clock speeds, distributed systems such as Ariadne measure migration times in 10,000s to 100,000s of microseconds [161]. Neither of these systems are designed for low-overhead embedded architectures so the comparison is not entirely accurate, but it does serve to illustrate that

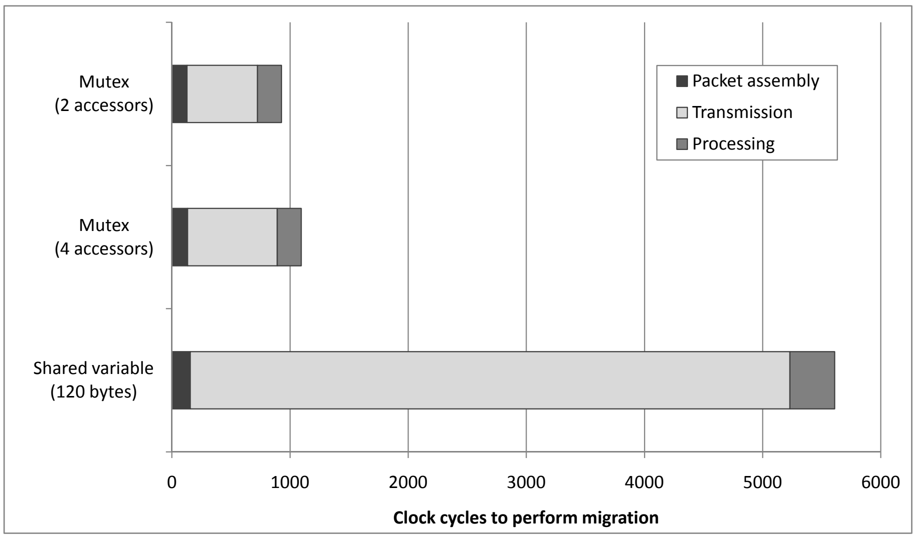

Figure 5.32: Total migration times separated into processing and transmission times.

Anvil's lightweight, less fully-featured approach that requires extra programmer input (through the AnvilADL description) allows for migration times that are much lower than transparent, yet more heavyweight solutions.

Figure 5.32 further examines the results for three of the migrations in this table. The results are similar to those obtained in section 5.2.4 (which discussed transfer efficiency of the shared memory system) because the underlying transfer mechanism is the same. For the mutex with two accessors, 57% of the total migration time is consumed waiting for the communications medium to transmit the actual state information from source to target. For the mutex with four accessors, more state needs to be transferred and this rises to 64%. For the 120 byte shared variable, over 90% of the migration time is consumed by transmission time alone. This shows that the main limiting factor for migration times in the Anvil system is the speed of the underlying transmission medium and that the overheads imposed by the OM system are therefore acceptable.

Figure 5.31 lists the number of other objects in the system that access the migrating object because, as detailed in section 5.2.3, the state of a managed object in Anvil is proportional in size to the maximum number of requests that can be queued upon it. Anvil currently guarantees that the worst-case situation in which all requests are placed simultaneously can still be handled (by setting `_ANVIL_REQUEST_QUEUE_SIZE` to the number of other objects that access this object). This is pessimistic because as systems grow this forces the state size of managed objects to grow also increasing migration times accordingly.

To account for larger systems, the implementation can use clustering information from the Anvil system model. An object is much more likely to be accessed by its siblings (items in the same cluster) than items from other clusters. A better implementation can statically-allocate state space for siblings but dynamically-assign space for extra-cluster objects, thereby placing an upper limit on state size. This is much better suited for large systems, but does include the requirement that if, at a critical instant, the request queue fills up then the requester must be able to back off and try again later similar to CSMA/CD in broadcast networks, decreasing the implementation's predictability.

5.3.4 Updating the communications layer

After an object has migrated, the communications layer must be updated so that each transmitter is aware of the new location of the node. The algorithms for doing this are covered in section 3.7.1. Anvil implements the location propagation algorithm, in which after a migration all nodes are informed of the new location. Accordingly, the worst-case cost c to update the communications layer after migrating an object from processor x is termed c_x and given as:

$$c_x = \sum_{p \in P}^{p} comms(x, p)$$

where P is the set of processors in the system and $comms(a, b)$ is the cost of sending a communications update message from a to b. This is the worst-case cost, because in architectures that contain broadcast communication channels the update can be sent to multiple processors simultaneously.

This compares favourably with Ethernet-based systems which are designed to cope with a greater amount of run-time variability. In Ethernet-based systems, when a host moves none of its peers are directly updated. Instead, when one of those peers attempts to communicate with the migrated host the connection times out, which triggers a broadcast message from the peer to determine the host's new location. This is more stable for large, unreliable networks with a large number of migrations, but does not provide very tight bounds on communication delays which can cause problems when developing real-time systems.

For the size of systems that Anvil has currently been used on this overhead is small. The size of the position update message (`M_POSITION_UPDATE`) is only eight bytes, so for Microblaze-based FPGA systems transmission times are approximately 600 clock cycles per processor, depending on exact communication topology. Therefore, for a modest five-core system, a migration requires at most four short messages transferring 32 bytes of information in total. This is much less than the overhead associated with reading even a modestly-sized shared variable.

Still, for larger systems it is possible to use clustering information to avoid updating all processors after each migration. Described in section 4.7.2, the intermediary manager algorithm allocates a manager for each cluster that is responsible for handling requests for objects outside the cluster. Objects outside the cluster are not updated with the current position of migratable elements inside the cluster. Instead, they forward the request through the intermediary manager meaning that `M_POSITION_UPDATE` messages need to only be sent to the siblings of a migrating object, reducing the cost further and ensuring scalability.

The cost for migrating an object under this scheme becomes:

$$c_x = \sum_{p \in C}^{p} comms(x, p)$$

where C is the cluster target that contains x.

5.3.5 Summary

In summary, evaluation of the clustering model through both experiments on actual hardware and simulations has shown that it can be instrumental in supporting future architectures with a large number of processing cores. The model has particularly been shown to be beneficial for supporting systems that use thread and data migration.

Simulations identified that the clustering model is not always appropriate, and in extreme cases its extra migration cost (migrating all threads and data of a cluster rather than only individual items) can result in a less efficient system. Cluster-based migration is unsuitable if the target architecture is very small, shared memory accesses are very rare, or migrations are incredibly frequent. However, in cases that describe more typical scenarios, cluster-based migration can result in greatly improved memory access times that scale well to larger architectures.

5.4 Conclusion

This chapter has evaluated the effectiveness of CTV-based approaches for targeting complex, non-standard architectures. The CTV system model has been shown to be sufficiently expressive to allow the representation of modern embedded architectures and the software that executes on them. A range of theoretical and actual architectures were demonstrated with a range of different applications executing on them. CTV has also been shown to be able to leverage its compile-time nature to reduce overheads in many key areas of the final implementation whilst still offering the developer the programming model that they are used to.

It was identified that due to Anvil's chosen source language of C with pthreads, Anvil cannot effectively target vector processors, such as DSP cores or GPUs. Anvil must be extended before such architectures can be used, either with a compiler than can extract vector operations automatically or by adding support for a language extension that allows the expression of vector operations (like OpenCL or OpenMP).

Anvil has been shown to impose very small overheads that are in some cases orders of magnitude less than equivalent run-time systems. This is because such run-time systems often provide a much greater range of functionality than is required at each processor of the target system. Anvil's OMs only need to provide the specific services used by the threads of the system, and so can use their compile-time information to reduce their run-time impact accordingly. Equally, architecture information is used to hard code as much of the low-level drivers as possible, leading to efficient library implementations. In the worst case, if the source application is highly dynamic then CTV-based techniques will impose no extra overhead than run-time systems as they can choose to implement the run-time system in its entirety.

This section has shown that CTV cannot completely hide from the application developer that they are writing software for an embedded architecture. Although the CTV VP presents a standard programming model and Anvil's refactoring engine can make any code that is within its model execute correctly, experiments have shown that the shared memory system in particular can show very poor performance on certain code examples. The programmer must be aware when using shared data that the data will flow from a remote source, so they should code their algorithm to support bulk transfers of data and access data sequentially in a predictable manner. If this is done then the shared memory system can approach optimality, but if it is not then highly inefficient behaviour can be created. It was identified that IDE support could greatly assist with this.

Chapter 6

Conclusions

This work has introduced Compile-Time Virtualisation, a technique for assisting the development of embedded software for deployment on complex, non-standard architectures. The hypothesis (section 1.5) stated that:

> Abstraction and virtualisation have been shown to be useful techniques for hiding implementation complexity and providing a high-level programming model to aid software development. However, they introduce significant overheads and the programmer cannot influence the mapping of their application when targeting non-standard architectures. This thesis contends that moving the virtualisation layer from run-time to compile-time will allow the programmer more control over the implementation of the system, resulting in support for a much wider range of target architectures, the exploitation of unique hardware features, and lower run-time overheads.

This hypothesis has been demonstrated throughout the thesis. Chapter 3 described the manner in which a virtualisation system can be moved from run-time to compile-time, and highlighted that the technique is language and implementation fabric agnostic. Chapter 4 showed a method by which this design can be implemented in practice.

Experiments and simulations described in chapter 5 demonstrated the benefits of CTV over existing run-time virtualisation systems. Section 5.1 describes the additional control afforded to the programmer over the implementation of their software, and details the range of target architectures that a compile-time system is capable of supporting. Section 5.2 details the overheads in the CTV implementation and describes the areas in which it displays lower overheads than existing run-time systems. Sections 3.2, 3.7 and the evaluation in section 5.3 argue about the scalability benefits that are demonstrated by the CTV approach, and show that CTV avoids inherent implementation bottlenecks. The scalability of the final implementation is only limited by that of the input programming language.

Section 6.1 summarises the findings of the work and section 6.2 discusses possible future developments from the work introduced by this thesis. Finally, section 6.3 concludes.

6.1 Summary of findings

As discussed in general in chapter 2 and specifically in section 2.6.1, the programming models of high-level languages have traditionally attempted to abstract away architectural information. The reason for this is that early computing architectures were fixed so the programmer did not need to be aware of the specifics of the implementation toolchain. However, as architectures become increasingly heterogeneous and complex, the programmer's inability to reason about the architecture leads to two main problems:

Inflexibility Because the architecture is hidden by the abstraction layers of the programming model, a program cannot use unique architectural features (such as function accelerators, DSP cores, I/O, vector processors etc.) without inserting architecture-specific, abstraction-breaking code. Non-uniform memory architectures require either the use of a run-time shared memory system (section 3.6.2) or direct manipulation of the linker (exemplified in section 2.6.1).

Inefficiency The programming models of many existing languages assume a single-processor (or uniform SMP) system with contiguous shared memory. As systems deviate from this, layers are inserted to hide discrepancies. These layers may take the form of hardware (such as memory management units) or run-time operating systems and middleware layers. These all add inefficiency to the final implementation.

Increasing flexibility

This thesis demonstrates that CTV helps to solve the 'inflexibility' problem by replacing the existing layers of abstraction with a virtualisation layer that creates a *Virtual Platform* (VP). The VP is an idealised view of the underlying hardware designed to present a virtual architecture that is compatible with the programming model of the source language. The features that the VP must expose are discussed in section 3.6.2.

VP-based development is very flexible because the architectural mappings that it implements can be influenced by the programmer. This is unlike the assumptions of the source language which are fixed by the language definition, and unlike the behaviour of compilers which can only be changed by creating a new compiler and associated toolchain. The VP mappings can be influenced by the programmer to effectively exploit the target architecture, making use of any unique hardware features as appropriate. The virtualisation ensures that the specified mapping will execute correctly.

Allowing the programmer to influence the implementation toolchain in this way is shown to allow the capabilities of a pre-existing language and compiler to be extended in a controlled way to meet changing future demands. This aids code reuse, because existing code is not made obsolete when a new architectural paradigm is developed that would previously have required a new language or language extension.

The CTV approach is orthogonal to automatic parallelisation, co-design, or automated system synthesis. Such systems are designed to assist the programmer with the development of systems, either by creating hardware from analysis of the input software, or by altering the software specification to increase its potential for parallelism, etc. These techniques can be used alongside CTV to mutual benefit. For example a VP could be used to assist co-design frameworks. By insulating input software from hardware changes, the CTV VP could allow greater flexibility in evaluating new candidate architectures. Equally, auto-parallelising compilers assist CTV by ensuring that the input program has sufficient parallelism to allow for efficient distribution over a multi-core architecture.

Increasing efficiency

The second problem of 'inefficiency' is solved by moving the virtualisation of the VP to compile-time rather than run-time. The thesis demonstrates in section 3.5.1 that such a move trades run-time flexibility for implementation efficiency. In a Compile-Time Virtualisation (CTV) system, because the structure of the program and architecture must be known by the compiler the run-time behaviour of the application is slightly restricted. These limitations are detailed section 3.6.1, and exemplified in section 4.15.

The benefit is that for the majority of embedded applications that do not require much run-time dynamism, very low overhead solutions can be created. Section 5.2 demonstrated that the level of information available to the compiler in a CTV-based system allows for static sections of the system to vastly reduce the amount of run-time support that they require, reducing memory and processing requirements.

The thesis introduces the Object Manager (OM) model, which is a model for the provision of distributed OS services in a non-uniform embedded system. The OM model uses the extra compile-time information provided by CTV to implement a low-overhead system that provides distributed shared memory, coordination, concurrency and migration without imposing bottlenecks in the way that a traditional OS kernel does. The OS model is shown to support the CTV system model to distribute applications over complex architectures.

In section 2.6.3, the thesis observes that many existing programming models demonstrate poor scalability by assuming universal shared memory with globally-coherent caches and a totally-connected grid of processing nodes. This does not reflect the structure of actual applications, in which each shared data item tends to only be accessed by a small subset of the system threads. Also, these models do not link threads with the data that they use, leading to poor support for systems with dynamic migration.

Accordingly, a clustering-based system model is introduced in section 3.2 which defines the concept of *coupling*. Coupling allows the programmer to specify the extent to which elements of their application interact. Tightly-coupled elements interact frequently and are more time-critical. The clustering model is shown to allow the implementation to relax some of the coherency and communication assumptions of the programming model and to concentrate on only tightly-coupled items. Section 5.3 shows the benefits that this has for large systems, or systems that support thread and data migration.

The VP and clustering model are both part of an overall system model used by this thesis, introduced in section 3.4 and refined for compile-time use by CTV in section 3.6. This model is shown to be suitable for representing three languages commonly used in embedded development (C, Ada and Java) in section 3.4.5. The model is shown to be expressive enough to allow CTV to target a wide range of architectures in section 5.1.

The decentralised and inherently parallel nature of the OM model allows for the VP to provide an implementation that will scale to large systems as much as the input program allows it to. Clearly if the input program is structured as a large number of tasks which all depend on the same mutexes and shared data then scalability is restricted and the resulting implementation will be similar to a standard distributed operating system. However, as programmers (and programming languages) increasingly consider parallelisation and complex architectures, CTV can take advantage of this to provide more scalable and efficient implementations. The bottlenecks are placed back on the specification of the programming model, and onto the programmer themselves.

Finally, the thesis presents Anvil, an implementation of CTV based on the C programming language. Anvil is compared to existing run-time systems and demonstrates very low overheads compared to existing run-time approaches.

6.2 Future work

There are a number of areas of future development based on the work in this thesis.

6.2.1 More expressive source languages

As highlighted in chapter 5, Anvil currently cannot effectively target vector processors because its chosen input language of C with pthreads does not allow the programmer to describe data-parallel and SIMD-style operations. Also, in order to support dynamic data structures over a NUMA, Anvil required extra compositional information from the programmer in the form of Anvil POs (section 4.12).

As emphasised in this thesis, a key advantage of CTV-based systems over run-time systems is that because CTV's virtualisation layer exists at compile-time it can take advantage of the high-level expressive capabilities of the source language. The mapping capabilities of CTV-based systems increase with the expressive power of the input language, whereas a run-time system only has access to compiled machine instructions of individual target processors. Higher-level semantics and inter-thread interactions are lost.

Therefore, an interesting avenue of research involves looking at applying CTV to a language with greater expressive power than that of C. The following areas are of particular interest:

OpenMP As mentioned in section 5.1.7, OpenMP allows the programmer to express both coarse-grained and fine-grained parallelism. This would allow a CTV system to make more efficient use of SIMD-based processing architectures.

Occam Languages that are based on CSP and similar computation models provide explicit enumeration of the communication channels in a program and how they are used. This greatly assists mapping to the communication channels of embedded architectures.

Hardware-description languages See below.

6.2.2 CTV and hardware development

In CTV as presented in this thesis, the target architecture is provided through a hardware description in terms of the CTV system model. An alternative to this is that CTV could be used to assist with the development and refinement of the target architecture. There are three main ways in which this can be done:

Co-design CTV can be integrated into an existing co-design framework that designs and refines the target architecture instead. A common difficulty experienced by co-design is that the application software needs to adapt to the iterating architecture. Currently, this is done by limiting memory access, communications and I/O, and providing a run-time wrapper layer that the software must use. CTV could instead be used to automatically refactor an architecturally-neutral program so that it executes efficiently on the various candidate architectures as they are evaluated by the framework. This would ease the software development of co-design systems, and allow more flexible architectural refinement, in excess of the standard co-design architecture that most systems mandate (described in section 2.4.4).

High-level synthesis An interesting potential area of research is the integration of high-level synthesis into the CTV system model. Currently, CTV models custom hardware elements as part of the target architecture and shared objects as part of the input application. However, the only practical difference between these two concepts is the manner in which they are implemented. If the input language is a high-level synthesis language (section 2.4.2) such as Handel-C or Catapult-C (or if a synthesising compiler is used) then the CTV implementation could allow the programmer to change between hardware and software implementations automatically. CTV's communications layer would automatically ensure that messages are correctly routed. Similar work has already been done in the JEOPARD project [199] in order to add hardware-implemented object methods to the Java programming language, so results and techniques from this project would be useful.

Run-time reconfiguration Currently the CTV system model targets a fixed architecture, but FPGA-based systems can make use of run-time reconfiguration to dynamically change the target hardware as the system is executing (see section 2.5.6). CTV's communication and migration facilities could be used to provide support for this reconfiguration.

The unifying trend of these three research directions is that CTV frees the architecture designer from the limitations imposed by the source language. CTV's VP is guaranteed to expose a compatible programming model that will ensure correct functional operation of the input application. This allows iterations and improvements to be made to the target architecture independently of the input software.

6.2.3 Hierarchical Virtual Platforms

Moving virtualisation from run-time to compile-time allows for presence of many virtualisation layers in the same system without imposing extra run-time cost. Section 5.2 showed how compile-time virtual layers collapse during the compilation process to impose the minimum

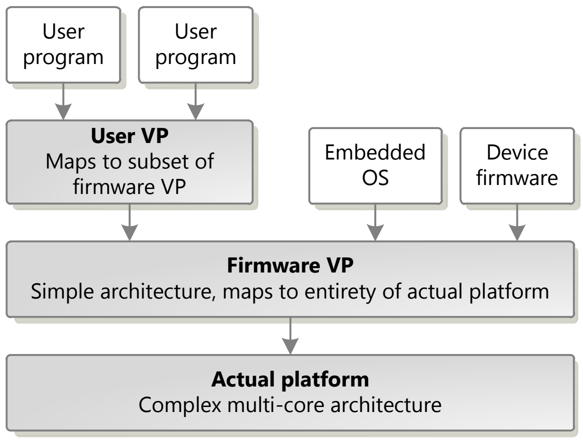
User program
User program
User VP
Maps to subset of firmware VP
Embedded OS
Device firmware
Firmware VP
Simple architecture, maps to entirety of actual platform
Actual platform
Complex multi-core architecture

Figure 6.1: Hierarchical VPs allow for different views of the actual platform to be presented to different classes of program.

level of run-time overhead required. This property will remain if multiple nested or hierarchical VPs are used in the same system.

Accordingly, the situation depicted in figure 6.1 is possible. The figure shows the complex architecture of a mobile computing device, such as a smartphone. A VP is present to hide the underlying complexity of the architecture and assist the development of firmware and operating system software. However, user applications (which might be from an external source and therefore are untrusted) exist on top of a further VP. This VP hides a greater amount of the architecture so only the application processor is visible and the real-time tasks of the baseband processor cannot be affected.

Hierarchical VPs could be used for the following purposes:

- **Simplified development:** Applications that do not need access to the full architecture can be presented with a highly-simplified (virtual) target architecture.
- **Sandboxing**: Sandbox VPs can prevent access to real-time areas of the hardware, or guard against the behaviour of malicious software. Also, because the sandbox VP has access to the entire source code of the sandboxed software, it can actually *change* its behaviour rather than simply prevent certain actions. For example, a sandbox VP can alter its hosted applications so that all memory accesses are directed through a given communication link, therefore freeing other links for other potentially real-time tasks.

6.2.4 Implications on WCET analysis

When developing hard real-time systems, it is necessary to perform Worst-Case Execution Time (WCET) analysis to ensure that the system will meet its deadlines. One of the main challenges associated with WCET analysis is the reduction of analysis pessimism, which results from uncertainty in the analysis. If analysis is overly pessimistic then a system that uses it will be safe, but will be very wasteful due to over-provision.

The following areas of further research are areas in which CTV can potentially reduce analysis pessimism or improve the WCET of hard real-time systems:

- CTV reduces the layers of run-time middleware support required. For a static architecture CTV has a very predictable performance because the communications and shared memory libraries are hard-coded for each target processor.
- The clustering model, and the provision of distributed OS services by the OM model, helps to make the system more scalable by bounding the interference experienced by a thread from other clusters in the system. Interactions still have to be modelled by WCET analysis, but the overall pessimism is likely to be reduced.
- Commonly, some parts of a target architecture will be fast (complex processors with caches and dynamic memory), and some parts will be slower but predictable (short pipelines, no caching, static memory). The extra mapping information provided by CTV's system model allows for time-critical code to be appropriately mapped to predictable areas of an otherwise heterogenous architecture, thereby assisting WCET analysis on a wider range of architectures.

6.2.5 Anvil improvements

There are a number of areas in which Anvil's implementation can be improved. These are:

- Implementation of modern cache coherency algorithms to improve the performance of the shared memory system in SMP-style architectures.
- Support for true heterogeneous thread migration (an algorithm similar to the work in [85] is appropriate). Anvil currently does not migrate threads, only shared data, shared objects and OMs.
- Optimisation of OM code and the refactoring engine to reduce Anvil's run-time code footprint and compile-time overheads. Anvil is a research project and so the code is more focussed on clarity and debugging capabilities than optimisation. The Anvil compiler and refactoring engine can be rewritten in a faster language.
- Support for a wider range of embedded processors.

These items were not important to develop the work contained within this thesis, but would help to allow further investigation into the efficacy of Anvil for use in industrial contexts.

6.3 Conclusion

This thesis has demonstrated how Compile-Time Virtualisation can aid the development of software for architecturally-complex embedded systems. The CTV system model is defined as a language-neutral model that can be used with a wide range of input languages and is expressive enough to target current and future embedded systems. CTV uses a compile-time Virtual Platform that is lightweight and introduces a very amount of low run-time overhead. It is still necessary to restrict the code used by the programmer in order to obtain efficient mapping, as CTV requires that the input code is largely amenable to static analysis. Use of an analysable language subset (for example MISRA-C [220]) would provide guidelines and coding styles to assist developers. An implementation of CTV based on the C programming language is presented and evaluated using actual implementations and simulations to show the efficiency and flexibility of the CTV approach.

Appendix A

Object Manager message protocol

Anvil defines an internal messaging protocol that is used by the OMs to communicate with each other. This protocol implements the various OM operations required by Anvil's implementation of CTV. Other implementations will have different requirements and so will need a different set of messages. These messages are conveyed by the communication channels of the target implementation and sent by the architecture layer of Anvil's libraries (sections 4.8, 4.9 and 4.10). In the current implementation, only 16-bit identifiers are used for thread and CPU IDs, limiting the system to 65,536 of both. A full production system for future massively-parallel systems may wish to increase this limit by using 32-bit identifiers. This protocol assumes that all threads and OM managers share an ID space and are all unique.

Also, messages are currently limited to 256 bytes in length, although in practice are always much shorter. This is because on-chip communications tend to take negligible time to establish a connection so the overhead of sending a batch of short messages is much less than would be the case over networks such as Ethernet or the internet. The advantage of short messages is that sending and receiving delays are shorter, latency is reduced, and they do not cause as lengthy pauses in processing on the source and destination nodes. These features can all be tuned by the implementation later, however. In a production system it is likely that average message length would be a factor of the communication medium.

Messages are currently not acknowledged because on-chip communications media are reliable. Clearly if an unreliable medium is used then acknowledges, flow control, and packet reordering will all need to be added to the protocol. Due to the fact that the CTV implementation knows offline whether a given medium is reliable or unreliable, it can switch between the two protocol styles depending on the link, resulting in lower overheads on better transmission media. These are all future implementation work.

The rest of this section details the messages used in the Anvil implementation. Each message is named, its structure detailed, and then notes given that describe interesting features of its use.

A.0.1 Thread operations

M_THREADCREATE

Source Thread ID	Length	Operation ID	Thread ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x01	ID of thread to create

Generated by a `pthread_create` call. This message should be sent to the target thread's OM which will then wake the appropriate thread. In the case where the thread's OM is not hosted on the same processor as the thread (or when the thread is on a logical processor and may therefore move at run-time, see section 3.4.1) the OM will repeat the message to the appropriate processor. This message should be idempotent.

M_THREADEXIT

Source Thread ID	Length	Operation ID
2 bytes	1 byte	2 bytes
	5	0x02

Sent from a thread which is exiting by calling `pthread_exit`, to the thread's OM. The manager notes that the thread is now exited, so that threads joining it can now be notified.

M_THREADJOIN

Source Thread ID	Length	Operation ID	Thread ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x03	ID of thread to join

Sent by a `pthread_join` call to the OM of the thread being joined. The OM will reply with a `M_THREADJOINREPLY` message. The OM will reply immediately if the requested thread is not currently executing, or it will queue join requests and reply later once the thread to join exits (with a `M_THREADEXIT` message).

M_THREADJOINREPLY

Source Thread ID	Length	Operation ID	Thread ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x05	ID of thread being joined

Sent as by the OM as a reply to a `M_THREADJOIN` message.

A.0.2 Mutex operations

M_MUTEXLOCK

Source Thread ID	Length	Operation ID	Mutex ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x10	ID of requested mutex

Sent by a `pthread_mutex_lock` call to the OM of the mutex being requested. The OM will maintain a queue of requests and grant them in FIFO order. The manager replies with a `M_MUTEXREPLY` message. The requesting thread must block until the `M_MUTEXREPLY` message is received.

M_MUTEXREPLY

Source OM ID	Length	Operation ID	Thread ID	Mutex ID
2 bytes	1 byte	1 byte	2 bytes	2 bytes
	8	0x11	ID of requesting thread	ID of mutex

Sent by the OM in reply to a `M_MUTEXLOCK` or `M_MUTEXTRYLOCK` to indicate that the named thread has obtained the mutex lock.

M_MUTEXTRYLOCK

Source Thread ID	Length	Operation ID	Mutex ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x12	ID of requested mutex

Similar to the `M_MUTEXLOCK` message, but the OM will always reply immediately with either a `M_MUTEXREPLY` if the lock is granted or a `M_MUTEXREPLYNEGATIVE` if the mutex is currently locked by another thread. The requesting thread must block until one of these messages is received, but not until the mutex lock is granted. The OM will not add an unsuccessful lock request to the lock queue, the thread must try again later.

M_MUTEXUNLOCK

Source Thread ID	Length	Operation ID	Mutex ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x13	ID of locked mutex

Send to the mutex's OM by a call to `pthread_mutex_unlock` to relinquish a mutex lock. This message should only be sent by threads that currently hold the mutex lock in question. Upon receipt of this message the OM will send a `M_MUTEXREPLY` to the next thread (if any) that is in the request queue for that mutex.

M_MUTEXREPLYNEGATIVE

Source OM ID	**Length**	**Operation ID**	**Thread ID**	**Mutex ID**
2 bytes	1 byte	1 byte	2 bytes	2 bytes
	8	0x14	ID of requesting thread	ID of mutex

Sent in response to a `M_MUTEXTRYLOCK` to indicate that the lock request was unsuccessful. The requesting thread is not added to the request queue.

A.0.3 Condition Variable operations

M_CVWAIT

Source Thread ID	**Length**	**Operation ID**	**CV ID**
2 bytes	1 byte	1 byte	2 bytes
	6	0x20	ID of CV

Sent by a call to `pthread_cond_wait`. The requesting thread is added to the CV's wait queue. The requesting thread should block until it is sent a `M_CVWAITREPLY`.

M_CVWAITREPLY

Source OM ID	**Length**	**Operation ID**	**Thread ID**	**CV ID**
2 bytes	1 byte	1 byte	2 bytes	2 bytes
	8	0x21	ID of released thread, or -1 to release all threads	ID of CV

Sent when a CV's OM receives either a `M_CVSIGNAL` or `M_CVBROADCAST`. If the CV is signalled, only one thread from the CV's wait queue is signalled. If the CV is broadcast then all threads are signalled by setting thread ID to -1.

M_CVSIGNAL

Source Thread ID	Length	Operation ID	CV ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x22	ID of CV

Sent from a `pthread_cond_signal` call to the OM of the CV. The OM will respond by waking a single thread from the CV's wait queue (if any) with a `M_CVWAITREPLY` message.

M_CVBROADCAST

Source Thread ID	Length	Operation ID	CV ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x23	ID of CV

Sent from a `pthread_cond_broadcast` call to the OM of the CV. The OM will respond by waking all threads from the CV's wait queue with a `M_CVWAITREPLY` message in which the target thread ID is set to -1.

A.0.4 Shared variable operations

M_SVREAD

Source Thread ID	Length	Operation ID	SV ID	Offset	RequestLength
2 bytes	1 byte	1 byte	2 bytes	4 bytes	1 bytes
	11	0x36	ID of SV	Byte offset into the SV	Bytes of the SV requested

Reads `RequestLength` bytes from the shared variable, starting at offset `Offset`. The OM of the shared variable will respond with one of three messages:

- `M_SVREPLY` if the requesting thread does not have direct access to the requested data.
- `M_SVREPLYSMP` if the thread does have access (such as in an SMP architecture) and the data has not changed since it was last fetched.
- `M_SVREPLYCLEARCACHE` if the thread does have access (such as in an SMP architecture) and the data has changed since it was last fetched so caches should be flushed by the requesting processor.

M_SVREPLY

Source OM ID	Length	Operation ID	Target thread ID	SV ID	Offset	Data
2 bytes	1 byte	1 byte	2 bytes	2 bytes	4 bytes	Variable
	12 + Data	0x37		ID of SV	Byte offset into the SV	

Sent in response to a `M_SVREAD` message. As this message might be very long (up to 256 bytes in this implementation but potentially longer in others) the implementation processes this message as it is being received byte by byte. This prevents the need for potentially large internal message buffers and minimises the memory footprint of the message handling subsystem.

Note that if the OM and the requesting thread both have direct access to the shared variable (such as in a SMP architecture), then the data field is omitted (and length updated accordingly).

M_SVWRITE

Source OM ID	Length	Operation ID	SV ID	Offset	Data
2 bytes	1 byte	1 byte	2 bytes	4 bytes	Variable
	10 + Data	0x38	ID of SV	Byte offset into the SV	

Writes a block of data into the requested SV at the requested offset. Like `M_SVREPLY` this message might be very long so it is processed incrementally in a similar way.

M_SVREPLYSMP

Source OM ID	Length	Operation ID	SV ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x39	ID of SV

Sent in response to a `M_SVREAD` message when the requesting thread also has direct access to the requested shared variable (such as in an SMP architecture). This message informs the requesting thread that it is OK to proceed, and that the data in the shared variable has not changed since the last time it was requested so it is not necessary to clear caches before.

M_SVREPLYCLEARCACHE

Source OM ID	Length	Operation ID	SV ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x3A	ID of SV

Sent in response to a `M_SVREAD` message when the requesting thread also has direct access to the requested shared variable (such as in an SMP architecture). This message informs the requesting thread that it is OK to proceed, but that the data in the shared variable has changed since the last time it was requested so data cache lines corresponding to this data item should be cleared first.

A.0.5 Protected object operations

M_UDSOCALL

Source OM ID	Length	Operation ID	PO ID	Function ID	Arguments
2 bytes	1 byte	1 byte	2 bytes	1 byte	Variable
		0x40			Byte array interpreted as function arguments

Used by the Anvil protected object model (section 4.12.1) to call a protected function. The `arguments` field is a variable length field which is the arguments of the protected function concatenated as a byte array. All arguments are passed by value.

M_UDSOREPLY

Source OM ID	Length	Operation ID	PO ID	Function ID	Return value
2 bytes	1 byte	1 byte	2 bytes	1 byte	Variable
		0x41			Protected function return value

Sent in response to a `M_POCALL` message, contains the return value of the protected function.

A.0.6 Miscellaneous operations

M_GETENDIANNESS

Source OM ID	Length	Operation ID	Target CPU ID
2 bytes	1 byte	1 byte	2 bytes
	6	0x42	

Request the endianness of a processor.

M_GETENDIANNESSREPLY

Source OM ID	Length	Operation ID	Target OM ID	Endianness
2 bytes	1 byte	1 byte	2 bytes	1 byte
	7	0x43		0 for little-endian 1 for big-endian

Sent in response to a `M_GETENDIANNESS` message.

A.0.7 Migration operations

M_MIGRATE

Source OM ID	Length	Operation ID	Item ID	Target OM ID	Object state
2 bytes	1 byte	1 byte	2 bytes	2 bytes	Variable
		See below			

Sent to migrate an object from one OM to another. The ‘Operation ID’ field is used to identify the type of object being migrated according to the following table:

Operation ID	Migrating object type
0x50	Thread helper
0x51	Mutex
0x52	Condition Variable
0x53	Shared variable

M_POSITION_UPDATE

Source OM ID	Length	Operation ID	Item ID	New CPU ID
2 bytes	1 byte	1 byte	2 bytes	2 bytes
	8	See below		

Sent after a migration from the source OM to inform other processors of the new location of the migrated object. The ‘Operation ID’ field is used to identify the type of object according to the following table:

Operation ID	Migrating object type
0x60	Thread helper
0x61	Mutex
0x62	Condition Variable
0x63	Shared variable

Appendix B

Experimental data

B.1 Effect of undirected migration

Data for figure 5.25. Shows resultant slowdowns experienced after a single migration. ‘Migrated’ shows results if only the thread is migrated. ‘Clusters’ shows results for migrating the thread and its data.

Benchmark	**Local (cycles)**	**Migrated (cycles)**	**Clusters (cycles)**	**Migrated (normalised)**	**Clusters (normalised)**
fdct	9908	216350	28217	21.835	2.847
quicksort	26157	320142	36944	12.239	1.412
binarysearch	1114	9502	10041	8.529	9.013
bsort	78044	1075473	91013	13.780	1.166
sobel	574097	21811154	942935	37.992	1.642

B.2 Transfer efficiency timings

Data for figure 5.21. Shows transfer rates of shared data between the two shared memory blocks of the architecture in figure 5.20 using different transfer methods. Each value is the mean of 16 transfers.

Transfer Size (bytes)	**Sequential access (cycles)**	**Random access (cycles)**	**DMA (cycles)**
4	260	278	203
8	460	461	212
16	839	860	311
32	1605	1668	524
64	3119	3266	911
128	6152	6479	1697
256	12203	12899	3281
512	24324	25737	6365

B.3 Simulation results: varying execution time

Data for figures 5.27 and 5.28. The effect of varying task execution time with and without cluster-based migration. Each value is the mean of 200 simulations.

Cycles	**2x2**	**4x4**	**2x2 (clusters)**	**4x4 (clusters)**
1000	34.44	46.51	49.70	60.70
2000	38.17	54.28	58.49	86.13
3000	42.52	64.80	64.41	96.84
4000	45.85	79.11	75.59	104.15
5000	48.72	85.09	79.90	116.77
8000	57.21	111.39	91.15	131.20
10000	61.26	123.28	93.47	144.77
30000	91.22	206.36	110.76	171.60
50000	99.29	234.05	114.18	176.35
75000	106.14	250.60	116.10	180.21
100000	107.73	260.08	116.92	181.76
500000	114.43	281.56	119.27	185.48
1000000	114.91	284.25	119.34	186.09

Cycles	8x8	16x16	8x8 (clusters)	16x16 (clusters)
1000	64.61	97.63	77.68	84.46
2000	85.74	146.24	91.75	102.73
3000	109.09	189.96	99.76	117.07
4000	140.16	275.68	113.49	129.70
5000	159.61	312.77	120.50	136.80
8000	221.13	433.72	139.45	144.58
10000	257.75	481.49	148.43	161.19
30000	438.31	874.45	175.04	185.85
50000	502.41	1052.13	182.23	195.33
75000	548.97	1142.95	185.15	194.86
100000	570.00	1176.10	186.95	195.41
500000	612.87	1277.88	190.74	200.88
1000000	620.07	1293.10	190.41	202.06

B.4 Simulation results: varying data access frequency

Data for figure 5.30. The effect of varying how frequently tasks access shared data on an 8x8 grid. Each value is the mean of 200 simulations.

Average cycles between shared data access	No clusters	With cluster migration
5000	566.59	793.71
4000	567.75	662.33
3000	569.41	555.75
2000	565.41	423.14
1000	570.55	299.99
800	569.16	275.43
500	570.57	234.84
300	558.77	213.38
200	568.35	200.77
100	560.79	187.07
80	568.00	184.72
50	565.43	182.44
30	567.08	179.69
10	562.96	176.55
5	563.47	175.48

B.5 Simulation results: varying migration frequency

Data for figure 5.29. The effect of varying task migration chance with and without cluster-based migration on an 8x8 grid. Each value is the mean of 200 simulations.

Average cycles between migrations	No clusters	With cluster migration
8000000	34.16	34.60
6000000	35.12	34.50
4000000	36.33	36.65
2000000	46.92	44.02
1000000	58.77	54.37
800000	67.24	59.30
600000	71.18	75.81
400000	97.01	76.90
200000	161.50	113.92
100000	255.37	136.82
80000	287.95	145.90
60000	335.71	153.90
40000	398.39	163.56
20000	506.13	178.18
10000	566.03	186.52
8000	581.66	189.37
6000	594.93	194.72
4000	600.53	205.46
2000	613.82	232.26
1000	620.55	285.51
800	620.64	308.85
600	621.78	355.99
400	623.67	445.22
200	624.33	694.34
100	625.32	1227.86

Appendix C

Hardware generation from Anvil

The presentation of CTV and Anvil in this thesis focussed on its ability to target and refactor a software application for execution on a pre-existing architecture. The architecture is described to the compile-time system by the programmer through the VP mappings. However, as discussed in section 6.2 it would also be possible to leverage the architectural information provided by the programmer to automatically generate an implementation of the target architecture in a hardware description language (HDL) such as VHDL or Verilog. This appendix describes one way in which this could be achieved.

Note that the presented system does not use hardware/software co-design techniques (section 2.4.4). Co-design frameworks build the execution architecture by analysing and measuring the behaviour of the input code and then using a form of search to determine which features to implement in hardware and which in software. This example assumes that the hardware has already been designed at the block level (perhaps using co-design) and describes how Anvil could be extended to provide a fast way to implement it.

Xilinx's Platgen tool [263] is suitable for use as an underlying HDL generator. Platgen is an tool provided by Xilinx for automatically instantiating IP cores and buses to create an embedded system. It takes as its input a system specification file and outputs VHDL, EDIF, and Verilog code that can then be passed through normal FPGA synthesis and place and route tools for implementation. The system specification is a microprocessor hardware specification (MHS) file [264] which describes the following features:

Processors: Instantiated from IP cores in Xilinx's core libraries.

Address spaces: Memories are declared and their connections to microprocessors described in terms of the address ranges that they inhabit and the bus connection that they use.

Bus architectures: Buses are instantiated from a selection of supported types. Currently supported are Processor Local Bus (PLB) from IBM's CoreConnect architecture [112], Local Memory Bus (LMB), Fast Simplex Link (FSL), On-Chip Peripheral Bus (OPB), Xilinx Cache Link (XCL) and a few others.

Peripherals (IP cores): Instantiated from IP cores in Xilinx's core libraries, peripherals may be entirely internal to the system (such as a timer), or may connect to external resources (such as a UART or memory controller).

Connectivity: External signals are defined and connected to internal peripherals allowing the system to communicate with the outside world.

During instantiation, the connections between cores and the various system buses are defined. Cores may export internal signals that are connected to other cores or to external ports. This allows, for example, a timer core to export an interrupt signal that can be routed to the IRQ pin on an embedded processor. Finally, each core has a set of parameters that may be adjusted to change the core's behaviour. Examples of these parameters are the number of internal buffers in a network interface or the baud rate of a serial transceiver.

The process of generating hardware from Anvil therefore concerns the translation of AnvilADL (section 4.2 to an MHS file. Due to the fact that the AnvilADL description is a high-level description of the hardware, the translation has a lot of freedom over implementation-specific details. A possible translation algorithm is as follows:

1. Parse the AnvilADL to determine the elements in the system (processors, channels etc.) and the attributes applied to them.
2. Translate Anvil types to corresponding Xilinx IP cores. Because Platgen is used as the back-end HDL generator, the types requested in the ADL are translated to corresponding supported IP cores. For example, in the current version of Xilinx's tools, all processors must be implemented using either Picoblaze, Microblaze or OpenRISC.
3. Instantiate cores. Code is created to instantiate the processors (Microblaze etc.), channels (UART, mailbox, CAN etc.), and other peripherals (timer etc.) in the target system. No addresses or ports are set at this stage.
4. Determine the connectivity of the system. This is done in two stages, memory connectivity and communications connectivity.
   - A graph of memory connectivity is an unconnected directed graph (N, E) of nodes N and edges E where N is the union of the processors and memory spaces in the system, and an edge $e \in E$ from n_1 to n_2 indicates that memory space n_2 is mapped into the address range of processor n_1. This is determined by parsing the `memory` attributes of processor objects.
   - A graph of communications connectivity is an unconnected directed graph (N, E) of nodes N and edges E where N is the union of the processors and communication channels in the system, and an edge $e \in E$ from n_1 to n_2 indicates that processor n_1 is connected to channel n_2, and therefore can communicate with other processors that are connected to the same channel. This is determined by parsing the `endpoints` attribute of channel objects.

   From these two graphs the bus topology of the system can be generated.
5. Generate bus topology. The bus topology of the system can now be determined. There are two topology schemes that can be used, and schemes could be mixed.

- **Dedicated bus:** This scheme generates a single memory bus and a single peripheral bus for each processor in the target system. (If a processor has no peripherals then the peripheral bus would not be generated.) This involves the instantiation of a bus in the MHS file and then setting the `BUS_INTERFACE` parameter of each processor instance. Then, the channels are connected to the peripheral buses of their attached processors, according to the communications connectivity graph. Peripherals are attached to the peripheral bus. This involves setting the `BUS_INTERFACE` parameter of the peripheral instance. For IP cores that need to be connected to the buses of many processors, a shared bus can be created and bridged to all desired buses.
- **Shared bus:** The scheme described above provides high data throughput as each processor has its own peripheral and memory bus. However it is also possible to create an shared bus architecture in which processors share the same peripheral and memory bus. This results in reduced bandwidth, but also lower logic area requirements as fewer bus controllers need to be instantiated. Some buses have hard limits on the maximum number of slaves or masters that can be connected which must be checked.

Extra attributes would need to be added to the AnvilADL description to allow the programmer to specify the bus topology required.

6. Generate address maps. Once the connected items have been determined for each bus, the address map can be built. Each peripheral has a required address size which can be determined automatically by checking the definition of the IP core. For memory controllers, the size is determined from the `depth` and `width` attributes in the AnvilADL of the corresponding memory object.
7. Connect interrupts. The `ports` attribute of peripherals is used to inform the hardware generation process which elements of hardware can interrupt which processors. If a processor does not provide enough native interrupt lines then an interrupt controller instance can be created and connected appropriately by instantiating a pre-built processor-specific template.
8. Synthesise MHS file.

The resulting MHS file could then be passed to Platgen, which will generate the target architecture as a set of HDL files and synthesis scripts for implementation of the architecture on an FPGA.

C.1 Hardware generation example

This section gives an example of how the Anvil hardware generation could operate. The example system is a simple shared-bus, dual-processor system with no shared memory but mailbox communication and an external serial interface. This is shown in figure C.1.

Such an architecture is described by the following ADL:

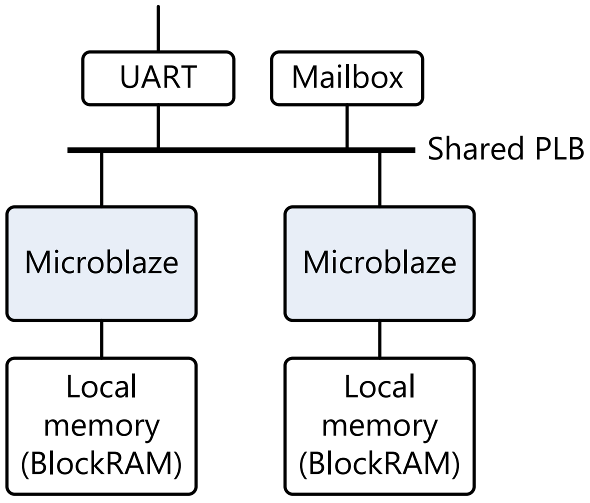

Figure C.1: Example of an architecture that could be generated.

```
cpu1, cpu2 : processor Microblaze;
mem1, mem2 : memory BlockRAM;
cpu1^memory = mem1;
cpu2^memory = mem2;

mbox = channel Mailbox;
uart = hardware UART(115200, 100000000);

mbox^endpoints = [cpu1(0x81e20000, "mem", 0), cpu2(0x82040000, "mem", 0)];
uart^ports = [cpu1(0x84000000, "mem", None), cpu2(0x84000000, "mem", None)];
```

Although given in the above example, is it not necessary for the programmer to specify addresses if they are not otherwise constrained as the system could generate and assign them automatically. From analysis of the ADL it is possible to build an MHS file around a single shared PLB bus. A number of extra components are added automatically (a clock generator and reset generator, and instruction and data local memory buses) from templates provided by Xilinx. An example resulting MHS file is shown here.

```
PORT sys_clk_pin = dcm_clk_s, DIR = I, SIGIS = CLK
PORT sys_rst_pin = sys_rst_s, DIR = I, RST_POLARITY = 0, SIGIS = RST
PORT fpga_0_RS232_Uart_RX_pin = fpga_0_RS232_Uart_RX, DIR = I
PORT fpga_0_RS232_Uart_TX_pin = fpga_0_RS232_Uart_TX, DIR = O

BEGIN clock_generator
    #Details skipped for brevity
END

BEGIN proc_sys_reset
    #Details skipped for brevity
END
```

```
BEGIN microblaze
    PARAMETER INSTANCE = microblaze_0
    PARAMETER C_INTERCONNECT = 1
    PARAMETER HW_VER = 7.20.d
    BUS_INTERFACE DPLB = plb_0
    BUS_INTERFACE IPLB = plb_0
    BUS_INTERFACE DLMB = dlmb_0
    BUS_INTERFACE ILMB = ilmb_0
    PORT MB_RESET = mb_reset
    PORT INTERRUPT = xps_mailbox_0_Interrupt_0
END

BEGIN microblaze
    PARAMETER INSTANCE = microblaze_1
    PARAMETER HW_VER = 7.20.d
    BUS_INTERFACE IPLB = plb_0
    BUS_INTERFACE DPLB = plb_0
    BUS_INTERFACE ILMB = ilmb_1
    BUS_INTERFACE DLMB = dlmb_1
    PORT MB_RESET = mb_reset
    PORT INTERRUPT = xps_mailbox_0_Interrupt_1
END

BEGIN plb_v46
    PARAMETER INSTANCE = plb_0
    PARAMETER HW_VER = 1.04.a
    PORT PLB_Clk = sys_clk_s
    PORT SYS_Rst = sys_bus_reset
END

BEGIN lmb_v10
    PARAMETER INSTANCE = dlmb_0
    PARAMETER HW_VER = 1.00.a
    PORT SYS_Rst = sys_bus_reset
    PORT LMB_Clk = sys_clk_s
END
#And similar LMB declarations for ilmb_0, dlmb_1 and ilmb_1

BEGIN lmb_bram_if_cntlr
    PARAMETER INSTANCE = dlmb_cntlr_0
    PARAMETER HW_VER = 2.10.b
    PARAMETER C_BASEADDR = 0x00000000
    PARAMETER C_HIGHADDR = 0x00007FFF
    BUS_INTERFACE SLMB = dlmb_0
    BUS_INTERFACE BRAM_PORT = dlmb_cntlr_0_BRAM_PORT
END
#And similar BRAM controller declarations for ilmb_cntlr_0,
#dlmb_cntlr_1 and ilmb_cntlr_1
```

```
BEGIN bram_block
    PARAMETER INSTANCE = lmb_bram_0
    PARAMETER HW_VER = 1.00.a
    BUS_INTERFACE PORTB = dlmb_cntlr_0_BRAM_PORT
    BUS_INTERFACE PORTA = ilmb_cntlr_0_BRAM_PORT
END
#And a similar BRAM declaration for lmb_bram_1

BEGIN xps_mailbox
    PARAMETER INSTANCE = xps_mailbox_0
    PARAMETER HW_VER = 1.00.a
    PARAMETER C_SPLB0_BASEADDR = 0x81e20000
    PARAMETER C_SPLB0_HIGHADDR = 0x81e2ffff
    PARAMETER C_SPLB1_BASEADDR = 0x82040000
    PARAMETER C_SPLB1_HIGHADDR = 0x8204ffff
    BUS_INTERFACE SPLB0 = plb_0
    BUS_INTERFACE SPLB1 = plb_0
    PORT SYS_Rst = mb_reset
    PORT Interrupt_0 = xps_mailbox_0_Interrupt_0
    PORT Interrupt_1 = xps_mailbox_0_Interrupt_1
END

BEGIN xps_uartlite
    PARAMETER INSTANCE = RS232_Uart
    PARAMETER HW_VER = 1.01.a
    PARAMETER C_BAUDRATE = 115200
    PARAMETER C_DATA_BITS = 8
    PARAMETER C_ODD_PARITY = 0
    PARAMETER C_USE_PARITY = 0
    PARAMETER C_SPLB_CLK_FREQ_HZ = 100000000
    PARAMETER C_BASEADDR = 0x84000000
    PARAMETER C_HIGHADDR = 0x8400ffff
    BUS_INTERFACE SPLB = plb_0
    PORT RX = fpga_0_RS232_Uart_RX
    PORT TX = fpga_0_RS232_Uart_TX
END
```

Appendix D

Anvil Example

This appendix shows a simple system with two threads (the main thread and one invoked thread) to perform Sobel edge filtering in both the X and Y directions in parallel. This would typically be used in the first stage of a vision system.

D.1 Input application

The following four files are the input to the Anvil refactoring engine. The application's code is contained in `main.c`, `sobel.c` and `sobel.h`. `mappings.anv` contains the AnvilADL mappings.

D.1.1 mappings.anv

```
cpu1 : processor Microblaze;
cpu2 : processor Picoblaze;
mem1, mem2, sharedmem : memory BlockRAM;

cpu1^memory = [mem1(0x00000000)];
cpu2^memory = [mem2(0x00000000)];
cpu1^extramemory = [sharedmem(0x8C000000)];
cpu2^extramemory = [sharedmem(0x8C000000)];

cpu1^threads = "main";
cpu2^threads = "thread2";
sharedmem^variables = ["sourceimage", "outputx", "outputy"];

mbox : channel Mailbox;
mbox^endpoints = [cpu1(0x84000000, "mem", 0), cpu2(0x84000000, "mem", 0)];

om : manager;
om^manages = ["main", "thread2", "sourceimage", "outputx", "outputy"];
```

D.1.2 main.c

```c
#include <mblib.h>
#include <pthread.h>
#include "addresses.h"
#include "sobel.h"

char sourceimage[IMSIZE][IMSIZE];
char outputx[IMSIZE][IMSIZE];
char outputy[IMSIZE][IMSIZE];

void* thread2body(void *arg) {
 yfilter(sourceimage, outputy);
 pthread_exit(0);
}

int main(void){
 pthread_t thread2;

 //Enable interrupts
 intc_enable_all_interrupts(intc_0);
 intc_master_enable(intc_0);
 mb_enable_interrupts();

 uart_send_char(external_uart, '!');
 uart_send_char(external_uart, '\n');

 //Start timer
 initialise_timer(timer_0);
 start_timer(timer_0);

 //Set off other thread
 pthread_create(&thread2, 0, thread2body, 0);

 //Filter here
 xfilter(sourceimage, outputx);

 //Wait for other thread
 pthread_join(thread2, 0);

 //Print time
 uart_print_int(external_uart, *(timer_0 + 2));
 uart_send_char(external_uart, '\n');

 return 0;
}
```

D.1.3 sobel.h

```c
#ifndef _SOBEL_H
#define _SOBEL_H

#define IMSIZE 200
```

```
void xfilter(char image[IMSIZE][IMSIZE], char dest[IMSIZE][IMSIZE]);
void yfilter(char image[IMSIZE][IMSIZE], char dest[IMSIZE][IMSIZE]);

#endif
```

D.1.4 sobel.c

```
#include "sobel.h"

void xfilter(char image[IMSIZE][IMSIZE], char dest[IMSIZE][IMSIZE])
{
 int weight[3][3] = {{-1,0,1},{-2,0,2},{-1,0,1}};
 int x, y, i, j;

 for (y = 1; y < IMSIZE - 1; y++)
  for (x = 1; x < IMSIZE - 1; x++)
  {
   dest[y][x] = 0;
   for (j = -1; j <= 1; j++)
    for (i = -1; i <= 1; i++)
     dest[y][x] += weight[j + 1][i + 1] * image[y + j][x + i];
  }
}

void yfilter(char image[IMSIZE][IMSIZE], char dest[IMSIZE][IMSIZE])
{
 int weight[3][3] = {{-1,-2,1},{0,0,0},{-1,2,1}};
 int x, y, i, j;

 for (y = 1; y < IMSIZE - 1; y++)
  for (x = 1; x < IMSIZE - 1; x++)
  {
   dest[y][x] = 0;
   for (j = -1; j <= 1; j++)
    for (i = -1; i <= 1; i++)
     dest[y][x] += weight[j + 1][i + 1] * image[y + j][x + i];
  }
}
```

D.2 Anvil output

Nine files are produced by the Anvil refactoring engine:

File	Language	Purpose
_anvil_settings_cpu0.h	C header	Configures the Anvil pthreads library
_anvil_settings_cpu1.h	C header	Configures the Anvil pthreads library
_anvil_specific_cpu0.c	C code	Low-level hardware drivers
_anvil_specific_cpu1.c	C code	(Same as _anvil_specific_cpu0.c, see below)
linkscript.ld		LD link script
microblaze_0.c	C code	Code for CPU 0
microblaze_1.c	C code	Code for CPU 1
sobel.c	C code	(As input, not shown)
sobel.h	C header	(As input, not shown)

The following points can be observed in the output files:

- The code has been split so that the thread bodies are now in separate files, one for each processor. A custom main function was generated for the invoked thread.
- The shared data items have been moved into shared memory using attributes, and a custom linker script generated based on the architecture description (`linkscript.ld`) to inform the linker of this.
- The code now depends on Anvil's embedded pthreads implementation. This implements the internals of the OM in the system (located on processor 0).
- The pthreads calls are refactored accordingly.
- The `_anvil_specific_cpuX.c` files are created to implement the low-level communications and interrupt handling. As this architecture is symmetric these files are identical, but on more complex architectures this file can grow to be quite large, if a lot of custom hardware must be accessed. Drivers for hardware devices (such as the Mailbox device in this case) are included here, and offline routing implemented here.
- The `_anvil_settings_cpuX.h` files are created to configure the Anvil pthreads library according to any OMs hosted by the processor to ensure that only the minimal subset of the library is linked in.

D.2.1 _anvil_settings_cpu0.h

```
#define _ANVIL_THISTHREADID 0
#define _ANVIL_TOTAL_MANAGED_THREADS 2
#define _ANVIL_TOTAL_MANAGED_MUTEXES 0
#define _ANVIL_TOTAL_MANAGED_VARS 3
#define _ANVIL_TOTAL_MANAGED_CVS 0
#define _ANVIL_TOTAL_ACCESSED_VARS 0
#define _ANVIL_TOTAL_SYSTEM_THREADS 2
#define _ANVIL_TOTAL_SYSTEM_MUTEXES 0
#define _ANVIL_TOTAL_SYSTEM_CVS 0
#define _ANVIL_MANAGED_THREADS_INITIALISER {0, 0, 0}, {1, 0, 0}
#define _ANVIL_MANAGED_MUTEXES_INITIALISER
#define _ANVIL_MANAGED_CVS_INITIALISER
#define _ANVIL_MANAGED_VARS_INITIALISER {0, 0, 4}, {1, 0, 4}, {2, 0, 4}, {3, 0, 4}
```

```
#define _ANVIL_ACCESSED_VARS_INITIALISER
#define _ANVIL_CIRC_BUFF_SIZE 8
#define _ANVIL_MSG_IRQS 1
#define _ANVIL_MSG_IRQS_INITIALISER {0, 0, -1}
#define _ANVIL_ACCS 0
#define _ANVIL_ACCS_INITIALISER
```

D.2.2 _anvil_settings_cpu1.h

```
#define _ANVIL_THISTHREADID 1
#define _ANVIL_TOTAL_MANAGED_THREADS 0
#define _ANVIL_TOTAL_MANAGED_MUTEXES 0
#define _ANVIL_TOTAL_MANAGED_VARS 0
#define _ANVIL_TOTAL_MANAGED_CVS 0
#define _ANVIL_TOTAL_ACCESSED_VARS 0
#define _ANVIL_TOTAL_SYSTEM_THREADS 2
#define _ANVIL_TOTAL_SYSTEM_MUTEXES 0
#define _ANVIL_TOTAL_SYSTEM_CVS 0
#define _ANVIL_MANAGED_THREADS_INITIALISER
#define _ANVIL_MANAGED_MUTEXES_INITIALISER
#define _ANVIL_MANAGED_CVS_INITIALISER
#define _ANVIL_MANAGED_VARS_INITIALISER
#define _ANVIL_ACCESSED_VARS_INITIALISER
#define _ANVIL_CIRC_BUFF_SIZE 8
#define _ANVIL_MSG_IRQS 1
#define _ANVIL_MSG_IRQS_INITIALISER {0, 0, -1}
#define _ANVIL_ACCS 0
#define _ANVIL_ACCS_INITIALISER
```

D.2.3 _anvil_specific_cpu0.c

```
#include "anvil_pthreads.h"
#include "mblib.h"

volatile int *mailbox = (int *)0x84000000;
volatile int *timer_0 = (int *)0x85000000 ;

void _anvil_send_to_thread(int targetthreadid, int *packet, int len)
{
 int x;

 for (x = 0; x < len; x++)
 {
  switch(targetthreadid)
  {
   case 0:
    break;
   case 1:
    mbox_write(mailbox, packet[x]);
    break;
  }
 }
}

void _anvil_interrupts_disable(void)
{
 mb_disable_interrupts();
}

void _anvil_interrupts_enable(void)
{
 mb_enable_interrupts();
}

int _anvil_read_from_interrupt(int vec)
{
 switch(vec)
 {
  case 0:
   //mbox
   return mbox_read(mailbox_0);
   break;
  default:
   //Unknown interrupt, disable it
   intc_disable_interrupt(intc_0, vec);
   return 0;
   break;
 }
}

int _anvil_interrupt_ready(int vec)
{
 switch(vec)
 {
  case 0:
   //mbox
   return mbox_check(mailbox_0);
   break;
  default:
   //Unknown interrupt, disable it
   intc_disable_interrupt(intc_0, vec);
   return 0;
   break;
 }
}

int _anvil_get_interrupt_vector(void)
{
 return 0;
}

void _anvil_acknowledge_interrupt(int vec)
{
 //nop
}
```

```
void _anvil_sleep(void)
{
 //nop
}
```

D.2.4 linkscript.ld

```
OUTPUT_FORMAT("elf32-microblaze", "", "")
ENTRY(_start)
_TEXT_START_ADDR = DEFINED(_TEXT_START_ADDR) ? _TEXT_START_ADDR : 0x50;
_HEAP_SIZE = DEFINED(_HEAP_SIZE) ? _HEAP_SIZE : 0x0;
_STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE : 0x400;
SECTIONS
{
 .vectors.reset 0x0 : { KEEP (*(.vectors.reset)) } = 0
 .vectors.sw_exception 0x8 : { KEEP (*(.vectors.sw_exception)) } = 0
 .vectors.interrupt 0x10 : { KEEP (*(.vectors.interrupt)) } = 0
 .vectors.debug_sw_break 0x18 : { KEEP (*(.vectors.debug_sw_break)) } = 0
 .vectors.hw_exception 0x20 : { KEEP (*(.vectors.hw_exception)) } = 0
 . = _TEXT_START_ADDR;
  _ftext = .;
 .text : {
  *(.text)
  *(.text.*)
  *(.gnu.linkonce.t.*)
 }
  _etext = .;
 .init : { KEEP (*(.init)) } =0
 .fini : { KEEP (*(.fini)) } =0
 PROVIDE (__CTOR_LIST__ = .);
 PROVIDE (___CTOR_LIST__ = .);
 .ctors :
 {
  KEEP (*crtbegin.o(.ctors))
  KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
  KEEP (*(SORT(.ctors.*)))
  KEEP (*(.ctors))
 }
 PROVIDE (__CTOR_END__ = .);
 PROVIDE (___CTOR_END__ = .);
 PROVIDE (__DTOR_LIST__ = .);
 PROVIDE (___DTOR_LIST__ = .);
  .dtors     :
 {
  KEEP (*crtbegin.o(.dtors))
  KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
  KEEP (*(SORT(.dtors.*)))
  KEEP (*(.dtors))
 }
 PROVIDE (__DTOR_END__ = .);
 PROVIDE (___DTOR_END__ = .);
  . = ALIGN(4);
  _frodata = . ;
 .rodata : {
  *(.rodata)
  *(.rodata.*)
  *(.gnu.linkonce.r.*)
  CONSTRUCTORS; /* Is this needed? */
 }
  _erodata = .;
 /* Alignments by 8 to ensure that _SDA2_BASE_ on a word boundary */
 /* Note that .sdata2 and .sbss2 must be contiguous */
 . = ALIGN(8);
  _ssrw = .;
 .sdata2 : {
  *(.sdata2)
  *(.sdata2.*)
  *(.gnu.linkonce.s2.*)
 }
 . = ALIGN(4);
 .sbss2 : {
  PROVIDE (__sbss2_start = .);
  *(.sbss2)
  *(.sbss2.*)
  *(.gnu.linkonce.sb2.*)
  PROVIDE (__sbss2_end = .);
 }
 . = ALIGN(8);
  _essrw = .;
  _ssrw_size = _essrw - _ssrw;
  PROVIDE (_SDA2_BASE_ = _ssrw + (_ssrw_size / 2 ));
  . = ALIGN(4);
  _fdata = .;
 .data : {
  *(.data)
  *(.gnu.linkonce.d.*)
  CONSTRUCTORS; /* Is this needed? */
 }
  _edata = . ;
  /* Added to handle pic code */
 .got : {
  *(.got)
 }
 .got1 : {
  *(.got1)
 }
 .got2 : {
  *(.got2)
 }
 /* Added by Sathya to handle C++ exceptions */
 .eh_frame : {
  *(.eh_frame)
 }
 .jcr : {
  *(.jcr)
```

```
}
.gcc_except_table : {
 *(.gcc_except_table)
}
/* Alignments by 8 to ensure that _SDA_BASE_ on a word boundary */
/* Note that .sdata and .sbss must be contiguous */
. = ALIGN(8);
 _ssro = .;
.sdata : {
 *(.sdata)
 *(.sdata.*)
 *(.gnu.linkonce.s.*)
}
. = ALIGN(4);
.sbss : {
 PROVIDE (__sbss_start = .);
 *(.sbss)
 *(.sbss.*)
 *(.gnu.linkonce.sb.*)
 PROVIDE (__sbss_end = .);
}
. = ALIGN(8);
 _essro = .;
 _ssro_size = _essro - _ssro;
PROVIDE (_SDA_BASE_ = _ssro + (_ssro_size / 2 ));
. = ALIGN(4);
 _fbss = .;
.bss : {
 PROVIDE (__bss_start = .);
 *(.bss)
 *(.bss.*)
 *(.gnu.linkonce.b.*)
 *(COMMON)
 . = ALIGN(4);
 PROVIDE (__bss_end = .);
}

. = ALIGN(4);
.heap : {
  _heap = .;
  _heap_start = .;
  . += _HEAP_SIZE;
  _heap_end = .;
}

. = ALIGN(4);
.stack : {
  _stack_end = .;
  . += _STACK_SIZE;
  . = ALIGN(8);
  _stack = .;
  _end = .;
}
```

```
 .tdata : {
  *(.tdata)
  *(.tdata.*)
  *(.gnu.linkonce.td.*)
 }
 .tbss : {
  *(.tbss)
  *(.tbss.*)
  *(.gnu.linkonce.tb.*)
 }

 . = 0x8C000000;
 sharedmemory : { }
}
```

D.2.5 microblaze_0.c

```
#include <mblib.h>
#include "anvil_pthreads.h"
#include "addresses.h"
#include "sobel.h"

char sourceimage[IMSIZE][IMSIZE] __attribute__((section ("sharedmemory")));
char outputx[IMSIZE][IMSIZE] __attribute__((section ("sharedmemory")));
char outputy[IMSIZE][IMSIZE] __attribute__((section ("sharedmemory")));

int main(void)
{
 //Enable interrupts
 intc_enable_all_interrupts(intc_0);
 intc_master_enable(intc_0);
 mb_enable_interrupts();

 uart_send_char(external_uart, '!');
 uart_send_char(external_uart, '\n');

 //Start timer
 initialise_timer(timer_0);
 start_timer(timer_0);

 //Set off other thread
 pthread_create(1, 0); //Thread ID 1, OM is on CPU 0

 //Filter here
 xfilter(sourceimage, outputx);

 //Wait for other thread
 pthread_join(1, 0); //Thread ID 1, OM is on CPU 0

 //Print time
 uart_print_int(external_uart, *(timer_0 + 2));
 uart_send_char(external_uart, '\n');
 return 0;
}
```

D.2.6 microblaze_1.c

```
#include <mblib.h>
#include "anvil_pthreads.h"
#include "addresses.h"
#include "sobel.h"

char sourceimage[IMSIZE][IMSIZE] __attribute__((section ("sharedmemory")));
char outputx[IMSIZE][IMSIZE] __attribute__((section ("sharedmemory")));
char outputy[IMSIZE][IMSIZE] __attribute__((section ("sharedmemory")));

void* thread2body(void *arg) {
 yfilter(sourceimage, outputy);
 pthread_exit(0);
}

int main(void)
{
 //Enable interrupts
 mb_enable_interrupts();

 //Wait until we (thread 1) are started by the main thread
 _anvil_wait_until_released();

 //Call the thread body
 thread2body();

 return 0;
}
```

Appendix E

Anvil UDSO Example

This appendix contains the full source of the Anvil UDSO example presented in section 4.12.3. This example is mapped onto a simple dual core SMP system using the AnvilADL in `udsoexample.anv`. The input source files are `main.c`, `queues.h` and `queues.c`. After Anvil's refactoring has been applied, the two output sources `cpu1.c` and `cpu2.c` are produced. Note that in the output files, the entire code for the queue datatype has been removed. This is because the queue code is compiled into the OM which manages the UDSO (which in this case is mapped to `cpu1`.

E.1 udsoexample.anv

```
cpu1, cpu2 : processor Microblaze;
mem1, mem2 : memory BlockRAM;
cpu1^memory = [mem1(0x0, 0x7FFF)];
cpu2^memory = [mem2(0x0, 0x7FFF)];
mem1^size = 8192;
mem2^size = 8192;
mem1^width = 32;
mem2^width = 32;

mbox : channel Mailbox;
mbox^endpoints = [cpu1(0x80000000, 0), cpu2(0x80000000, 0)];

cpu1^threads = ["main", "producer_thread"];
cpu2^threads = ["consumer_thread"];

queue : protected;
queue^identifier = "thequeue";
queue^functions = ["queue_put"(1), "queue_get"(1), "queue_empty"(1)];

om : manager;
om^manages = [queue, "mutex", "cond", "main", "producer_thread", "consumer_thread"];
om^executes_on = [cpu1];
```

E.2 main.c

```
/*
Producer / Consumer example

Note that the code in this implementation has been simplified for clarity
and does not represent a complete solution to the producer/consumer problem.
*/

#include "queues.h"
#include <pthread.h>
#include <stdio.h>

queue_t thequeue = QUEUE_INIT;
pthread_t producer_thread, consumer_thread;

pthread_cond_t cond;
pthread_mutex_t mutex;


//Dummy produce and consume functions
int produce_next_item(void) {
 static index = 0;
 sleep(1);
 return index++;
}
void consume_item(int item) {
 printf("%d\n", item);
}

//Thread bodies
void *producer_thread_body(void *t)
{
   int item;
   while(1) //Run constantly for this example
   {
      item = produce_next_item(); //Produce an item
   pthread_mutex_lock(&mutex);
      queue_put(&thequeue, item); //Insert the item into the queue
   pthread_cond_signal(&cond); //Signal the consumer
   pthread_mutex_unlock(&mutex);
   }
 pthread_exit(0);
}

void *consumer_thread_body(void *t)
{
   int item;
   while(1) //Run constantly for this example
   {
   pthread_mutex_lock(&mutex);
   pthread_cond_wait(&cond, &mutex); //Wait to be signalled
   while(!queue_empty(&thequeue)) {
    item = queue_get(&thequeue); //Get an item from the queue
    consume_item(item); //Consume the item
   }
   pthread_mutex_unlock(&mutex);
   }
 pthread_exit(0);
}


//Main function
int main(void)
{
 pthread_mutex_init(&mutex, 0);
 pthread_cond_init(&cond, 0);

   pthread_create(&producer_thread, 0, producer_thread_body, 0);
   pthread_create(&consumer_thread, 0, consumer_thread_body, 0);

 pthread_join(producer_thread, 0);
 pthread_join(consumer_thread, 0);
}
```

E.3 queues.h

```
#ifndef _QUEUE_H
#define _QUEUE_H

typedef struct q_node {
   int dataitem;
   struct q_node *next;
} queue_t;

#define QUEUE_INIT {0,0}

void queue_put(queue_t *q, int data);
int queue_get(queue_t *q);
int queue_count(queue_t *q);
int queue_empty(queue_t *q);

#endif
```

E.4 queues.c

```
#include "queues.h"

void queue_put(queue_t *q, int data) {
 while (q->next != 0) q = q->next;
 q->next = (queue_t *) malloc (sizeof (queue_t));
 q = q->next;
 q->next = 0;
 q->dataitem = data;
}

int queue_get(queue_t *q) {
 queue_t *temp;
 int rv;

 if(q->next != 0) {
  temp = q->next;
  rv = temp->dataitem;
  q->next = temp->next;
  free(temp);
  return rv;
 }
 return -1;
}

int queue_count(queue_t *q) {
 int count = 0;
 if(q != 0) {
  while(q->next != 0) {
   count++;
   q = q->next;
  }
 }
 return count;
}


int queue_empty(queue_t *q) {
 if(q->next == 0) return 1; else return 0;
}
```

E.5 cpu1.c

```
#include "queues.h"
#include "anvil_pthreads.h"
#include <stdio.h>

//Dummy produce and consume functions
int produce_next_item(void) {
 static index = 0;
 sleep(1);
 return index++;
}

//Thread bodies
void *producer_thread_body(void *t)
{
  int item;
  while(1) //Run constantly for this example
  {
    item = produce_next_item(); //Produce an item
  pthread_mutex_lock(0, 0); //Mutex ID 0, OM is on CPU 0
    _anvil_udso(0, 0, item); //UDSO ID 0, queue_put
  pthread_cond_signal(0, 0); //CV ID 0, OM is on CPU 0
  pthread_mutex_unlock(0, 0); //Mutex ID 0, OM is on CPU 0
  }
 pthread_exit(1, 0); //Thread ID 1, OM is on CPU 0
}

//Main function
int main(void)
{
 pthread_mutex_init(0, 0); //Mutex ID 0, OM is on CPU 0
 pthread_cond_init(0, 0); //CV ID 0, OM is on CPU 0

  pthread_create(1, 0); //Thread ID 1, OM is on CPU 0
  pthread_create(2, 0); //Thread ID 2, OM is on CPU 0

 pthread_join(1, 0); //Thread ID 1, OM is on CPU 0
 pthread_join(2, 0); //Thread ID 2, OM is on CPU 0
}
```

E.6 cpu2.c

```
#include "anvil_pthreads.h"
#include <stdio.h>

//Dummy produce and consume functions
void consume_item(int item) {
 printf("%d\n", item);
}

//Thread bodies
void *consumer_thread_body(void *t)
{
  int item;
  while(1) //Run constantly for this example
  {
  pthread_mutex_lock(0, 0); //Mutex ID 0, OM is on CPU 0
  pthread_cond_wait(0, 0, 0, 0); //CVID 0, OM on CPU 0, Mutex ID 0, OM on CPU 0
  while(!_anvil_udso(0, 2)) { //UDSO ID 0, queue_empty
   item = _anvil_udso(0, 1); //UDSO ID 0, queue_get
   consume_item(item); //Consume the item
  }
  pthread_mutex_unlock(0, 0);
  }
 pthread_exit(2, 0); //Thread ID 2, OM is on CPU 0
}
```

Appendix F

3-DES Example

This appendix contains the full source of the 3-DES program presented in section 5.1.5. The input application is shown, and then an instance of the output when it is mapped over three of the five cores in the target architecture (shown in figure 5.10).

F.1 Input application

F.1.1 mappings.anv

```
cpu1 : processor Microblaze;
cpu2 : processor Microblaze;
cpu3 : processor Microblaze;
mem1, mem2, mem3, sharedmem : memory BlockRAM;

cpu1^memory = [mem1(0x00000000), sharedmem(0x8C000000)];
cpu2^memory = [mem2(0x00000000), sharedmem(0x8C000000)];
cpu3^memory = [mem2(0x00000000), sharedmem(0x8C000000)];

cpu1^threads = ["main", "workers[0]"];
cpu2^threads = ["workers[1]"];
cpu3^threads = ["workers[2]"];
sharedmem^variables = ["outputdata"];

outputdata : variable;
outputdata^mutex = "outputstream";
outputdata^writecache = 64;

mbox1 : channel Mailbox;
mbox1^endpoints = [cpu1(0x84000000, "mem", 0), cpu2(0x84000000, "mem", 0)];

mbox2 : channel Mailbox;
mbox2^endpoints = [cpu1(0x85000000, "mem", 1), cpu3(0x84000000, "mem", 0)];
```

```
om : manager;
om^manages = ["main", "workers", "outputdata", "inputstream", "outputstream"];
```

F.1.2 main.c

```
#include <pthread.h>
#include "des.h"

#define BLOCKS 5000
#define BLOCKSIZE 64
#define WORKER_THREADS 3

//Prototypes
void *worker_body(void *a);
void outputblock(bool *block, int blocknum);
void getblock(bool *block, int blockno);

//pthread API objects
pthread_t workers[WORKER_THREADS];
pthread_mutex_t inputstream = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t outputstream = PTHREAD_MUTEX_INITIALIZER;

bool outputdata[BLOCKS][BLOCKSIZE];

//------------------------------------------------------------

//Main function
int main(int argc, char *argv[]) {
 int x;

 //Create worker threads
 for(i = 0; i < WORKER_THREADS; i++)
  pthread_create(&workers[i], NULL, worker_body, &i);

 //Join workers
 for(i = 0; i < WORKER_THREADS; i++)
  pthread_join(workers[i], 0);

 return 0;
}

//------------------------------------------------------------

//Body function for the workers
//Should be passed an integer to give it an ID
void *worker_body(void *a) {
 int blockno, x;
 bool inblock[BLOCKSIZE], outblock[BLOCKSIZE];

 //3DES keys, distributed to each thread
 bool key1[56] = {
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
   0,0,0,0,0,0,0,0};
 bool key2[56] = {
   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1};
 bool key3[56] = {
   1,1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,
   1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,
   1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,
   1,0,0,0,0,0,0,0};

 //Read the ID
 num = *(int *)a;

 //Begin encryption loop
 for(blockno = num; blockno < BLOCKS; blockno += WORKER_THREADS) {
  //Grab the mutex for the input stream
  pthread_mutex_lock(&inputstream);
  //Get the block
  getblock(inblock, num);
  //Release the mutex
  pthread_mutex_unlock(&inputstream);

  //Perform 3DES
  EncryptDES(key1, outblock, inblock, 0);
  EncryptDES(key2, inblock, outblock, 0);
  EncryptDES(key3, outblock, inblock, 0);

  //Grab the mutex for the output stream
  pthread_mutex_lock(&outputstream);
  //Write the block out
  outputblock(outblock, blockno);
  //Release the mutex
  pthread_mutex_unlock(&outputstream);
 }

 pthread_exit(0);
}

//------------------------------------------------------------

void getblock(bool *block, int blockno) {
 //Implementation omitted. Could be read from memory or an external I/O device
}

void outputblock(bool *block, int blocknum) {
 //Implementation omitted. Could be written to memory or an external I/O device
}
```

F.1.3 des.h

```
/*****************************************************************************
 * des.h                                                 *
 *                  Header file for des.c                *
 *                                                       *
 *  Written 1995-8 by Cryptography Research (http://www.cryptography.com) *
 *  Original version by Paul Kocher. Placed in the public domain in 1998. *
 * THIS IS UNSUPPORTED FREE SOFTWARE. USE AND DISTRIBUTE AT YOUR OWN RISK. *
 *                                                       *
 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT OF THIS PROGRAM. *
 *                                                       *
 *****************************************************************************
 *                                                       *
 *  REVISION HISTORY:                                    *
 *                                                       *
 *  Version 1.0: Initial release -- PCK.                 *
 *  Version 1.1: Changes and edits for EFF DES Cracker project. *
 *                                                       *
 *****************************************************************************/

#ifndef __DES_H
#define __DES_H

typedef char bool;
void EncryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose);
void DecryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose);

#endif
```

F.1.4 des.c

```
/*****************************************************************************
 * des.c                                                 *
 *           Software Model of ASIC DES Implementation *
 *                                                       *
 *  Written 1995-8 by Cryptography Research (http://www.cryptography.com) *
 *  Original version by Paul Kocher. Placed in the public domain in 1998. *
 * THIS IS UNSUPPORTED FREE SOFTWARE. USE AND DISTRIBUTE AT YOUR OWN RISK. *
 *                                                       *
 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT OF THIS PROGRAM. *
 *                                                       *
 *****************************************************************************
 *                                                       *
 *  IMPLEMENTATION NOTES:                                *
 *                                                       *
 *  This DES implementation adheres to the FIPS PUB 46 spec and produces *
 *  standard output. The internal operation of the algorithm is slightly *
 *  different from FIPS 46. For example, bit orderings are reversed *
 *  (the right-hand bit is now labelled as bit 0), the S tables have *
 *  rearranged to simplify implementation, and several permutations have *
 *  been inverted. For simplicity and to assist with testing of hardware *
 *  implementations, code size and performance optimizations are omitted. *
 *                                                     *
 ***************************************************************************
 *                                                     *
 *  REVISION HISTORY:                                  *
 *                                                     *
 *  Version 1.0: Initial release -- PCK.               *
 *  Version 1.1: Altered DecryptDES exchanges to match EncryptDES. -- PCK *
 *  Version 1.2: Minor edits and beautifications. -- PCK *
 *  Version 1.3: Changes and edits for EFF DES Cracker project. *
 *                                                     *
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "des.h"

static void ComputeRoundKey(bool roundKey[56], bool key[56]);
static void RotateRoundKeyLeft(bool roundKey[56]);
static void RotateRoundKeyRight(bool roundKey[56]);
static void ComputeIP(bool L[32], bool R[32], bool inBlk[64]);
static void ComputeFP(bool outBlk[64], bool L[32], bool R[32]);
static void ComputeF(bool fout[32], bool R[32], bool roundKey[56]);
static void ComputeP(bool output[32], bool input[32]);
static void ComputeS_Lookup(int k, bool output[4], bool input[6]);
static void ComputePC2(bool subkey[48], bool roundKey[56]);
static void ComputeExpansionE(bool expandedBlock[48], bool R[32]);
static void DumpBin(char *str, bool *b, int bits);
static void Exchange_L_and_R(bool L[32], bool R[32]);

static int EnableDumpBin = 0;



/**********************************************************************/
/*                                                    */
/*                    DES TABLES                      */
/*                                                    */
/**********************************************************************/


/*
 * IP: Output bit table_DES_IP[i] equals input bit i.
 */
static int table_DES_IP[64] = {
   39, 7, 47, 15, 55, 23, 63, 31,
   38, 6, 46, 14, 54, 22, 62, 30,
   37, 5, 45, 13, 53, 21, 61, 29,
   36, 4, 44, 12, 52, 20, 60, 28,
   35, 3, 43, 11, 51, 19, 59, 27,
   34, 2, 42, 10, 50, 18, 58, 26,
   33, 1, 41, 9, 49, 17, 57, 25,
   32, 0, 40, 8, 48, 16, 56, 24
};


/*
 * FP: Output bit table_DES_FP[i] equals input bit i.
 */
static int table_DES_FP[64] = {
   57, 49, 41, 33, 25, 17, 9, 1,
   59, 51, 43, 35, 27, 19, 11, 3,
   61, 53, 45, 37, 29, 21, 13, 5,
   63, 55, 47, 39, 31, 23, 15, 7,
   56, 48, 40, 32, 24, 16, 8, 0,
   58, 50, 42, 34, 26, 18, 10, 2,
   60, 52, 44, 36, 28, 20, 12, 4,
   62, 54, 46, 38, 30, 22, 14, 6
};


/*
 * PC1: Permutation choice 1, used to pre-process the key
 */
static int table_DES_PC1[56] = {
   27, 19, 11, 31, 39, 47, 55,
   26, 18, 10, 30, 38, 46, 54,
   25, 17, 9, 29, 37, 45, 53,
   24, 16, 8, 28, 36, 44, 52,
   23, 15, 7, 3, 35, 43, 51,
   22, 14, 6, 2, 34, 42, 50,
   21, 13, 5, 1, 33, 41, 49,
   20, 12, 4, 0, 32, 40, 48
};


/*
 * PC2: Map 56-bit round key to a 48-bit subkey
 */
static int table_DES_PC2[48] = {
   24, 27, 20, 6, 14, 10, 3, 22,
    0, 17, 7, 12, 8, 23, 11, 5,
   16, 26, 1, 9, 19, 25, 4, 15,
   54, 43, 36, 29, 49, 40, 48, 30,
   52, 44, 37, 33, 46, 35, 50, 41,
   28, 53, 51, 55, 32, 45, 39, 42
};


/*
 * E: Expand 32-bit R to 48 bits.
 */
static int table_DES_E[48] = {
   31, 0, 1, 2, 3, 4, 3, 4,
    5, 6, 7, 8, 7, 8, 9, 10,
   11, 12, 11, 12, 13, 14, 15, 16,
   15, 16, 17, 18, 19, 20, 19, 20,
   21, 22, 23, 24, 23, 24, 25, 26,
   27, 28, 27, 28, 29, 30, 31, 0
};


/*
 * P: Permutation of S table outputs
 */
static int table_DES_P[32] = {
   11, 17, 5, 27, 25, 10, 20, 0,
   13, 21, 3, 28, 29, 7, 18, 24,
   31, 22, 12, 6, 26, 2, 16, 8,
   14, 30, 4, 19, 1, 9, 15, 23
};


/*
 * S Tables: Introduce nonlinearity and avalanche
 */
static int table_DES_S[8][64] = {
   /* table S[0] */
      {  13, 1, 2, 15, 8, 13, 4, 8, 6, 10, 15, 3, 11, 7, 1, 4,
         10, 12, 9, 5, 3, 6, 14, 11, 5, 0, 0, 14, 12, 9, 7, 2,
          7, 2, 11, 1, 4, 14, 1, 7, 9, 4, 12, 10, 14, 8, 2, 13,
          0, 15, 6, 12, 10, 9, 13, 0, 15, 3, 3, 5, 5, 6, 8, 11 },
   /* table S[1] */
      {   4, 13, 11, 0, 2, 11, 14, 7, 15, 4, 0, 9, 8, 1, 13, 10,
          3, 14, 12, 3, 9, 5, 7, 12, 5, 2, 10, 15, 6, 8, 1, 6,
          1, 6, 4, 11, 11, 13, 13, 8, 12, 1, 3, 4, 7, 10, 14, 7,
         10, 9, 15, 5, 6, 0, 8, 15, 0, 14, 5, 2, 9, 3, 2, 12 },
   /* table S[2] */
      {  12, 10, 1, 15, 10, 4, 15, 2, 9, 7, 2, 12, 6, 9, 8, 5,
          0, 6, 13, 1, 3, 13, 4, 14, 14, 0, 7, 11, 5, 3, 11, 8,
          9, 4, 14, 3, 15, 2, 5, 12, 2, 9, 8, 5, 12, 15, 3, 10,
          7, 11, 0, 14, 4, 1, 10, 7, 1, 6, 13, 0, 11, 8, 6, 13 },
   /* table S[3] */
      {   2, 14, 12, 11, 4, 2, 1, 12, 7, 4, 10, 7, 11, 13, 6, 1,
          8, 5, 5, 0, 3, 15, 15, 10, 13, 3, 0, 9, 14, 8, 9, 6,
          4, 11, 2, 8, 1, 12, 11, 7, 10, 1, 13, 14, 7, 2, 8, 13,
         15, 6, 9, 15, 12, 0, 5, 9, 6, 10, 3, 4, 0, 5, 14, 3 },
   /* table S[4] */
      {   7, 13, 13, 8, 14, 11, 3, 5, 0, 6, 6, 15, 9, 0, 10, 3,
          1, 4, 2, 7, 8, 2, 5, 12, 11, 1, 12, 10, 4, 14, 15, 9,
         10, 3, 6, 15, 9, 0, 0, 6, 12, 10, 11, 1, 7, 13, 13, 8,
         15, 9, 1, 4, 3, 5, 14, 11, 5, 12, 2, 7, 8, 2, 4, 14 },
   /* table S[5] */
      {  10, 13, 0, 7, 9, 0, 14, 9, 6, 3, 3, 4, 15, 6, 5, 10,
          1, 2, 13, 8, 12, 5, 7, 14, 11, 12, 4, 11, 2, 15, 8, 1,
         13, 1, 6, 10, 4, 13, 9, 0, 8, 6, 15, 9, 3, 8, 0, 7,
         11, 4, 1, 15, 2, 14, 12, 3, 5, 11, 10, 5, 14, 2, 7, 12 },
   /* table S[6] */
```

```
     {  15, 3, 1, 13, 8, 4, 14, 7, 6, 15, 11, 2, 3, 8, 4, 14,
         9, 12, 7, 0, 2, 1, 13, 10, 12, 6, 0, 9, 5, 11, 10, 5,
         0, 13, 14, 8, 7, 10, 11, 1, 10, 3, 4, 15, 13, 4, 1, 2,
         5, 11, 8, 6, 12, 7, 6, 12, 9, 0, 3, 5, 2, 14, 15, 9 },
   /* table S[7] */
     {  14, 0, 4, 15, 13, 7, 1, 4, 2, 14, 15, 2, 11, 13, 8, 1,
         3, 10, 10, 6, 6, 12, 12, 11, 5, 9, 9, 5, 0, 3, 7, 8,
         4, 15, 1, 12, 14, 8, 8, 2, 13, 4, 6, 9, 2, 1, 11, 7,
        15, 5, 12, 11, 9, 3, 7, 14, 3, 10, 10, 0, 5, 6, 0, 13 }
};




/**********************************************************************/
/*                                                  */
/*                    DES CODE                      */
/*                                                  */
/**********************************************************************/



/*
 * EncryptDES: Encrypt a block using DES. Set verbose for debugging info.
 * (This loop does both loops on the "DES Encryption" page of the flowchart.)
 */
void EncryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose) {
 int i,round;
 bool R[32], L[32], fout[32];
 bool roundKey[56];

 //int x;
 //for(x = 0; x < 100000; x++);


 EnableDumpBin = verbose;        /* set debugging on/off flag */
 DumpBin("input(left)", inBlk+32, 32);
 DumpBin("input(right)", inBlk, 32);
 DumpBin("raw key(left )", key+28, 28);
 DumpBin("raw key(right)", key, 28);

 /* Compute the first roundkey by performing PC1 */
 ComputeRoundKey(roundKey, key);

 DumpBin("roundKey(L)", roundKey+28, 28);
 DumpBin("roundKey(R)", roundKey, 28);

 /* Compute the initial permutation and divide the result into L and R */
 ComputeIP(L,R,inBlk);

 DumpBin("after IP(L)", L, 32);
 DumpBin("after IP(R)", R, 32);

 for (round = 0; round < 16; round++) {
   if (verbose)
     printf("-------------- BEGIN ENCRYPT ROUND %d -------------\n", round);
   DumpBin("round start(L)", L, 32);
   DumpBin("round start(R)", R, 32);

   /* Rotate roundKey halves left once or twice (depending on round) */
   RotateRoundKeyLeft(roundKey);
   if (round != 0 && round != 1 && round != 8 && round != 15)
     RotateRoundKeyLeft(roundKey);
   DumpBin("roundKey(L)", roundKey+28, 28);
   DumpBin("roundKey(R)", roundKey, 28);

   /* Compute f(R, roundKey) and exclusive-OR onto the value in L */
   ComputeF(fout, R, roundKey);
   DumpBin("f(R,key)", fout, 32);
   for (i = 0; i < 32; i++)
     L[i] ^= fout[i];
   DumpBin("L^f(R,key)", L, 32);

   Exchange_L_and_R(L,R);

   DumpBin("round end(L)", L, 32);
   DumpBin("round end(R)", R, 32);
   if (verbose)
     printf("--------------- END ROUND %d --------------\n", round);
 }

 Exchange_L_and_R(L,R);

 /* Combine L and R then compute the final permutation */
 ComputeFP(outBlk,L,R);
 DumpBin("FP out( left)", outBlk+32, 32);
 DumpBin("FP out(right)", outBlk, 32);
}



/*
 * DecryptDES: Decrypt a block using DES. Set verbose for debugging info.
 * (This loop does both loops on the "DES Decryption" page of the flowchart.)
 */
void DecryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose) {
 int i,round;
 bool R[32], L[32], fout[32];
 bool roundKey[56];

 EnableDumpBin = verbose;        /* set debugging on/off flag */
 DumpBin("input(left)", inBlk+32, 32);
 DumpBin("input(right)", inBlk, 32);
 DumpBin("raw key(left )", key+28, 28);
 DumpBin("raw key(right)", key, 28);

 /* Compute the first roundkey by performing PC1 */
 ComputeRoundKey(roundKey, key);

 DumpBin("roundKey(L)", roundKey+28, 28);
 DumpBin("roundKey(R)", roundKey, 28);

 /* Compute the initial permutation and divide the result into L and R */
 ComputeIP(L,R,inBlk);

 DumpBin("after IP(L)", L, 32);
 DumpBin("after IP(R)", R, 32);

 for (round = 0; round < 16; round++) {
  if (verbose)
   printf("-------------- BEGIN DECRYPT ROUND %d -------------\n", round);
  DumpBin("round start(L)", L, 32);
  DumpBin("round start(R)", R, 32);

  /* Compute f(R, roundKey) and exclusive-OR onto the value in L */
  ComputeF(fout, R, roundKey);
  DumpBin("f(R,key)", fout, 32);
  for (i = 0; i < 32; i++)
   L[i] ^= fout[i];
  DumpBin("L^f(R,key)", L, 32);

  Exchange_L_and_R(L,R);

  /* Rotate roundKey halves right once or twice (depending on round) */
  DumpBin("roundKey(L)", roundKey+28, 28); /* show keys before shift */
  DumpBin("roundKey(R)", roundKey, 28);
  RotateRoundKeyRight(roundKey);
  if (round != 0 && round != 7 && round != 14 && round != 15)
   RotateRoundKeyRight(roundKey);

  DumpBin("round end(L)", L, 32);
  DumpBin("round end(R)", R, 32);
  if (verbose)
   printf("--------------- END ROUND %d --------------\n", round);
 }

 Exchange_L_and_R(L,R);

 /* Combine L and R then compute the final permutation */
 ComputeFP(outBlk,L,R);
 DumpBin("FP out( left)", outBlk+32, 32);
 DumpBin("FP out(right)", outBlk, 32);
}



/*
 * ComputeRoundKey: Compute PC1 on the key and store the result in roundKey
 */
static void ComputeRoundKey(bool roundKey[56], bool key[56]) {
 int i;

 for (i = 0; i < 56; i++)
  roundKey[table_DES_PC1[i]] = key[i];
}



/*
 * RotateRoundKeyLeft: Rotate each of the halves of roundKey left one bit
 */
static void RotateRoundKeyLeft(bool roundKey[56]) {
 bool temp1, temp2;
 int i;

 temp1 = roundKey[27];
 temp2 = roundKey[55];
 for (i = 27; i >= 1; i--) {
  roundKey[i] = roundKey[i-1];
  roundKey[i+28] = roundKey[i+28-1];
 }
 roundKey[ 0] = temp1;
 roundKey[28] = temp2;
}



/*
 * RotateRoundKeyRight: Rotate each of the halves of roundKey right one bit
 */
static void RotateRoundKeyRight(bool roundKey[56]) {
 bool temp1, temp2;
 int i;

 temp1 = roundKey[0];
 temp2 = roundKey[28];
 for (i = 0; i < 27; i++) {
  roundKey[i] = roundKey[i+1];
  roundKey[i+28] = roundKey[i+28+1];
 }
 roundKey[27] = temp1;
 roundKey[55] = temp2;
}



/*
 * ComputeIP: Compute the initial permutation and split into L and R halves.
 */
static void ComputeIP(bool L[32], bool R[32], bool inBlk[64]) {
 bool output[64];
 int i;
 /* Permute
  */
 for (i = 63; i >= 0; i--)
  output[table_DES_IP[i]] = inBlk[i];

 /* Split into R and L. Bits 63..32 go in L, bits 31..0 go in R.
  */
 for (i = 63; i >= 0; i--) {
  if (i >= 32)
   L[i-32] = output[i];
  else
   R[i] = output[i];
 }
}



/*
 * ComputeFP: Combine the L and R halves and do the final permutation.
 */
static void ComputeFP(bool outBlk[64], bool L[32], bool R[32]) {
 bool input[64];
 int i;

 /* Combine L and R into input[64]
  */
 for (i = 63; i >= 0; i--)
  input[i] = (i >= 32) ? L[i - 32] : R[i];

 /* Permute
  */
 for (i = 63; i >= 0; i--)
  outBlk[table_DES_FP[i]] = input[i];
}



/*
 * ComputeF: Compute the DES f function and store the result in fout.
 */
static void ComputeF(bool fout[32], bool R[32], bool roundKey[56]) {
 bool expandedBlock[48], subkey[48], sout[32];
 int i,k;

 /* Expand R into 48 bits using the E expansion */
 ComputeExpansionE(expandedBlock, R);
 DumpBin("expanded E", expandedBlock, 48);

 /* Convert the roundKey into the subkey using PC2 */
 ComputePC2(subkey, roundKey);
 DumpBin("subkey", subkey, 48);

 /* XOR the subkey onto the expanded block */
 for (i = 0; i < 48; i++)
   expandedBlock[i] ^= subkey[i];

 /* Divide expandedBlock into 6-bit chunks and do S table lookups */
 for (k = 0; k < 8; k++)
   ComputeS_Lookup(k, sout+4*k, expandedBlock+6*k);

 /* To complete the f() calculation, do permutation P on the S table output */
 ComputeP(fout, sout);
}



/*
 * ComputeP: Compute the P permutation on the S table outputs.
 */
static void ComputeP(bool output[32], bool input[32]) {
 int i;

 for (i = 0; i < 32; i++)
   output[table_DES_P[i]] = input[i];
}



/*
 * Look up a 6-bit input in S table k and store the result as a 4-bit output.
 */
static void ComputeS_Lookup(int k, bool output[4], bool input[6]) {
 int inputValue, outputValue;

 /* Convert the input bits into an integer */
 inputValue = input[0] + 2*input[1] + 4*input[2] + 8*input[3] +
       16*input[4] + 32*input[5];

 /* Do the S table lookup */
 outputValue = table_DES_S[k][inputValue];

 /* Convert the result into binary form */
 output[0] = (outputValue & 1) ? 1 : 0;
 output[1] = (outputValue & 2) ? 1 : 0;
 output[2] = (outputValue & 4) ? 1 : 0;
 output[3] = (outputValue & 8) ? 1 : 0;
}



/*
 * ComputePC2: Map a 56-bit round key onto a 48-bit subkey
 */
static void ComputePC2(bool subkey[48], bool roundKey[56]) {
 int i;
 for (i = 0; i < 48; i++)
   subkey[i] = roundKey[table_DES_PC2[i]];
}




/*
 * ComputeExpansionE: Compute the E expansion to prepare to use S tables.
 */
static void ComputeExpansionE(bool expandedBlock[48], bool R[32]) {
 int i;

 for (i = 0; i < 48; i++)
   expandedBlock[i] = R[table_DES_E[i]];
}




/*
 * Exchange_L_and_R: Swap L and R
 */
static void Exchange_L_and_R(bool L[32], bool R[32]) {
 int i;

 for (i = 0; i < 32; i++)
   L[i] ^= R[i] ^= L[i] ^= R[i];   /* exchanges L[i] and R[i] */
}




/*
 * DumpBin: Display intermediate values if emableDumpBin is set.
 */
static void DumpBin(char *str, bool *b, int bits) {
 int i;

 if ((bits % 4)!=0 || bits>48) {
   printf("Bad call to DumpBin (bits > 48 or bit len not a multiple of 4\n");
   exit(1);
 }

 if (EnableDumpBin) {
   for (i = strlen(str); i < 14; i++)
     printf(" ");
   printf("%s: ", str);
   for (i = bits-1; i >= 0; i--)
     printf("%d", b[i]);
   //printf(" ");
   printf("\n");
   /*for (i = bits; i < 48; i++)
     printf(" ");
   printf("(");
   for (i = bits-4; i >= 0; i-=4)
```

```
    printf("%X", b[i]+2*b[i+1]+4*b[i+2]+8*b[i+3]);
  printf(")\n");*/
  }
}
```

F.2 Output application

When mapped over three of the processors of the target architecture, 12 output files are produced by the Anvil refactoring engine:

File	Language	Purpose
_anvil_settings_cpu0.h	C header	Configures the Anvil pthreads library
_anvil_settings_cpu1.h	C header	Configures the Anvil pthreads library
_anvil_settings_cpu2.h	C header	Configures the Anvil pthreads library
_anvil_specific_cpu0.c	C code	Low-level hardware drivers
_anvil_specific_cpu1.c	C code	Low-level hardware drivers
_anvil_specific_cpu2.c	C code	Low-level hardware drivers
linkscript.ld		LD link script
microblaze_0.c	C code	Code for CPU 0
microblaze_1.c	C code	Code for CPU 1
microblaze_1.c	C code	Code for CPU 2
des.c	C code	(As input, not shown)
des.h	C header	(As input, not shown)

F.2.1 _anvil_settings_cpu0.h

```
#define _ANVIL_THISTHREADID 0, 1
#define _ANVIL_TOTAL_MANAGED_THREADS 4
#define _ANVIL_TOTAL_MANAGED_MUTEXES 2
#define _ANVIL_TOTAL_MANAGED_VARS 1
#define _ANVIL_TOTAL_MANAGED_CVS 0
#define _ANVIL_TOTAL_ACCESSED_VARS 1
#define _ANVIL_TOTAL_SYSTEM_THREADS 4
#define _ANVIL_TOTAL_SYSTEM_MUTEXES 2
#define _ANVIL_TOTAL_SYSTEM_CVS 0
#define _ANVIL_MANAGED_THREADS_INITIALISER {0, 0, 0}, {1, 0, 0}, {2, 0, 0}, {3, 0, 0}
#define _ANVIL_MANAGED_MUTEXES_INITIALISER {0, 0, 0}, {1, 0, 0}
#define _ANVIL_MANAGED_CVS_INITIALISER
#define _ANVIL_MANAGED_VARS_INITIALISER {0, 0, 4}
#define _ANVIL_ACCESSED_VARS_INITIALISER {0, 0}
#define _ANVIL_CIRC_BUFF_SIZE 8
#define _ANVIL_MSG_IRQS 1
#define _ANVIL_MSG_IRQS_INITIALISER {0, 0, -1}
#define _ANVIL_ACCS 0
#define _ANVIL_ACCS_INITIALISER {0, 0}
```

F.2.2 _anvil_settings_cpu1.h

```
#define _ANVIL_THISTHREADID 2
#define _ANVIL_TOTAL_MANAGED_THREADS 0
#define _ANVIL_TOTAL_MANAGED_MUTEXES 0
#define _ANVIL_TOTAL_MANAGED_VARS 0
#define _ANVIL_TOTAL_MANAGED_CVS 0
#define _ANVIL_TOTAL_ACCESSED_VARS 1
#define _ANVIL_TOTAL_SYSTEM_THREADS 4
#define _ANVIL_TOTAL_SYSTEM_MUTEXES 2
#define _ANVIL_TOTAL_SYSTEM_CVS 0
#define _ANVIL_MANAGED_THREADS_INITIALISER
#define _ANVIL_MANAGED_MUTEXES_INITIALISER
#define _ANVIL_MANAGED_CVS_INITIALISER
#define _ANVIL_MANAGED_VARS_INITIALISER
#define _ANVIL_ACCESSED_VARS_INITIALISER {0, 0}
#define _ANVIL_CIRC_BUFF_SIZE 8
#define _ANVIL_MSG_IRQS 1
#define _ANVIL_MSG_IRQS_INITIALISER {0, 0, -1}
#define _ANVIL_ACCS 0
#define _ANVIL_ACCS_INITIALISER {0, 0}
```

F.2.3 _anvil_settings_cpu2.h

Identical to `_anvil_settings_cpu1.h` apart from the definition of `_ANVIL_THISTHREADID` which is set to 3.

F.2.4 _anvil_specific_cpu0.c

```
#include "anvil_pthreads.h"
#include "mblib.h"

volatile int *mbox1 = (int *)0x84000000;
volatile int *mbox2 = (int *)0x85000000;
volatile int *intc0 = (int *)0x86000000;

void _anvil_send_to_thread(int targetthreadid, int *packet, int len)
{
 int x;

 for (x = 0; x < len; x++)
 {
  switch(targetthreadid)
  {
   case 0:
    break;
   case 1:
    break;
   case 2:
    mbox_write(mailbox1, packet[x]);
    break;
   case 3:
    mbox_write(mailbox2, packet[x]);
    break;
  }
 }
}

void _anvil_interrupts_disable(void)
{
 mb_disable_interrupts();
}

void _anvil_interrupts_enable(void)
{
 mb_enable_interrupts();
}

int _anvil_read_from_interrupt(int vec)
{
 switch(vec)
 {
  case 0:
   //mbox
   return mbox_read(mailbox_0);
   break;
  default:
   //Unknown interrupt, disable it
   intc_disable_interrupt(intc_0, vec);
   return 0;
   break;
 }
}

int _anvil_interrupt_ready(int vec)
{
 switch(vec)
 {
  case 0:
   //mbox1
   return mbox_check(mbox1);
   break;
  case 1:
   //mbox2
   return mbox_check(mbox2);
   break;
  default:
   //Unknown interrupt, disable it
   intc_disable_interrupt(intc0, vec);
   return 0;
   break;
 }
}

int _anvil_get_interrupt_vector(void)
{
 return intc_get_vector(intc0);
}

void _anvil_acknowledge_interrupt(int vec)
{
 intc_acknowledge_interrupt(intc0, vec);
}

void _anvil_sleep(void)
{
 //nop
}
```

F.2.5 _anvil_specific_cpu1.c

```
#include "anvil_pthreads.h"
#include "mblib.h"

volatile int *mbox = (int *)0x84000000;

void _anvil_send_to_thread(int targetthreadid, int *packet, int len)
{
 int x;

 for (x = 0; x < len; x++)
 {
  switch(targetthreadid)
  {
   case 0:
    mbox_write(mbox, packet[x]);
    break;
   case 1:
    break;
   case 2:
    break;
   case 3:
    break;
  }
 }
}

void _anvil_interrupts_disable(void)
{
 mb_disable_interrupts();
}

void _anvil_interrupts_enable(void)
{
 mb_enable_interrupts();
}

int _anvil_read_from_interrupt(int vec)
{
 switch(vec)
 {
  case 0:
   //mbox
   return mbox_read(mbox);
   break;
  default:
   //Unknown interrupt, disable it
   intc_disable_interrupt(intc_0, vec);
   return 0;
   break;
 }
}

int _anvil_interrupt_ready(int vec)
{
 switch(vec)
 {
  case 0:
   //mbox
   return mbox_check(mbox);
   break;
  default:
   return 0;
   break;
 }
}

int _anvil_get_interrupt_vector(void)
{
 return 0;
}

void _anvil_acknowledge_interrupt(int vec)
{
 //nop
}

void _anvil_sleep(void)
{
 //nop
}
```

F.2.6 _anvil_specific_cpu2.c

Identical to `_anvil_specific_cpu1.c`.

F.2.7 linkscript.ld

Identical to `linkscript.ld` from appendix D.

F.2.8 microblaze_0.c

```
#include "anvil_pthreads.h"
#include "des.h"

#define BLOCKS 5000
#define BLOCKSIZE 64
#define WORKER_THREADS 3

//Prototypes
void *worker_body(void *a);
void outputblock(bool *block, int blocknum);
void getblock(bool *block, int blockno);

bool outputdata[BLOCKS][BLOCKSIZE] __attribute__((section ("sharedmemory")));;

//------------------------------------------------------------

//Main function
int main(int argc, char *argv[]) {
 int x;

 //Enable interrupts (microblaze, intc)
 extern volatile int * intc;
 intc_enable_all_interrupts(intc);
 intc_master_enable(intc);
 mb_enable_interrupts();

 //Create worker threads
 pthread_create(1, 0, 1); //Thread ID 1, OM CPU ID 0, Arg 1
 pthread_create(2, 0, 2); //Thread ID 2, OM CPU ID 0, Arg 2
 pthread_create(3, 0, 3); //Thread ID 3, OM CPU ID 0, Arg 3

 //Join workers
 for(i = 0; i < WORKER_THREADS; i++)
 pthread_join(1, 0); //Thread ID 1, OM CPU ID 0
 pthread_join(2, 0); //Thread ID 2, OM CPU ID 0
 pthread_join(3, 0); //Thread ID 3, OM CPU ID 0

 return 0;
}

//------------------------------------------------------------

//Body function for the workers
//Should be passed an integer to give it an ID
void *worker_body(void *a) {
 int blockno, x;
 bool inblock[BLOCKSIZE], outblock[BLOCKSIZE];

 //3DES keys, distributed to each thread
 bool key1[56] = {
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0};
  bool key2[56] = {
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1};
  bool key3[56] = {
    1,1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,
    1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,
    1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,
    1,0,0,0,0,0,0,0};

  //Read the ID
  num = _anvil_get_int_threadarg();

  //Begin encryption loop
  for(blockno = num; blockno < BLOCKS; blockno += WORKER_THREADS) {
    //Grab the mutex for the input stream
    pthread_mutex_lock(0, 0); //Mutex ID 0, OM CPU ID 0
    //Get the block
    getblock(inblock, num);
    //Release the mutex
    pthread_mutex_unlock(0, 0); //Mutex ID 0, OM CPU ID 0

    //Perform 3DES
    EncryptDES(key1, outblock, inblock, 0);
    EncryptDES(key2, inblock, outblock, 0);
    EncryptDES(key3, outblock, inblock, 0);

    //Grab the mutex for the output stream
    pthread_mutex_lock(1, 0); //Mutex ID 1, OM CPU ID 0
    //Write the block out
    outputblock(outblock, blockno);
    //Release the mutex
    pthread_mutex_unlock(1, 0); //Mutex ID 1, OM CPU ID 0
  }

  pthread_exit(1, 0); //Thread ID 1, OM CPU ID 0
}

//------------------------------------------------------------

void getblock(bool *block, int blockno) {
  //Implementation omitted. Could be read from memory or an external I/O device
}

void outputblock(bool *block, int blocknum) {
  //Implementation omitted. Could be written to memory or an external I/O device
}
```

F.2.9 microblaze_1.c

```c
#include "anvil_pthreads.h"
#include "des.h"

#define BLOCKS 5000
#define BLOCKSIZE 64
#define WORKER_THREADS 3

//Prototypes
void *worker_body(void *a);
void outputblock(bool *block, int blocknum);
void getblock(bool *block, int blockno);

bool outputdata[BLOCKS][BLOCKSIZE] __attribute__((section ("sharedmemory")));;

//------------------------------------------------------------

//Main function
int main(int argc, char *argv[]) {
 //Enable interrupts (microblaze)
 mb_enable_interrupts();

 _anvil_wait_until_released();
}

//------------------------------------------------------------

//Body function for the workers
//Should be passed an integer to give it an ID
void *worker_body(void *a) {
 int blockno, x;
 bool inblock[BLOCKSIZE], outblock[BLOCKSIZE];

 //3DES keys, distributed to each thread
 bool key1[56] = {
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0};
 bool key2[56] = {
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1};
 bool key3[56] = {
  1,1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,
  1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,
  1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,
  1,0,0,0,0,0,0,0};

 //Read the ID
 num = _anvil_get_int_threadarg();
 //Begin encryption loop
 for(blockno = num; blockno < BLOCKS; blockno += WORKER_THREADS) {
  //Grab the mutex for the input stream
  pthread_mutex_lock(0, 0); //Mutex ID 0, OM CPU ID 0
  //Get the block
  getblock(inblock, num);
  //Release the mutex
  pthread_mutex_unlock(0, 0); //Mutex ID 0, OM CPU ID 0

  //Perform 3DES
  EncryptDES(key1, outblock, inblock, 0);
  EncryptDES(key2, inblock, outblock, 0);
  EncryptDES(key3, outblock, inblock, 0);

  //Grab the mutex for the output stream
  pthread_mutex_lock(1, 0); //Mutex ID 1, OM CPU ID 0
  //Write the block out
  outputblock(outblock, blockno);
  //Release the mutex
  pthread_mutex_unlock(1, 0); //Mutex ID 1, OM CPU ID 0
 }

 pthread_exit(2, 0); //Thread ID 1, OM CPU ID 0
}

//-----------------------------------------------------------

void getblock(bool *block, int blockno) {
 //Implementation omitted. Could be read from memory or an external I/O device
}

void outputblock(bool *block, int blocknum) {
 //Implementation omitted. Could be written to memory or an external I/O device
}
```

F.2.10 microblaze_2.c

Identical to `microblaze_1.c`.

Bibliography

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. *SIGARCH Comput. Archit. News*, 16(2):280–298, 1988.

[2] Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, 2003.

[3] Jason Agron and David Andrews. Building heterogeneous reconfigurable systems with a hardware microkernel. In *Proceedings of CODES+ISSS '09*, pages 393–402, New York, NY, USA, 2009. ACM.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] Robert Allen and David Garlan. The Wright architectural specification language. Technical report, Carnegie Mellon School of Computer Science, 1996.

[6] Altera Corporation. Stratix IV device family overview. http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/stxiv-index.jsp, Nov 2009.

[7] Craig Anderson and Jean-Loup Baer. Two techniques for improving performance on bus-based multiprocessors. *Future Generation Computer Systems*, 11(6):537 – 551, 1995. High-Performance Computer Architecture.

[8] Jose N Arabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Mike Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical report, Pittsburgh, PA, USA, 1995.

[9] Ken Arnold and James Gosling. *The Java programming language (2nd ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[10] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM.

[11] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02*, pages 73–78, 2002.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.

[13] William C Barker. Recommendation for the triple data encryption algorithm (TDEA) block cipher, nist special publication; 800-67., May 2004.

[14] Greg Barnes. A method for implementing lock-free shared-data structures. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270, New York, NY, USA, 1993. ACM.

[15] J. Barnes. *Programming in Ada95*. Addison Wesley, 1995.

[16] Jonn Barnes, editor. *Ada 95 Rationale - Annex E - Distributed Systems*. Springer-Verlag, 1995.

[17] Baumann et al. The Multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of SOSP '09*, pages 29–44, New York, NY, USA, 2009. ACM.

[18] Marc Baumslag and Fred Annexstein. A unified framework for off-line permutation routing in parallel networks. *Theory of Computing Systems*, 24 - 1:233–251, 1990.

[19] David Beazley. Ply (python-lex-yacc). http://www.dabeaz.com/ply/, Accessed Dec 2009.

[20] J. Becker, M. Hubner, K. D. Muller-Glaser, R. Constapel, J. Luka, and J. Eisenmann. Automotive control unit optimization perspectives: Body functions on-demand by dynamic reconfiguration. In *Design, Automation and Test Eur. Conf. Exhibition (DATE 2005)*, 2005.

[21] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[22] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *Design & Test of Computers, IEEE*, 17(2):74–85, Apr-Jun 2000.

[23] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *ICFP*, pages 174–184, 1998.

[24] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. *Proceedings of the workshop on languages and compilers for parallel computing*, pages 10–1, 1994.

[25] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide, Second Edition*. Addison Wesley, 2005.

[26] M. Bowen. *Handel-C Language Reference Manual, 2.1 edition*. Embedded Solutions Limited, 1998.

[27] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-programmable gate arrays*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[28] R. Brukardt. The Ada95 language reference manual - Appendix E, Distributed Systems (International Standard ISO/IEC 8652:1995). http://www.adaic.org/standards/95lrm/html/RM-E.html.

[29] Alex Buckley. JSR 201: Extending the Java programming language with enumerations, autoboxing, enhanced for loops and static import. http://jcp.org/en/jsr/detail?id=201, Sep 2004.

[30] Alan Burns, Brian Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Ada-Europe '98*, pages 263–275. Springer-Verlag, 1998.

[31] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[32] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti. Transaction-level models for AMBA bus architecture using SystemC 2.0. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 20026, Washington, DC, USA, 2003. IEEE Computer Society.

[33] Joco M. P. Cardoso and Horacio C. Neto. Macro-based hardware compilation of Java(tm) bytecodes into a dynamic reconfigurable computing system. In *FCCM '99*, Washington, DC, USA, 1999. IEEE Computer Society.

[34] William W. Carlson, David E. Culler, and Eugene Brooks. Introduction to UPC and language specification. *CCS-TR-99-157*, 1999.

[35] W. Cesario et al. Component-based design approach for multicore SoCs. *DAC*, 00:789, 2002.

[36] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

[37] Sayantan Chakravorty, Celso Mendes, and Laxmikant Kal. Proactive fault tolerance in MPI applications via task migration. In *High Performance Computing - HiPC 2006*, volume 4297 of *Lecture Notes in Computer Science*, pages 485–496. Springer Berlin / Heidelberg, 2006.

[38] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[39] Robit Chandra et al. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[40] Jyh-Biau Chang, Ce-Kuen Shieh, and Tyng-Yeu Liang. A transparent distributed shared memory for clustered symmetric multiprocessors. *The Journal of Supercomputing*, 37(2):145–160, 2006.

[41] Daniel Marcos Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford Univ., CA., 1984.

[42] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.

[43] Jr. Charles H. Roth. *Digital systems design using VHDL*. Pws Pub. Co., 1998.

[44] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: parallel programming on a network of multiprocessors. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 147–158, New York, NY, USA, 1989. ACM.

[45] Liqun Cheng, J.B. Carter, and Donglai Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 328–339, Feb. 2007.

[46] Shigeru Chiba. A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, October 1995.

[47] Kevin Chiew and Yingjiu Li. Multistage off-line permutation packet routing on a mesh: An approach with elementary mathematics. *Journal of Computer Science and Technology*, 24 - 1:175–180, 2009.

[48] David Chisnall. *The definitive guide to the Xen hypervisor*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[49] Paul C. Clements. A survey of architecture description languages. In *IWSSD '96*. IEEE Computer Society, 1996.

[50] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[51] Katherine Compton, Zhiyuan Li, James Cooley, Stephen Knol, and Scott Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):209–220, 2002.

[52] Simone Corbetta, Massimo Morandi, Marco Novati, Marco Domenico Santambrogio, Donatella Sciuto, and Paola Spoletini. Internal and external bitstream relocation for partial dynamic reconfiguration. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17-11:1650–1654, 2009.

[53] Microsoft Corporation. Windows Virtual PC. http://www.microsoft.com/windows/virtual-pc/ - Accessed Dec 2009, 2009.

[54] CoWare, Inc. CoWare Virtual Platform - hardware/software integration and testing...without hardware. http://www.coware.com/products/virtualplatform.php (Accessed Aug 09).

[55] A.L. Cox and R.J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on*, pages 98–108, May 1993.

[56] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research & Development*, 25:483–490, 1981.

[57] Culler, Dusseau, Goldstein, Krishnamurthy, Lumetta, von Eicken, and Yelick. Parallel programming in Split-C. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, volume 0, pages 262–273, Los Alamitos, CA, USA, 1993. IEEE Computer Society.

[58] P. Cumming. *The TI OMAP Platform Approach to SoC*. Kluwer Academic Publ, Boston, MA., 2003.

[59] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. *DAC*, 2001.

[60] Francois-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[61] P.C. Dibble and A.J. Wellings. JSR-282 status report. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM International Conference Proceeding Series, pages 179–182, New York, NY, USA, 2009. ACM.

[62] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[63] Alastair Donaldson, Colin Riley, Anton Lokhmotov, and Andrew Cook. Auto-parallelisation of Sieve C++ programs. *Lecture Notes in Computer Science*, 4854/2008:18–27, 2008.

[64] Doulos. The designer's guide to Verilog. Date retrieved: January, 2007.

[65] Bernd Dreier, Markus Zahn, and Theo Ungerer. The Rthreads distributed shared memory system. In *3rd Int. Conf. on Massively Parallel Computing Systems*, 1998.

[66] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - reconfigurable pipelined datapath. In *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135, London, UK, 1996. Springer-Verlag.

[67] S. A. Edwards. Design and verification languages - tech. report CUCS-046-04. Technical report, Dept. of Computer Science, Columbia University, 2004.

[68] Ásgeir Th. Eiríksson and Kenneth L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 367–380, London, UK, 1995. Springer-Verlag.

[69] Electronic Industries Association. Electronic Design Interchange Format version 2.0.0 - technical report ANSI/EIA-548-1988, 1988.

[70] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. Hardware/software partitioning with iterative improvement heuristics. In *ISSS '96: Proceedings of the 9th international symposium on System synthesis*, page 71, Washington, DC, USA, 1996. IEEE Computer Society.

[71] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. System level hardware/-software partitioning based on simulated annealing and Tabu search. *Design Automation for Embedded Systems*, 2:5–32, 1997.

[72] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[73] Natalie D. Enright Jerger, Li-Shiuan Peh, and Mikko H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 35–46, Washington, DC, USA, 2008. IEEE Computer Society.

[74] Cagkan Erbas, Selin C. Erbas, and Andy D. Pimentel. A multiobjective optimization model for exploring multiprocessor mappings of process networks. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 182–187, New York, NY, USA, 2003. ACM.

[75] Rolf Ernst, Jorg Henkel, and Thomas Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test*, 10(4):64–75, 1993.

[76] Kayvon Fatahalian et al. Sequoia: programming the memory hierarchy. In *SC '06*, page 83, 2006.

[77] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990.

[78] Joseph A. Fisher. Very Long Instruction Word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150, New York, NY, USA, 1983. ACM.

[79] William Fornaciari and Vincenzo Piuri. Virtual FPGAs: Some steps behind the physical barriers. *Parallel and Distributed Processing*, 1388:7–12, 1998.

[80] Burkhard Freitag, Heribert Schütz, and Günther Specht. Lola - a logic language for deductive databases and its implementation. In *Proceedings of the Second International Symposium on Database Systems for Advanced Applications*, pages 216–225. World Scientific Press, 1992.

[81] Virginie Fresse, Olivier Déforges, and Jean-François Nezan. AVSynDEx: a rapid prototyping process dedicated to the implementation of digital image processing applications on multi-DSP and FPGA architectures. *EURASIP J. Appl. Signal Process.*, 2002(1):990–1002, 2002.

[82] Roy Friedman, Maxim Goldin, Ayal Itzkovitz, and Assaf Schuster. MILLIPEDE: Easy parallel programming in available distributed environments. In *Software: Practice and Experience*, volume 27, pages 929–965. John Wiley & Sons, Ltd., 1997.

[83] Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[84] D. D. Gajski, J. Zhu, R. Domer, A Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[85] Laurent Gantel, Salah Layouni, Mohamed El Amine Benkhelifa, F. Verdier, and Stphanie Chauvet. Multiprocessor task migration implementation in a reconfigurable platform. In *International conference on ReConFigurables Computing and FPGAs (ReConFig)*. IEEE Computer Society, 2009.

[86] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7, 1997.

[87] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[88] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[89] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *FCCM '00*, 2000.

[90] Robert Goldberg. Survey of virtual machine research. *Computer*, 7 - 6:34–45, 1974.

[91] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[92] Ian Gray and Neil Audsley. Application-defined virtualisation for embedded real-time software on complex architectures. *2nd Junior Researcher Workshop on Real-Time Computing*, pages 1–4, 2008.

[93] Ian Gray and Neil Audsley. Exposing non-standard architectures to embedded software using Compile-Time Virtualisation. *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '09)*, 2009.

[94] Ian Gray and Neil Audsley. Supporting islands of coherency for highly-parallel embedded architectures using Compile-Time Virtualisation. In *13th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2010.

[95] Eike Grimpe and Frank Oppenheimer. Extending the SystemC synthesis subset by object-oriented features. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 25–30, New York, NY, USA, 2003. ACM Press.

[96] W. W. Gropp and E. L. Lusk. A taxonomy of programming models for symmetric multiprocessors and SMP clusters. In *PMMP '95: Proceedings of the conference on Programming Models for Massively Parallel Computers*, page 2, Washington, DC, USA, 1995. IEEE Computer Society.

[97] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994.

[98] J.E. Gunn, K.S. Barron, and W. Ruczczyk. A low-power DSP core-based software radio architecture. *IEEE, Selected Areas in Communications*, 17(4):574–590, Apr 1999.

[99] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, 1993.

[100] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466, Jan. 2003.

[101] Reiner W. Hartenstein and Rainer Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *ASP-DAC '95: Proceedings of the 1995 Asia and South Pacific Design Automation Conference*, page 77, New York, NY, USA, 1995. ACM.

[102] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. *5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, 00:87, 1997.

[103] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. *5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, 00:12, 1997.

[104] Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In *Lecture Notes in Computer Science*, volume Volume 4085/2006, pages 1–15. Springer Berlin / Heidelberg, 2006.

[105] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[106] Paul N. Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium language reference manual. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2001.

[107] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[108] Chao Huang, Orion Lawlor, and L. V. Kal. Adaptive MPI. In *In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03*, pages 306–322, 2003.

[109] He Huang, Nan Yuan, Wei Lin, Guoping Long, Fenglong Song, Lei Yu, Yuping Liu, Lei Liu, Yongbin Zhou, Xiaochun Ye, Junchao Zhang, Dongrui Fan, and Zhimin Tang. Architecture supported synchronization-based cache coherence protocol for many-core processors. *Chinese Journal of Computers*, 8:1618–1630, 2009.

[110] Michael Huebner, Tobias Becker, and Juergen Becker. Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 28–32, New York, NY, USA, 2004. ACM Press.

[111] Ashley Hull. Nintendo DS memory layout. http://www.dev-scene.com/NDS/Tutorials_Day_2, 2008. Accessed 27/03/2008.

[112] IBM Corporation. CoreConnect bus architecture. https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture, May 2010.

[113] Impulse Accelerated Technologies. http://www.impulsec.com, Accessed Jan 2008.

[114] Institute of Electrical and Electronics Engineers. POSIX.1b, real-time extensions (IEEE Std 1003.1b-1993), 1995.

[115] Institute of Electrical and Electronics Engineers. POSIX.1c, threads extensions (IEEE Std 1003.1c-1995), 1995.

[116] Institute of Electrical and Electronics Engineers. SystemC language reference manual (IEEE std 1666-2005), 2005.

[117] International Telecommunication Union. X.200 : Information technology - open systems interconnection - basic reference model: The basic model. ITU-T Recommendation X.200, July 1994.

[118] ITRS. International technology roadmap for semiconductors, 2007 edition. http://www.itrs.net/, 2007.

[119] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread migration and its applications in distributed shared memory systems. *Journal of Systems and Software*, 42(1):71 – 87, 1998.

[120] Geraint Jones. *Programming in Occam*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1986.

[121] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[122] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[123] G. Kahn. The semantics of a simple language for parallel programming. In *Procceedings of the IFIP Congress 74*, 1974.

[124] A. Kalavade and E. A. Lee. The extended partitioning problem: hardware/software mapping and implementation-bin selection. *Design Automation for Embedded Systems*, 2:12, 1995.

[125] Asawaree Kalavade and Edward A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 42–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[126] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, New York, NY, USA, 1993. ACM.

[127] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:244–254, 1986.

[128] Torsten Kempf, Malte Doerper, R. Leupers, G. Ascheid, H. Meyr, Tim Kogel, and Bart Vanthournout. A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 876–881, Washington, DC, USA, 2005. IEEE Computer Society.

[129] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[130] S. Khawam, I. Nousias, M. Milward, Ying Yi, M. Muir, and T. Arslan. The Reconfigurable Instruction Cell Array. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16 - 1:75–85, 2008.

[131] G. Kiczales. Aspect-Oriented Programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[132] Gregor Kiczales, J. Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. *Metaobject Protocols: Why We Want Them and What Else They Can Do*, pages 101–118. The MIT Press, Cambridge, MA, 1993.

[133] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[134] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, Jonghee Yoon, and Yunheung Paek. Memory-aware application mapping on Coarse-Grained Reconfigurable Arrays. *Lecture Notes in Computer Science*, 5952/2010:171–185, 2010.

[135] Ray Klefstad, Mayur Deshpande, Carlos ORyan, Angelo Corsaro, Arvind S Krishna, Sumita Rao, and Krishna Raman. The performance of ZEN: A real time CORBA ORB using real time java. In *Proceedings of Real-time and Embedded Distributed Object Computing Workshop*. OMG, September 2002.

[136] Ravi Konuru, Jeremy Casas, Steve Otto, Robert Prouty, and Jonathan Walpole. A user-level process package for PVM. Technical report, Oregon Graduate Institute School of Science & Engineering, 1994.

[137] Ravi B. Konuru, Steve W. Otto, and Jonathan Walpole. A migratable user-level process package for PVM. *J. Parallel Distrib. Comput.*, 40(1):81–102, 1997.

[138] Andreas Krall and Reinhard Grafl. CACAO - a 64 bit JavaVM Just-in-Time compiler. In *In Concurrency: Practice and Experience*, pages 1017–1030. ACM, 1997.

[139] Shashi Kumar, Axel Jantsch, Mikael Millberg, Johny Oberg, Juha-Pekka Soininen, Martti Forsell, Kari Tiensyrja, and Ahmed Hemani. A network on chip architecture and design methodology. *ISVLSI*, 00:0117, 2002.

[140] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A high integrity profile for Real-Time Java. In *In Joint ACM Java Grande/ISCOPE Conference*, pages 131–140. ACM Press, 2002.

[141] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

[142] Yann-Hang Lee and C.M. Krishna. Voltage-clock scaling for low energy consumption in real-time embedded systems. *Real-Time Computing Systems and Applications, International Workshop on*, 0:272, 1999.

[143] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, 1992.

[144] Wen-Yew Liang, Chun-Ta King, and Feipei Lai. Adsmith: an efficient object-based distributed shared memory system on PVM. *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings. Second International Symposium on*, pages 173–179, Jun 1996.

[145] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, 1995.

[146] Calvin Lin and Lawrence Snyder. Zpl: An array sublanguage. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 96–114, London, UK, 1994. Springer-Verlag.

[147] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[148] X. P. Ling and H. Amano. WASMII: a data driven computer on a virtual hardware. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[149] Mirko Loghi, Massimo Poncino, and Luca Benini. Cache coherence tradeoffs in shared-memory MPSoCs. *ACM Trans. Embed. Comput. Syst.*, 5(2):383–407, 2006.

[150] Grant Edmund Martin Lou Scheffer, Luciano Lavagno. *EDA for IC system design, verification, and testing*. CRC Press, Taylor and Francis Group, 2006.

[151] D.B. Loveman. High performance fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, Feb 1993.

[152] Tom Lovett and Russell Clapp. STiNG: a CC-NUMA computer system for the commercial marketplace. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 308–317, New York, NY, USA, 1996. ACM.

[153] D.C. Luckham and J. Vera. An event-based architecture definition language. *Software Engineering, IEEE Transactions on*, 21(9):717–734, Sep 1995.

[154] Roman Lysecky, Greg Stitt, and Frank Vahid. Warp processors. In *ACM Transactions on Design Automation of Electronic Systems*, volume 11, pages 659–681, New York, NY, USA, 2006. ACM.

[155] Rafey Mahmud. An FPGA primer for ASIC designers. EETimes (http://www.eetimes.com/news/design/features/showArticle.jhtml?articleID=18901725, Accessed Dec 2009), 15th April 2004.

[156] F. Makedon and A. Symvonis. An efficient heuristic for permutation packet routing on meshes with low buffer requirements. *IEEE Transactions on Parallel and Distributed Systems*, 4:270–276, 1993.

[157] J. Malenfant, M. Jacques, and F. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of Reflection 96*, 1996.

[158] T. Marescaux, J. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, and R. Lauwereins. Networks on chip as hardware components of an OS for reconfigurable systems. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications, Lisbon*, 2003.

[159] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-Threading technology architecture and microarchitecture. In *Intel Technology Journal*, volume 6, pages 1–12, Feb 2002.

[160] P. Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[161] Edward Mascarenhas and Vernon Rego. Ariadne: architecture of a portable threads system supporting thread migration. In *Software - Practice and Experience*, volume 26, pages 327–356, 1996.

[162] P. Mattson, U. Kapasi, J. Owens, and S. Rixner. Imagine programming system user's guide. http://cva.stanford.edu/classes/ee482s/docs/ips_user.pdf, 2001. Accessed 23/07/2010.

[163] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. *Lecture Notes in Computer Science*, 2778/2003:61–70, 2003.

[164] Mentor Graphics. Modelsim. http://www.model.com/. Accessed Dec 2009.

[165] Mentor Graphics. Catapult-c synthesis. http://www.mentor.com/catapult, 2009.

[166] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. *SIGOPS Oper. Syst. Rev.*, 40(4):403–414, 2006.

[167] Takashi Miyamori and Kunle Olukotun. REMARC: Reconfigurable multimedia array coprocessor. *IEICE Transactions on Information and Systems*, pages 389–397, 1998.

[168] Anca Molnos, Aleksandar Milutinovic, Dongrui She, and Kees Goossens. Composable processor virtualization for embedded systems. In *In Proc. Workshop on Computer Architecture and Operating System Co-Design (CAOS)*, January 2010.

[169] Aaftab Munshi, editor. *The OpenCL Specification*. Khronos OpenCL Working Group, 2008.

[170] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26:62–76, 1993.

[171] Ralf Niemann and Peter Marwedel. Hardware/software partitioning using integer programming. In *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, page 473, Washington, DC, USA, 1996. IEEE Computer Society.

[172] Ralf Niemann and Peter Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2:165–193, 1997.

[173] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2009. ACM.

[174] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52 –60, Aug 1991.

[175] NVIDIA Corporation. CUDA programming guide ver 1.1. http://developer.nvidia.com, 2007. Accessed 23/07/2010.

[176] Flavio Oquendo. π-adl: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM Software Engineering Notes*, 29-3, 2004.

[177] S. Pasricha. Transaction level modelling of SoC with SystemC 2.0. In *Synopsys User Group Conference*, 2002.

[178] JoAnn M. Paul, Alex Bobrek, Jeffrey E. Nelson, Joshua J. Pieper, and Donald E. Thomas. Schedulers as model-based design elements in programmable heterogeneous multiprocessors. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 408–411, New York, NY, USA, 2003. ACM.

[179] Pierre Paulin, Chuck Pilkington, and Essaid Bensoudane. StepNP: A system-level exploration platform for network processors. *IEEE Des. Test*, 19(6):17–26, 2002.

[180] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation Cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):179 – 196, jan. 2006.

[181] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring embedded-systems architectures with Artemis. *Computer*, 34(11):57–63, 2001.

[182] Monica Pinto, Lidia Fuentes, and Jose Maria Troya. DAOP-ADL: An architecture description. In *In GPCE 03: Proceedings of the second international conference on Generative programming and component engineering*, pages 118–137. Springer Verlag, 2003.

[183] Alan LaMont Pope. *The CORBA reference guide: understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[184] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[185] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogeneous archtitectures. In *FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, page 58, Washington, DC, USA, 1997. IEEE Computer Society.

[186] Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi. Standard templates adaptive parallel library (STAPL). In *In Proc. of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR*, pages 402–409. Springer-Verlag, 1998.

[187] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[188] Mehrdad Reshadi, Bita Gorjiara, and Daniel Gajski. NISC technology and preliminary results - technical report CECS-05-11. Technical report, Center for Embedded Computer Systems, University of California, Irvine, August 24, 2005.

[189] Fred Rivard. Smalltalk: A reflective language. In *International Conference on Metalevel Architectures and Reflection*, 1996.

[190] J. Robelly, G. Cichon, H. Seidel, and G. Fettweis. A HW/SW design methodology for embedded SIMD vector signal processors, 2005.

[191] A. Rose, S. Swan, J. Pierce, and Jean-Michel Fernandez. Transaction Level Modeling in SystemC. In *Open SystemC Initiative*, 2005.

[192] Sumit Roy and Vipin Chaudhary. Strings: A high-performance distributed shared memory for symmetrical multiprocessor clusters. *High-Performance Distributed Computing, International Symposium on*, 0:90, 1998.

[193] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in Java. In David Kotz and Friedemann Mattern, editors, *Agent Systems, Mobile Agents, and Applications*, volume 1882 of *Lecture Notes in Computer Science*, pages 443–481. Springer Berlin / Heidelberg, 2000.

[194] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The Asynchronous Partitioned Global Address Space model. http://www.cs.rochester.edu/u/cding/amp/papers/full/The%20Asynchronous%20Partitioned%20Global%20Address%20Space%20Model.pdf, March 2010.

[195] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8:10–17, 2001.

[196] Martin Schoeberl. JOP: A Java optimized processor for embedded real-time systems. Master's thesis, Technischen Universitat Wien, 2005.

[197] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in Virtex FPGAs. In *IEE Proceedings of Computers and Digital Techniques*, volume 153-3, pages 157–164, 2006.

[198] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering, IEEE Transactions on*, 21(4):314–335, Apr 1995.

[199] Fridtjof Siebert. JEOPARD – Java environment for parallel real-time development. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:28–36, 2009.

[200] Elias Teodoro Silva Jr., David Andrews, Carlos Eduardo Pereira, and Flávio Rech Wagner. An infrastructure for hardware-software co-design of embedded Real-Time Java applications. *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 273–280, 2008.

[201] Hartej Singh, Ming hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh, and Eliseu M. C. Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49:465–481, 2000.

[202] Top500 Supercomputer Sites. The 34th Top500 list. http://www.top500.org/files/newsletter112009_tabloid_v3.pdf, November 2009. Accessed November 2009.

[203] Brian Smith. Proceedural reflection in programming lanugages. Technical report, Massachusetts Institute of Technology, 1982.

[204] Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison-Wesley Professional, 2008.

[205] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The Tui system. Technical report, Vancouver, BC, Canada, Canada, 1996.

[206] J. Sobel and D. Friedman. An introduction to reflection-oriented programming. In *Reflection '96*, 1996.

[207] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[208] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[209] Leon Stok. Data path synthesis. *Integration, the VLSI Journal*, 18(1):1 – 71, 1994.

[210] Dag Stranneby. *Digital Signal Processing, DSP & Applications*. Newnes, Oxford, 2001.

[211] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, May 1990.

[212] Sun Microsystems. Fortran 77 4.0 reference manual. http://www.physics.ucdavis.edu/ vem/F77_Ref.pdf, November 1995. Accessed Dec 2009.

[213] Inc. Sun Microsystems. Virtualbox.org. http://www.virtualbox.org/ - Accessed Dec 2009, 2009.

[214] Stuart Sutherland, Simon Davidmann, Peter Flake, and P. Moorby. *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[215] Herb Sutter. A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30, no. 3:16–20, 2005.

[216] Antonios Symvonis and Jonathon Tidswell. An empirical study of off-line permutation packet routing on 2-dimensional meshes based on the multistage routing method. Technical report, IEEE Transactions on Computers, 1994.

[217] David Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, 2000.

[218] Texas Instruments, Inc. *The TTL Data Book for Design Engineers*, volume ISBN: 0895121115. Dallas Texas Instruments, 1984.

[219] Texas Instruments Inc. OMAP3530/25 applications processor (Rev. F). http://www.ti.com/lit/gpn/omap3530, Oct 2009.

[220] The Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems*. MISRA Ltd., 2004.

[221] W. Thies et al. StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[222] K. Thitikamol and P. Keleher. Thread migration and communication minimization in dsm systems. *Proceedings of the IEEE*, 87(3):487 –497, mar 1999.

[223] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 9–14, New York, NY, USA, 2007. ACM.

[224] Tuomo Tikkanen, Timo Lappänen, and Jorma Kivelä. Structured analysis and VHDL in embedded ASIC design and verification. In *EURO-DAC '90: Proceedings of the conference on European design automation*, pages 107–111, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[225] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in Java. In David Kotz and Friedemann Mattern, editors, *Agent Systems, Mobile Agents, and Applications*, volume 1882 of *Lecture Notes in Computer Science*, pages 377–426. Springer Berlin / Heidelberg, 2000.

[226] M. Ullmann, M. Huebner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 135–, April 2004.

[227] Frank Vahid. Modifying min-cut for hardware and software functional partitioning. In *CODES '97: Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, page 43, Washington, DC, USA, 1997. IEEE Computer Society.

[228] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[229] John D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, New York, NY, USA, 1995. ACM.

[230] K. Van Rompaey, D. Verkest, I. Bolsens, and H. De Man. CoWare - a design environment for heterogeneous hardware/software systems. *Proceedings of the conference on European design automation*, pages 252–257, Sep 1996.

[231] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, New York, NY, USA, 2003. ACM.

[232] VMware, Inc. VMware ESX and VMware ESXi - product datasheet. item no: VMW_09Q3_DS_ESX_ESXi_USLET_EN_P6_R4. http://www.vmware.com/products/esx/, 2009.

[233] VMware, Inc. VMware Server 2 - product datasheet. item no: VMW_09Q3_DS_SERVER_USLET_EN_P2_R3. http://www.vmware.com/products/server/, 2009.

[234] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[235] Michael Ward. *Improving the Timing Analysis of Ravenscar / SPARK Ada by Direct Compilation to Hardware*. PhD thesis, York University Computer Science Dept., 2005.

[236] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1999.

[237] AJ Wellings and A Burns. Beyond Ada 2005: Allocating tasks to processors in SMP systems. *Ada User Journal*, 29-2:127–132, 2008.

[238] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications - technical report 02-02-01. Technical report, Univ. of Washington, 2002.

[239] T. Wiangtong, P.Y.K. Cheung, and W. Luk. Hardware/software codesign: a systematic approach targeting data-intensive applications. *Signal Processing Magazine, IEEE*, 22(3):14–22, May 2005.

[240] Daniel Wiklund and Dake Liu. SoCBUS: Switched network on chip for hard real time embedded systems. In *IPDPS '03*, page 78.1, 2003.

[241] Maurice V. Wilkes. The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News*, 29(1):2–7, 2001.

[242] Anthony Williams. The Boost.Thread library. http://www.boost.org, 2008. Accessed August 2010.

[243] J. Williams, N. Heintze, and B. Ackland. Communication mechanisms for parallel DSP systems on chip. *Design, Automation and Test in Europe (DATE'02)*, 2002.

[244] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[245] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. *3rd IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '95)*, 00:0099, 1995.

[246] Wayne Wolf. A decade of hardware/software codesign. *Computer*, 36:38–43, 2003.

[247] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh, and R. Johnston. Real-time CORBA. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, page 148, Washington, DC, USA, 1997. IEEE Computer Society.

[248] David H. Wolpert and William G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, IEEE Trans on Evolutionary Computation, Santa Fe, NM, 1995.

[249] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[250] Weng-Fai Wong, R.S. Nikhil, D.L. Rosenband, and N. Dave. High-level synthesis: an essential ingredient for designing complex ASICs. *Computer-Aided Design, International Conference on*, 0:775–782, 2004.

[251] Xilinx. *The Programmable Logic Data Book*. Xilinx Inc., 1999.

[252] Xilinx Corporation. RocketIO transceiver user guide. *Xilinx Application Notes*, UG024, 2003.

[253] Xilinx Corporation. DS007: Spartan-IIe FPGA family: Complete data sheet. www.xilinx.com/support/documentation/data_sheets/ds007.pdf, July 2004.

[254] Xilinx Corporation. Online ABEL reference. http://toolbox.xilinx.com/docsan/xilinx7/help/iseguide/mergedProjects/abelref/abelref.htm, 2005.

[255] Xilinx Corporation. Virtex-5 FPGA configuration user guide. *Xilinx User Guides*, UG191, 2006.

[256] Xilinx Corporation. Xilkernel. http://www.xilinx.com/ise/embedded/edk91i_docs/xilkernel_v3_00_a.pdf, December 2006.

[257] Xilinx Corporation. Virtex-4 user guide. *Xilinx User Guides*, UG070, 2007.

[258] Xilinx Corporation. Microblaze processor reference guide. UG081 v9.0, 2008.

[259] Xilinx Corporation. DS265: LogiCORE IP CAN v3.2: Data sheet. www.xilinx.com/support/documentation/ip_documentation/can_ds265.pdf, August 2009.

[260] Xilinx Corporation. DS312: Spartan-3e FPGA family: Data sheet. www.xilinx.com/support/documentation/data_sheets/ds312.pdf, August 2009.

[261] Xilinx Corporation. DS579: XPS central DMA controller v2.01.b: Data sheet. http://www.xilinx.com/support/documentation/ip_documentation/xps_central_dma.pdf, September 2009.

[262] Xilinx Corporation. DS632: XPS mailbox v2.00.b: Data sheet. http://www.xilinx.com/support/documentation/ip_documentation/xps_mailbox.pdf, December 2009.

[263] Xilinx Corporation. Embedded system tools reference guide - EDK 11.3.1. *Xilinx Application Notes*, UG111, 2009.

[264] Xilinx Corporation. UG642: Platform specification format reference manual. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/psf_rm.pdf, September 2009.

[265] Xilinx Corporation. DS643: Multi-port memory controller (MPMC) v6.01.a: Data sheet. http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf, July 2010.

[266] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:128, 1999.

[267] Andrew Chi-Chih Yao. New algorithms for bin packing. *J. ACM*, 27(2):207–227, 1980.

[268] Cesar Albenes Zeferino and Altamiro Amadeu Susin. SoCIN: A parametric and scalable network-on-chip. In *SBCCI '03*, page 169, Washington, DC, USA, 2003. IEEE Computer Society.

[269] Wenzhang Zhu, Cho-Li Wang, and F.C.M. Lau. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. pages 381 – 388, 2002.