Operating System Kernels on Multi-core Architectures

Hesham Moustafa Khaled Almatary

MSc by Research

University of York

Computer Science

January 2016

Abstract

Operating System (OS) kernels have been under research and development for decades, mainly assuming single processor and distributed hardware systems. With the recent rise of multi-core chips that may incorporate a network on chip (NoC), new challenges have appeared that were not considered before. Given that a complete multi-core system that works on a single system on chip (SoC) is now the normal case, different cores on a single SoC may share other physical resources and data. This new sharing scheme on a SoC affects crucial aspects of an overall system like correctness, performance, predictability, scalability and security. Both hardware and OSs to flexibly cooperate in order to provide solutions for such challenges.

SoC mimics the internet somehow now, with different cores acting as computer nodes, and the network medium is given in an advanced digital fabrics like buses or NoCs, that are a current research area. However, OSs are still assuming some (hardware) features like single physical memory and memory sharing for inter-process communication, page-based protection, cache operations, even when evolving from uniprocessor to multi-core processors. Such features not only may degrade performance and other system aspects, but also some of them make no sense for a multi-core SoC, and introduce some barriers and limitations. While new OS research is considering different kernel designs to cope up with multi-core systems, they are still limited by the current commercial hardware architectures.

The objective of this thesis is to assess different kernel designs and implementations on multi-core hardware architectures. Part of the contributions of the thesis is porting RTEMS (RTOS) and seL4 microkernel to Epiphany and RISC-V hardware architectures respectively, trading-off the design and implementation decisions. This hands-on experience gave a better understanding of the real-world challenges regarding kernel designs and implementations.

Contents

List of Figures

Acknowledgements

I would like to thank my father, who has always been my hero, and supported all my decisions, my mother for spreading joy and my sister for being a great friend. I would not have been what I am if it was not for my family. Special thanks to my supervisor Neil Audsley for helping me out during this great research journey, he has been of great support during my time at the university. Thanks to Gedare Bloom and Joel Sherrill of RTEMS, who not only provided me with resources and advices about RTEMS, but also gave me personal opinions, ideas and suggestions. Thanks for trusting me.

Author's Declaration

I declare that, except where explicit reference is made to the contribution of others, that this thesis is the result of my own work and has not been submitted for any other degree at the University of York or any other institution.

Scientific and open-source development contributions that I have made during the time of doing this degree and related to this thesis are:

- **RTEMS port to Epiphany** Patches for RTEMS to support the Epiphany architecture and Parallella BSP are merged with RTEMS upstream repository. I am the only and first maintainer for this port within RTEMS.
- **RTEMS ports to RISC-V and MicroBlaze** These ports are held on my own public github repositories (not merged with RTEMS upstream yet). There are future plans to merge these ports with RTEMS upstream.
- **seL4 port to RISC-V** This is a project that I worked on part of Google Summer of Code 2015 program with lowRISC/RISC-V organizations. I gave a presentation about this project at ORCONF 2015 conference at CERN, Switzerland.
- **Reducing the Implementation Overheads of IPCP and DFP** I co-authored this paper with Alan Burns and Neil Audsley. The paper was accepted and presented by Alan Burns at RTSS 2015 conference, and to be published.
- **Integration of VScale/RISC-V core with OpenRISC** I worked on changing the interface of the UC Berkeley's VScale core

implementation of RISC-V, and wrapped it up with a wishbone interface to work with other open cores. This work was merged with OpenRISC/FuseSoC.

Chapter 1

Introduction

Computer architecture has moved into a new era of the multi-core system on chips to overcome the bottleneck of overheating and power consumption within a high frequency uniprocessor system while preserving Moore's law, so that a single chip can accommodate smaller, and more, numbers of transistors. This is achieved by dividing up computation cores from a single high performance uniprocessor, into smaller and slower cores. Such cores can be identical (homogeneous) or different (heterogeneous). Hardware designers have striven to make this architecture change as transparent as possible to the programmer with the cost of making the hardware more complex, and sometimes lower performance. This is not an acceptable solution any more.

As Hennessy and Patterson state it clearly in their state-of-art computer architecture book: "The La-Z-Boy programmer era of relying on hardware designers to make their programs go faster without lifting a finger is officially over." [56]

The software has to be re-designed to run in parallel on such multi-core systems. This will make life (relatively) easier for hardware designers, while increasing the software performance. Re-designing a software that assumed uniprocessor systems for decades especially if this software has thousands and millions lines of code is not an easy task. Programmers now not only have to worry about their code correctness, but also they need to take parallelism, performance, scalability and security into consideration, each with its own challenges. Moreover, hardware and software designers have to work closely to introduce better solutions instead of working on their own problems separately.

Typically, an OS kernel is the first piece of software that deals with hardware directly, and provides common services to higher level applica-

tions. This means that OS kernel developers are the first to be exposed to the new multi-core hardware architecture. On the one hand, contemporary monolithic OSs like Linux have been trying to modify their design to work on multi-core systems in a Symmetric Multiprocessing (SMP) manner, and developers have to deal with problems and bugs as they arise. Furthermore, such adoption is not always scalable beyond some number of cores due to the inherent hardware and software sharing scheme such monolithic kernels have been designed for. On the other hand, the research community is working on introducing new OS kernel designs that are written from scratch [23, 41, 103] (rather than building on an existing kernel) with multi-core architecture in mind. The later has helped with introducing new better solutions that can help with scalability, performance, and security.

As the internet has proven that it is scalable and reliable, the new hardware and OS architectures are embracing many of the old distributed systems ideas. In fact, network on chips have implemented some of the same internet-based protocols to work on a single chip. The OS developers are also trying to adopt old distributed OS ideas that were working over the internet, to work on a single chip. Since microkernel distributed system design has been of a big success, current research OSs are inspired by microkernel and message passing designs. Such research OS kernels are like pure microkernel, multikernel and factored OSs.

Unfortunately such promising research OSs are still being limited by the current commercial hardware architecture constraints such as (non scalable) caches, page-based granularity for memory protections, memory sharing for inter-process communication, etc. Some of them are only prototypes that run on simulators only.

The goal of this thesis is to address the issues that different OS designs face when written (or ported) to run on multi-core hardware architectures. In fact, hardware architecture greatly affects the design and implementation of OS kernel designs and implementations. Some kernel designs work well on multi-core architectures like microkernels and multikernels, basically because of their message passing model, scalability and modularity, but others do not. Furthermore, the hardware architecture might prevent an OS from implementing one or more of its design principles. Basically, an OS manages the three basic components of a hardware computing system: CPU(s), memories and IO, and it exports

an interface to applications in the form of services. That is what an OS in general is for. OS designs differ in which service goes into which layer (kernel/privilege layer or user/application layer, and sometimes hypervisor layer) and how different OS and application components communicate with each other (securely). Figure1.1 is an example of different kernel designs.

A basic set of services that a typical OS would provide are memory management, scheduling, Inter Process Communication (IPC) and device drivers. Different kernel designs tend to end up into two old rivals: monolithic and microkernel. Other kernel designs like hybrid, multikernel and factored OS can be considered as derivatives of the two major ones.

On the one hand, monolithic kernels get the advantage of performance and a rich set of services and device drivers implemented in a privileged mode. Its high performance is due to the fact that the communication between applications, kernel and services are just direct function calls and/or system calls. On the other hand, microkernels get the advantage of modularity between its subsystems that enable it to be split up into separate physical hardware cores, making it highly scalable, unlike monolithic kernels. However, such scalability and modularity features come with a performance overhead incurred within the message-passing IPC component as every microkernel subsystem (including the microkernel, applications and device drivers) interacts with one another using this IPC layer.

With the beginning of the multi-core era both designs are trying to make use of the new cores and parallelism, and both face issues. To give an example, some hardware architectures with low memory resources might require the kernel services to be split up into different cores with distributed memory. This is possible with microkernels, but might be impossible with monolithic kernels. Similarly, some hardware features like cache coherency and shared memory work well with monolithic kernel, but not with microkernels. Networks on chip and new hardware security implementations can be easily/better utilised by microkernel's components like message-passing and capability management to boost performance and security, but might make no sense for a monolithic kernel (if it does not break its functionality and/or design principles).

In order to get an actual real-world experience of these issues, this

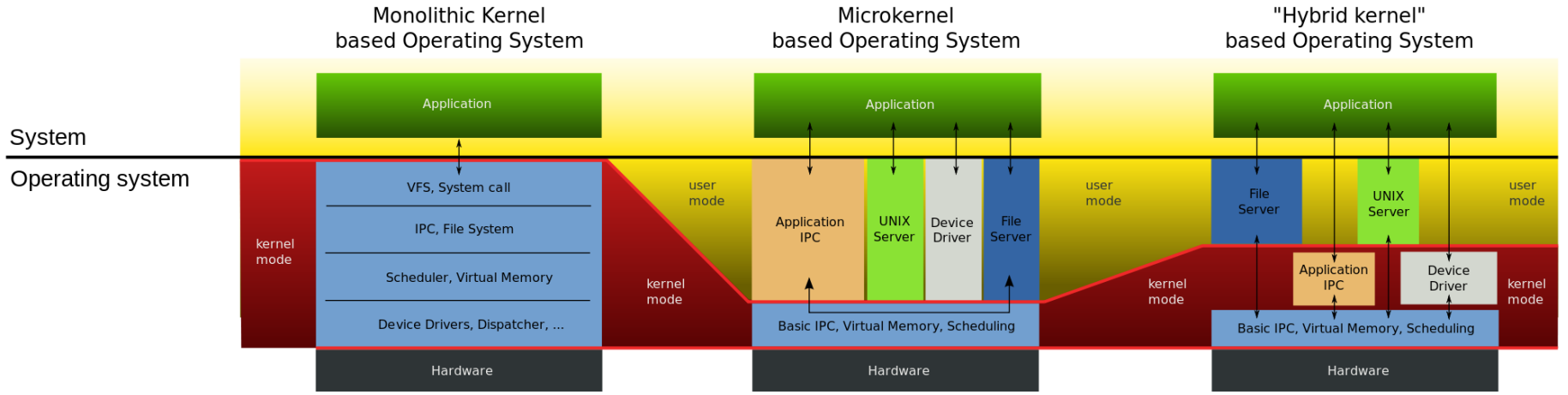


Figure 1.1: Operating System Structures [108]

thesis reports the effort of porting different kernel designs like RTEMS (as a monolithic RTOS), and seL4 microkernel to different multi-core systems like Epiphany/Parallella and RISC-V architectures. RTEMS and seL4 are cutting-edge OSs that have been deployed in important real-world safety critical projects (involving NASA and military projects), and both can be regarded as a state-of-art of its kind. Ports of RTEMS to both OpenRISC and Epiphany are already upstream, and currently being used and developed part of other projects. Furthermore, RTEMS is on its early stages of supporting SMP, so porting it to Epiphany (with 16 cores), enabled us to investigate pros and cons of RTEMS SMP support and Epiphany (as a new multi-core chip), along with proposing ideas to improve both. It is noticed that trying to convert a monolithic kernel like RTEMS to other designs was not practical on a multi-core architecture like Epiphany. The thesis discusses the reasons why that was not possible, giving a motivation to investigate another kernel design like seL4 on another hardware architecture. seL4 microkernel is more sophisticated than RTEMS (from design perspective), and requires more hardware features and resources. So, porting it to RISC-V architecture gives a better understanding of the hardware requirements needed for microkernels not only to run, but also to scale and maintain its simplicity and security features.

Chapter 2 provides a literature review of hardware architecures along with OS designs and examples. The chapter also dicusses how those OS implementations fit with hardware architecures, and the issues involved. Chapter 3 gives a hands-on experience (author's contribution) of the challenges faced while porting RTEMS to Epiphany multi-core NoC. The chapter concludes hardware limitations of Epiphany, and shows the motivation to use a more flexible hardware like RISC-V. Chapter 4 gives an overview of RISC-V hardware architecure and seL4 microkernel, and summarizes the porting experience of seL4 to RISC-V. Chapter 5 discusses the issues of OSs on multi-core architecures, and proposes solutions. Finally comes the conclusion in chapter 6.

Chapter 2

Field Survey and Review

The following sections cover the literature of the components of the stack that are involved with the issues discussed in the introduction. These components are OSs and multi-core hardware architectures. The review will only deal with multi-core hardware and OSs research topics giving some examples of them and how they propose solutions for scalability, performance and security.

2.1 Architecture of Multiprocessor Systems

In this section the research about multi-core multiprocessor System on Chip (SoC) is discussed, traditional single-core systems and clustered internet-based distributed systems are beyond this survey as the focus is on the single chip/board level. Designing and implementing Multi-Processor System on Chip (MPSoC) have trade-offs and challenges. Most notably the trade-off between hardware and software support for such a system. Making the programmer's life easier comes with the cost of a complex hardware support which consumes power, size, and sometimes introduces programming limitations, performance degradation and suffers from scalability issues (like cache-coherency). On the other hand, it is possible to reduce the hardware cost and complexity by exposing the MPSoC architecture to the programmer, who in turn can make the best use of this parallel hardware, even better than the hardware support, or might totally disrupt the hardware he/she is working on by writing buggy and error-prone programs due to ignorance of how to deal with such a new hardware design and software model (message passing model for example). In the following section some of the hardware architectures are provided and the focus will be on memory organisation of such systems

as it is the crucial component that affects correctness and performance and OS kernel designs and implementations. In the section that follows it, OS solutions are discussed.

2.1.1 Shared Memory Multiprocessor Architectures

Uniprocessors were easy to program, the programmer was only required to write a correct sequential software while performance and instruction-level parallelism techniques were left to the hardware designer. Since 2003, uniprocessors could not keep up with Moore's law by having more transistors on the same uniprocessor chip, mainly because of power consumption and heat bottlenecks that come with higher clock frequencies and an increase in the number of transistors. Although multiprocessor implementations were already there since the early days of computers, they were only part of research and mainframes fields (in data centres). So, to get over such an issue, multiprocessor designs have been implemented at the chip/board level that we know today as *multi-core* processors. Multiprocessor architectures vary but they fall into two super categories according to the memory organisation: Symmetric Shared Memory Multiprocessors (SMP), and Distributed Shared Memory Multiprocessors (DSM). Both architectures are discussed below.

Symmetric Shared Memory Multiprocessors

SMP means that for a system that has multiple processors sharing memory, each processor has the same memory access latency with one another. Thus, all processors share the same physical main memory with equivalent access latency as shown in figure 2.1.

The adoption of caches, originally to increase performance and reduce memory latency, for this architecture makes it more complicated than uniprocessors. If more than one processor have their own caches that may contain a copy of a shared data in main memory, there must be a way to enforce *coherency* of this shared data across different processor's caches when read and written. Different cache coherency protocols have been implemented to handle such a hurdle. Snoopy caches is a protocol to achieve coherency. In snoopy caches, each processor's cache that has a copy of a shared data monitors the memory bus for any updates by other processors. The main problem with snoopy caches is that with the increasing number of cores and caches, the monitoring process would add

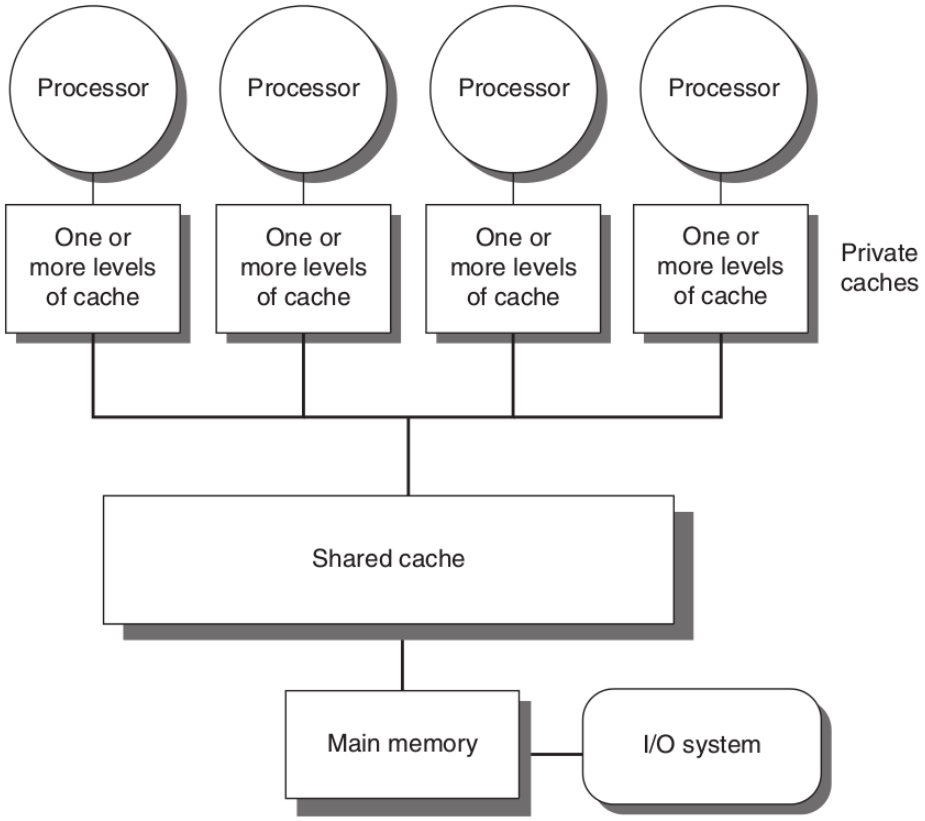


Figure 2.1: Basic structure of a centralized shared-memory multiprocessor based on a multi-core chip [56]

a memory latency overhead due to bus contention, and might saturate the memory bus. That is why snoopy caches and SMP do not scale well beyond small number of processors. New solutions have been proposed like multi-level caches, but still they do not scale well. So another architecture is needed if scalability is a concern, which is *Distributed Shared Memory*.

Distributed Shared Memory Multiprocessors

Distributed Shared Memory with caches, also known as ccNUMA (an abbreviation of Cache Coherent Non-Uniform Memory Access) is a way to build scalable multiprocessor systems that avoid the issues discussed in the previous SMP section. Attempts to build ccNUMA multiprocessor machines to address scalability challenges at the hardware level go back to the early 90s. DASH (Directory Architecture for Shared Memory) [67] is one of these early attempts to build a ccNUMA machine developed by Stanford University and is the first operational ccNUMA machine and is considered the base of modern ccNUMA architectures known today (SGI 2000 [65], SGI Altix [105], IBM Bluegene [79] and modern clustered SMP that will be discussed later). The motivation to build DASH was to figure out a way to build a scalable cache coherent machine that performs well with the increasing number of processors and competes with the scalable message-passing machines at this time. The solution was to divide up the physical shared memory into smaller physical ones, all of these smaller memories combined act as a single logical address space shared memory that the programmer assumes with cache coherency retained. The trick to construct this kind of physically scattered, logically shared address space was hidden by the hardware implementation.

Figure 2.2 shows a transition from a single shared memory architecture to ccNUMA.

A new hardware structure called the *directory* was introduced [1] to avoid the scalability limitations that come with the traditional snoopy caches. A directory is like a front-end gate of a small computing cluster consisting of one or more processors with a shared (snoopy) cache and physical memory; this directory connects the clusters to an interconnection network which connects other clusters together. Each directory

[1] Directories can be implemented in SMP machines where the outermost cache level (i.e. L3) implements a directory that keeps track of cache blocks held by private caches, and that is how Intel i7 cores work. However the directory concept discussed in this literature assumes DSM only.

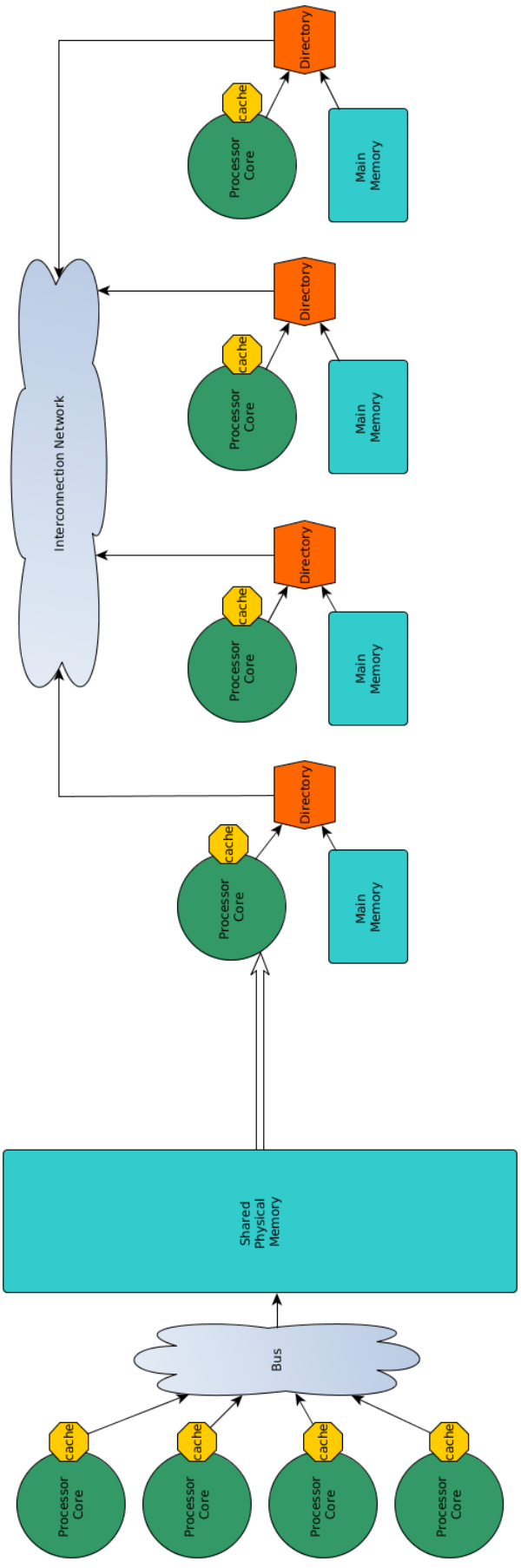


Figure 2.2: Distributed Shared Memory and Directory-based ccNUMA Architectures

keeps track of what data its memory has, and what data are cached in this cluster, and it has references to other clusters containing a copy of this data. This way the directory can make point-to-point invalidation operations to the caches that are involved only instead of the non-scalable broadcast method.

2.1.2 Memory Consistency

Memory consistency is a major concern when it comes to constructing multiprocessor shared-memory systems. The main question is, what is the order of memory operations from multiple processors as seen by the memory and/or the program? "The set of allowable memory access orderings forms the memory consistency model or event ordering model for an architecture." [50]. The consistency model is also defined to be a contract between the software and the memory [13]. There are different types of memory consistency models: strict, sequential [64], processor, weak, [13] and release [50] consistency models. Some of the consistency models are intuitive like strict consistency, but are inefficient from a performance point of view, while other models aim to relax the limitations incurred by each other to increase performance, but usually this makes it harder for the programmer.

2.1.3 Message Passing Networks-on-Chip-based Processors

Shared memory architectures are intuitively not scalable especially when there are many processors contending for a data structure on the same physical location of memory. Even with DSM architectures like directory-based ccNUMA that aims to enhance scalability, it is believed that even ccNUMA will not be able to cope up with increasing number of cores beyond 1000+ cores [103, 23] due to the complexity and communication overhead needed to preserve coherency between caches.

As the internet-based distributed system proved it can be scalable for millions of devices, current SoC trend tends to adopt the message passing nature of the internet on the chip level, using NoCs. This allows a processor to scale well on the chip/board level. Such NoC based many-core systems already exist and even manufactured by big semi-conductor companies like Intel [59], while it is currently an

important research area within universities [80, 5, 3].

Modern multi-core architectures tend to maintain a mixture of SMP and DSM and some integrate NoCs to get the best features of each. Tilera [39] is a recent non-uniform scalable multi-processor chip with caches and cores connected to each other by iMesh NoC and it also uses message passing. Coherency on Tilera is maintained by directories. AMD Opteron has 48 cores, cache coherency is implemented by MOESI protocol [106] while Intel Xeon has 80 cores divided up into 8 chips, each of which has two 10 cores. Each chip maintains cache coherency using snooping protocol, and broadcasting is used to achieve coherency across different chips. Sun Niagara 2 has 8 cores and each core is able to handle 8 hardware threads, thus, it can run up to 64 hardware threads. Communication between cores can occur using a crossbar network and cache coherency is directory-based. Epiphany [12] is a new NoC architecture with small 32/64 RISC cores each of which has its own 32 KiB addressable local memory, while all of them can access a shared off-chip RAM. Epiphany NoC completely discards any caches, and all the communication on the NoC is message-passing oriented. The founders of Epiphany claim that their architecture can be scaled up to millions of cores, with the only limitation being the size of the address space (hence, number of addressable cores, since every core has 1 MiB address space).

2.2 Operating Systems

OSs traditionally were the lowest level of software that is managing the hardware directly. They provide all the required functionalities that applications need in order to execute and do their job in a way that makes the hardware transparent to application developers. The core part of an OS that is dealing with hardware directly is usually called the *kernel*. Different kernel designs have been implemented, each with its pros and cons. Some kernel designs are better working on some hardware architectures than others. In this section a few well-known kernel designs are discussed along with their abstractions and, when it applies, some of their features that this thesis is concerned with.

2.2.1 RTOS

A Real-Time OS (RTOS) is mostly concerned with determinism and predictability. That is, knowing when a task would be triggered, its response time and when and for how long it can handle a specific request, all of these are crucial aspects of an RTOS. It may even sacrifice performance for the sake of predictability. There are two super categories of RTOSs (or real-time systems in general): hard real-time, and soft real-time.

In hard real-time systems, a task has a strict deadline; a system will fail if this deadline was not met. On the other hand, in soft real-time systems, some tasks have (soft) deadlines that can be missed, and the system would still operate correctly. Depending on application requirements, hard and soft deadlines for tasks are determined.

RTOSs have applications mostly in safety-critical embedded system world like: aerospace, medical devices, vehicle control, military applications, robots, cell phones, etc. There are many RTOSs, RTEMS [82] is a successful open-source RTOS that has made it to planet Mars [109, 45], and is discussed in details in the following section as an example of an RTOS. Other RTOSs share the same features with RTEMS, especially when it comes to scheduling algorithms. RTOSs may need some special hardware features like counters and timers.

2.2.2 Exokernel

Exokernel [46] design motivates the idea that applications are the best to know their requirements, and hence they can achieve great performance boost without being restricted to the underlying traditional OS interfaces. With exokernel, application developers should be aware of the hardware they are dealing with, thus, making application programming to require more knowledge and effort. Exokernel draws a firm line between what a kernel does, and what applications do. The exokernel is only responsible for security/protection and multiplexing hardware resources, no abstractions, management or policies are included part of the exokernel layer. It then exposes the hardware resources to applications, enabling them to have direct access to manage the hardware in an efficient and optimised way, given that they know what they are doing. This way, the exokernel implementation can focus on the security, and reduces the number of potential bugs, given that it would be very small, and hence can be easily debugged. The application level is the layer that is fully responsible of any management and policies, and it can be as big as a full libOS (library OS), that entirely operates in user-level mode. The library or applications are referred to as untrusted software. Unlike microkernels (discussed later), exokernel excludes virtual memory management and Inter Process Communication (IPC) from the kernel layer, and leaves it to the libOS. This is an issue with conventional hardware architectures because some operations (like virtual memory management) can not be done in user-level, and applications would need to trap to the kernel anyway. This is an example of how a hardware architecture may prevent a kernel design principle from being implemented.

For an application and/or a library OS to execute, exokernel has three main mechanisms to expose hardware resources through a secured, well-defined API:

- Secure Binding.
- Visible Resource Revocation.
- Abort Protocol.

Secure binding is a way to bind a libOS to a specific resource upon request. So, a libOS would ask for a given resource from the exokernel, and then the exokernel would check for potential protection violations, and if there are not any, it would grant the resource to the libOS in the

form of encrypted key, and subsequent accesses to this resource from the granted libOS would no longer require exokernel intervention (that depends on the type of resource). Secure binding can be implemented in three ways: 1) hardware, 2) software caching and 3) download code into the kernel.

Resource revocation is an exokernel operation that takes back a resource given to a libOS and may claim it to another libOS. The revocation process notifies the libOS that has the resource which is to be revoked with hardware information (e.g. page frame number to be taken back) so that it can take the proper actions. Code can be downloaded into the kernel by a libOS so that every time a given resource is to be revoked, the exokernel would automatically invoke this libOS handler in its kernel context, without the overhead of notifying the libOS.

Abort protocol defines the actions to be taken if a libOS failed to handle a resource revocation request from the exokernel; in this case, the exokernel takes the appropriate actions such as taking back the resources by force and/or killing the failed libOS and its associated applications.

Exokernel design is simple, small, scalable and secure, it can act as a hypervisor, the main issue is that it has not been researched well enough, and it can be considered as an extreme version of a microkernel, not a totally different design. Furthermore, due to the feature that enables applications to download code to the kernel to increase performance (e.g. exception handlers that might implement policies), exokernel tends to be converted to microkernels or even monolithic kernels. exokernel might do well on distributed multi-core systems (with low memory resources) thanks to its small-size and message-passing nature, but again, this has not been tried out or researched yet.

2.2.3 L4 Microkernel

L4 Microkernel is the fourth generation of Liedtke's microkernel [69] design. Microkernels were created to reduce the complexity of traditional monolithic OSs that try to implement every possible abstraction an application may request. Instead, microkernel embraces the idea of only

implementing the required code to manage the hardware in a privileged manner, and move any other abstractions/policies (that do not require privileged access) to user-level. This way, the kernel size would be minimised and it would be easier to debug and maintain. Because other discussed kernel designs (like exokernel, FOS and multikernel) tend to be microkernel-like and/or derivatives, microkernel topic is discussed in more details.

A microkernel should guarantee subsystems independence and integrity. Independence means that it is possible to create new subsystems that can not mess with each other's contexts. The main abstractions of a microkernel are:

- Address Spaces.
- Threads and IPC.
- Unique Identifiers.

A microkernel should act as a manager of the underlying hardware memory protection/translation mechanism (e.g. MMU). It should abstract away the management of such resources transparently. This means that a microkernel would keep track of each user-level thread mappings (i.e. page-tables mapping and/or TLB entries). Operations on a given address space are exposed by a microkernel to enable memory management handling in user-space, thus, this supports building new address spaces by user-level applications without knowing the hardware details. The microkernel achieves this goal by providing some operations associated with an address space:

- Grant: Means that a thread that owns a page frame can give it to another thread, and remove it from its own address space.
- Map: Unlike grant, mapping a page would enable the owner thread and the other thread that the owner maps one of its pages into its address space to access the same page.
- Flush: Revoke the pages (by the owner) that were given using grant/map operations from non-owner address spaces.

Threads and IPC are major characteristics about microkernels, and they are important topics when it comes to trade-offs and performance

analysis. IPC is the way to exchange messages between microkernel-based subsystems (between threads with each other, and/or between threads and the microkernel). IPC is a critical microkernel operation as it is the standard way of communication and its implementation has a great effect on the overall performance of a microkernel-based system. Thus, it should be carefully implemented and optimised to decrease its inherent overhead. In fact, one of the major criticisms for microkernels is its performance degradation due to IPC compared to monolithic kernel function calls.

The microkernel should also provide a way to generate unique identifiers for its subsystems. An example is when a thread wants to send a message to another one, it will use an identifier as an address for the destination thread. Similarly, an address space may have identifiers that can be useful during context switches (ASID).

History and changes of L4 microkernels

L4 microkernels, originally developed by Liedtke [69], have been in existance for over 20 years. Since then, microkernels have been improved and it was proven that the concepts of microkernels are not only applicable, but also competitive to other kernel designs from a performance perspective. Some implementations of L4 microkernels have been deployed to billions of mobile phones and safety critical systems [44], asserting microkernel capabilities as well as security advantage compared to other kernel designs. A time-line of microkernel implementations during the last 20 years is shown in figure2.3.

Some of the design principles have changed since Liedtke's paper that introduced [69] L4 microkernel, but all of the L4 derivatives adhere the minimality principle. Elphinstone and Heiser [44] discus the trade-offs and changes to microkernels in details, some of them are given below.

Long IPC Long IPC in L4 was a way to transfer long strings from a thread context (sender) to another (receiver). The problem with this long IPC is that when doing a send syscall on an IPC context of the sender that may not be fully mapped, the kernel may incur a page fault during the copying process. This is not a desired behaviour as the kernel assumes all interrupts and concurrency to be disabled. Moreover, L4 terminology is to handle page faults in user-level, not in the kernel, this

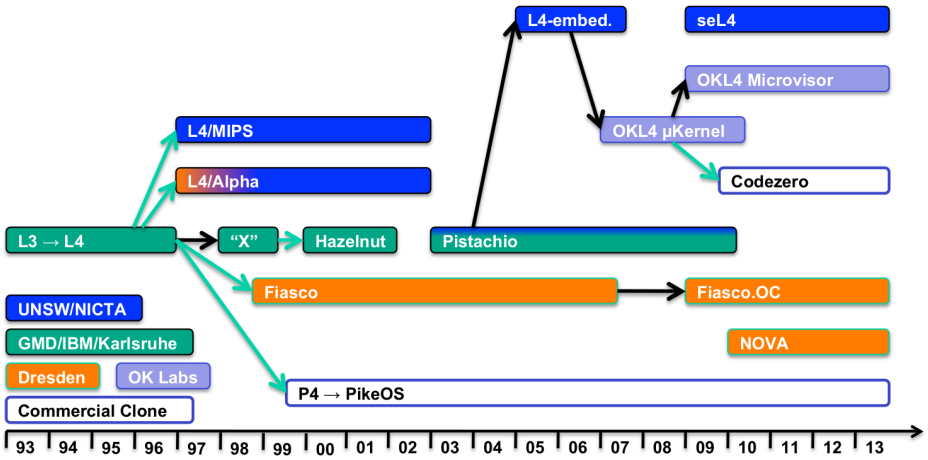


Figure 2.3: The L4 family tree (simplified). Black arrows indicate code, green arrows Application Binary Interface (ABI) inheritance. Box colours indicate origin as per key at the bottom left. [44]

user-level handler may use one or more syscalls, hence it raises nested exceptions in the kernel, a thing that is not desired. That is why almost all of the L4 kernels have discarded long IPC features, and used shared buffers.

IPC timeout has been abandoned in seL4 and OKL4. The main reason is that there was no way to determine the exact time for a timeout. Instead of making the kernel choose an arbitrary timeout, this can be done by the user for example using a timer's wait system call.

Synchronous IPC have been the major model so far, but recently some of the problems have appeared due to this model when it comes to multi-core scalability and multi-threaded approaches. For example a thread that wants to communicate with another thread will have to initiate an inter-core request (which is costly) and blocks waiting for a reply. The solution was given by seL4 by introducing asynchronous IPC.

As discussed before, original L4 model used thread IDs for communication between threads. There were some problems associated with thread IDs like poor performance and the possibility that a server may reply to the wrong thread. This was replaced in seL4 for example using a new **endpoint** implementation. Endpoint is a middle-ware entity that

accepts requests from senders and sends it to receivers. Each thread that wants to use any sort of communication should have a capability to a given endpoint (discussed later in seL4 example).

The Clans and Chief model that manages the way IPC occurs was believed to be very insufficient. This was replaced by capability access control in seL4 and Fiasco.OC

Microkernels have been known for their security, scalability and virtualisation features. To enforce security, they need some hardware units like MMU and different CPU privilege levels. Thanks to their minimality, modularity and message-passing IPC design principles, they can work greatly on DSM, multi-core and NoC-based hardware architectures. This might be the proper time for microkernels to rise again in the era of multi-core systems.

2.2.4 Multikernel

Multikernel [23] is a new kernel design originally proposed by ETHZ Systems Group and Microsoft Research. The authors of the paper argue that current OS designs will not cope up with the increasing number of cores and the MPSoC revolution, and propose the multikernel design as a solution that introduces new principles for an OS to be scalable:

- Inter-core communication should be explicit.
- OS should be hardware agnostic.
- No shared data structures, instead the state would be duplicated.

The main motivation for such a new design is that experimental (and practical) results exposed the inefficiency of current shared-memory OS designs that assume specific hardware features such as cache-coherency. The results doubt that such OS designs on cache-coherent shared-memory systems will scale to more than 80 cores. The reason is mainly due to cache update contentions. Moreover, the concept of shared data structures that OSs have been adopting limits scalability opportunities. When moving from the single-core design to multi-core, OSs had to implement new multiprocessor locking protocols. Such protocols either assume a given specific hardware design (ccNUMA), or they would not scale well. Besides, when re-factoring an OS that was written for a single

core system, to work on a multi-core environment, locking granularity becomes a bottleneck for scalability, and for software developers, and would introduce more bugs (i.e. a developer may forget to protect a data structure with a lock).

For heterogeneous systems, it is almost impossible to port a traditional OS to work with different Instruction Set Architectures (ISAs) at the same time, hence this restricts an OS to support only a homogeneous SoC, while this SoC may have other co-processors or cores that can be utilised to increase performance for custom application needs.

For the previous reasons, multikernel design adopts the no-sharing and hardware agnostic principles. No-sharing would avoid the use of locking at all, and consequently, its hardware dependency. Instead, (asynchronous) message passing is used for communication between different cores (no implicit cache-coherent transactions assumed). While there are arguments between the performance of shared-memory versus message passing models, the paper shows that message passing outperforms shared-memory when the number of cores increases (see figure 2.4). This way, a multikernel OS can resemble a distributed system and adopt the scalable algorithms that were invented in such a field, but this time, at the level of the SoC.

The main issue for the multikernel design, while it is a new promising one, is that its few implementations (mainly Barrelfish that was given as the first multikernel prototype in the same paper) are still restricted by the hardware limitations like cache-coherency despite the multikernel hardware-agnostic abstraction. Barrelfish has only been ported to Intel-based architectures and ARM. It would be more convenient for a multikernel implementation to work on FPGAs in order to easily discard the hardware limitations introduced by ASIC processors, and to add new features on the hardware level that fit with the multikernel requirements. Moreover, like exokernel, multikernel can be considered as separate instances of a microkernel running and communicating with each other. In fact, seL4 and Quest (discussed later) OSs can be configured as multikernels while they are originally microkernels.

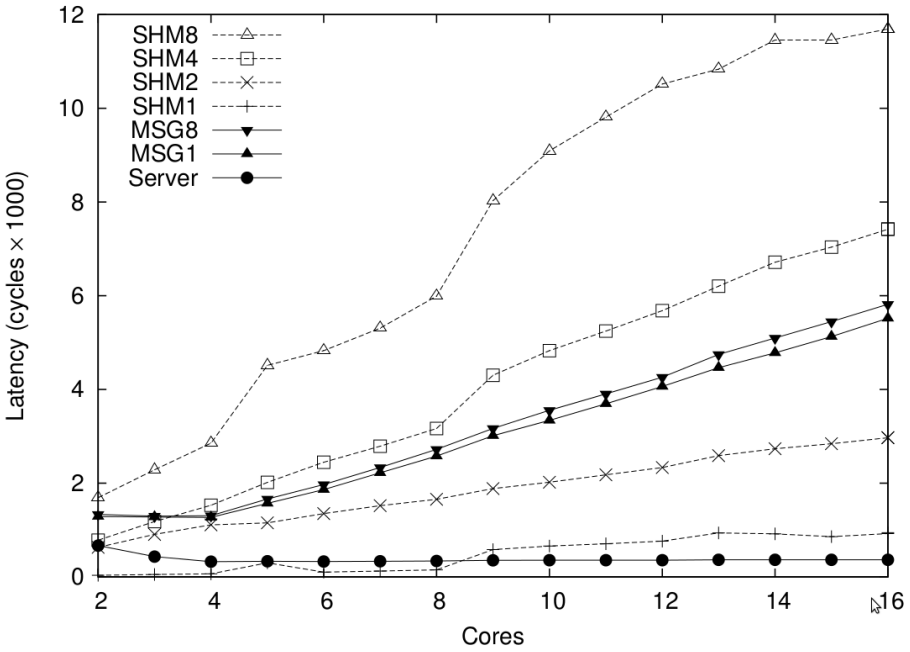


Figure 2.4: Comparison of the cost of updating shared state using shared memory and message passing. The curves labelled SHM1–8 show the latency per operation (in cycles) for updates that directly modify 1, 2, 4 and 8 shared cache lines respectively. The curves labelled MSG1 and MSG8, show the cost of synchronous RPC to the dedicated server thread. [23]

2.3 Multi-core Operating Systems Examples

2.3.1 RT PREEMPT and LITMUS - Linux

Despite being a monolithic kernel that is intended to run on desktops and servers, Linux has been patched to support hard real-time tasks via a patch known as RT PREEMPT [6]. Originally, Linux was only supporting soft real-time tasks. RT PREEMPT patch introduced some (relatively) minimal changes in order to convert the Linux kernel into a real-time one. The changes are mainly concerned with enabling kernel pre-emption within contexts like: spin-locks, critical sections, interrupt handling and interrupt-disable code.

The patch also implemented high resolution timers and timeouts, making it possible to support high resolution POSIX operations in user space. Moreover, it preserves priority inheritance implementation within kernel mutexes and spinlocks.

Some of the RT PREEMPT patch has made it to the Linux kernel mainline, while the rest is still kept synchronized against every release of the Linux kernel in a set of patches. The RT PREEMPT patch is currently maintained and has its own web-page [6] with details how to get the patches, apply it and even run a real-time hello world example.

LITMUS [36] is the testbed based on Linux that investigates and analyses real-time scheduling algorithms (global and partitioned) like Pfair [58] and Global EDF on SMP multiprocessors, and compares their performance.

2.3.2 RTEMS

RTEMS, Real-Time Executive for Multiprocessor Systems [82] is a free open-source RTOS. It has been ported to over 16 architectures and about 180 Board Support Packages (BSPs), and it includes API standards such as POSIX. RTEMS has been widely used for automotive control, robotics, medical devices, aerospace, military and industrial applications. Most notably, NASA [7, 8] has utilised RTEMS. More details are given about RTEMS (here and in the next chapter) as it is used part of this thesis as a representative monolithic RTOS.

RTEMS is modular and extendible [9, 49], and is structured as a set of managers (e.g. for semaphore, IO, barrier and memory), with a core manager used by other managers.

RTEMS has support for many scheduling policies. The scheduling framework in RTEMS [29] allows for user defined pluggable policies. Standard RTEMS schedulers include: priority, earliest deadline first and constant bandwidth scheduler.

SMP Support on RTEMS

RTEMS has basic support for SMP [83] on the SPARC [37], ARM, x86 and PowerPC CPUs.

The prerequisite for porting RTMES SMP support to a new architecture is the provision of an atomic operation. This can be achieved using atomic operation support in GCC (if available for the platform), or by extending RTEMS. Also, additional low-level functions are required within RTEMS, including synchronization management via ticket locks.

Platform dependent core identification functions are required, used for a purposes like distinguishing between the boot processor and other secondary processors, acquiring per-cpu information, assigning a thread to a specific processor and other operations. RTEMS uses linear numbers for cores starting with 0, up to the number of cores minus one. Normally a core with the 0 ID is called the boot processor, which is the one that initialises the whole system including the secondary processors, and makes them ready for executing parallel thread applications. Some inter-processor interrupt functions must exist for cores to communicate with each other.

SMP Initialisation

Figure 2.5 summarizes the current SMP initialisation process on RTEMS. Boxes with black background refer to processes that happen on both boot and secondary CPUs. Boxes with white background are for operations done by the boot CPU only while grey background is for secondary CPU(s) operations. Operations on both types of CPUs (boot and secondary) are done in parallel until the boot CPU indicates for other CPUs to start multi-tasking. Once all CPUs have started multi-tasking, they can execute threads, communicate, exchange messages and/or as-

sign threads to each other (if supported by the configured SMP scheduler). Currently, there are three types of messages that one CPU can send to another one: 1) shut-down, 2) test, and 3) cache management. A message type code is injected into $\_Per\_CPU\_Information$ structure associated with the targeted CPU, then an inter-processor interrupt is sent to this CPU.

A few SMP scheduling algorithms exist on RTEMS. The default one is Priority Fixed SMP which is an extension to the uniprocessor Priority Fixed Scheduler. It is a global scheduler that maintains ready threads in a shared array with each entry pointing to a queue of threads having the same priority level. Simple SMP scheduler is another extension to uniprocessor Simple Priority Fixed Scheduler. It uses just one queue (doubly linked list) to maintain ready threads. Partition/Cluster Scheduler can also be used using some utility functions. A major problem, that SMP on RTEMS currently faces, is the usage of giant global lock. This giant lock limits the peformance of the whole system especially when the numbers of cores and threads (that need frequent access to the kernel e.g. system calls and share kernel data structures) increase. Finer grained locks are needed to make RTEMS more scalable and allow concurrency within the kernel (hence increase performance).

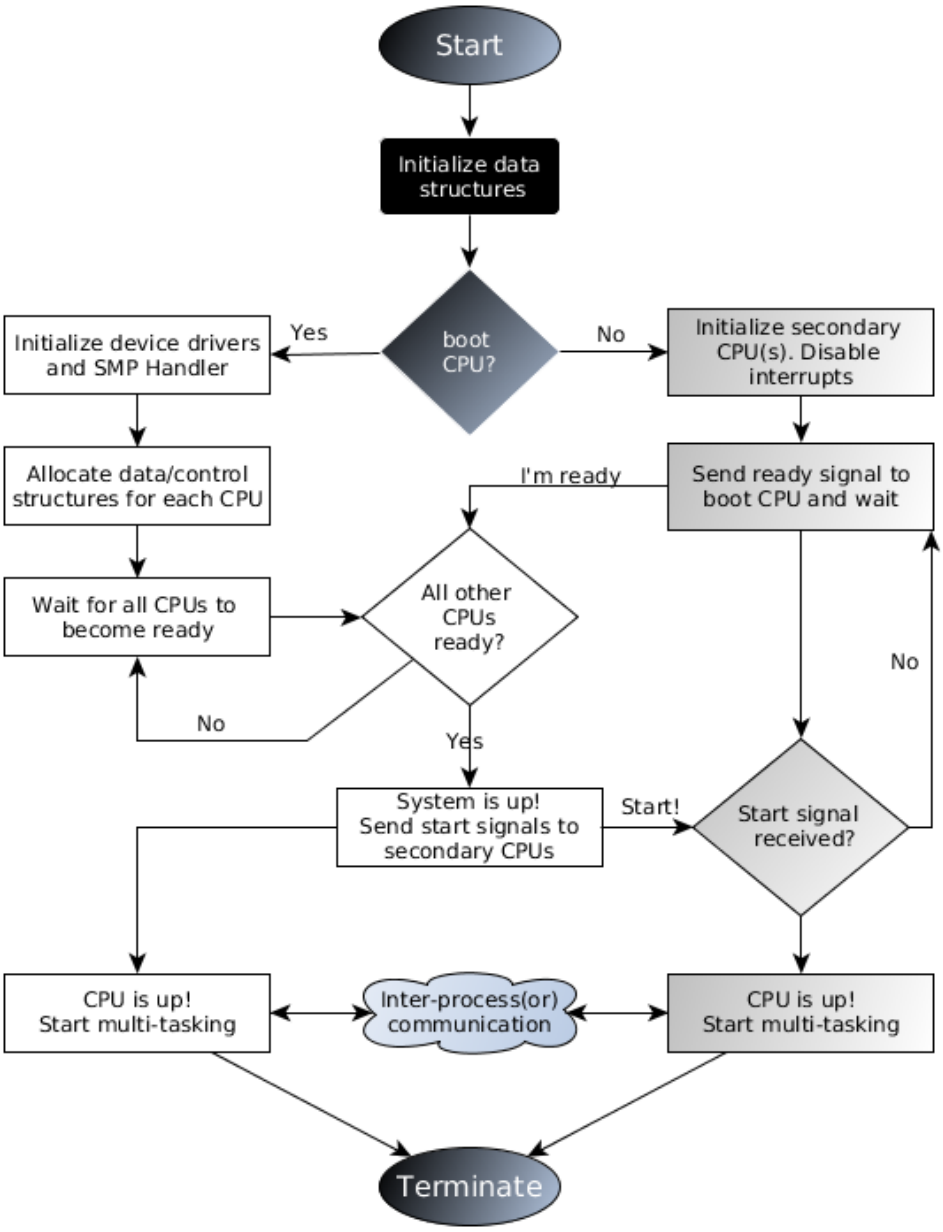


Figure 2.5: *SMP Initialization on RTEMS*

2.3.3 seL4

seL4 is a new open-source L4 microkernel developed by NICTA and now owned by General Dynamics C4 Systems. It gained its popularity being "The world's first operating-system kernel with an end-to-end proof of implementation correctness and security enforcement and is now open source. [107]" seL4 developers believe that it is the state-of-art L4 microkernel currently. L4 simplicity concept has been achieved in seL4

given that it has about 10K lines of C code, compared to Fiasco.OC which has 36K lines of C/C++ code. Currently seL4 is ported to only two architectures: ARM and IA-32 [2]. Only the ARM port is formally verified, and both support only 32-bit implementation, however the 64-bit implementation is a work in progress. The IA-32 port supports booting in multikernel mode unlike the ARM port.

There are different types of kernel abstractions:

- CNode
- TCB
- Endpoints (synchronous and asynchronous)
- Virtual Address Space
- Interrupts
- Untyped memory

CNode, is a key implementation of the capability-based management for access control. A CNode is a table of slots that may contain other CNodes or capabilities. Like the hardware page-tables, CNodes are constructed in multi-level tables layout. Each capability has its exclusive address within the task, provided its root CNode. This address is used to refer to this capability and do operations on it (i.e. mint, copy, revoke, etc).

TCB (Task Control Block) is a control structure for seL4 threads. Each thread has its own kernel TCB that contains related kernel data about this task. TCBs have fixed size for each architecture.

Endpoints enables tasks to communicate with each other and exchange messages through the IPC buffer and/or physical message registers. There are two kinds of Endpoints: synchronous and asynchronous. Synchronous endpoints are mainly used for communication between threads, while asynchronous endpoints can be used for interrupts delivery.

[2] RISC-V port is already there but has not been upstream (yet).

Each task has its own address space as with most OSs[3]. Depending on the architecture, address space structures are constructed and managed through the kernel. The kernel keeps track of each task's address space (page-tables) and ensures that there are no memory violations between tasks and other seL4 system components. This is a major abstraction for microkernels in general.

The kernel receives hardware interrupts and exceptions, and sends them to the right thread (that may contain a handler) through an asynchronous endpoint. There is a table for each IRQ that keeps track of IRQ states.

After seL4 has mapped its own kernel image and user-level frames, the remaining unused memory is given to the root task as untyped memory. Untyped memory can be then used by the applications to allocate new objects, and manage them, without the overhead of kernel syscalls.

seL4 is used as a representative of microkernel design in this thesis. It is discussed in more details in chapter 4.

2.3.4 Quest

Quest [104] is a new OS that can be configured for uniprocessor, SMP, or as a distributed multikernel in multi-core systems with predictable performance. It tries to make use of hardware virtualization capabilities as well as supporting real-time oriented events. The need for applying totally new OS policies and implementations motivated the authors of Quest to create it from scratch to avoid dealing with complexity of the existing large OS code (i.e. Linux) and not to go through conflicts with traditional OS policies like UNIX. So, Quest has the freedom to make its own design and implementation decisions. The most important resource from Quest OS perspective is time, thus, it supports real-time tasks with time-budget virtual CPUs separated from each other. Quest can support both time-triggered conventional tasks and event (interrupt driven) tasks; making it capable of scheduling both real-time and non real-time tasks.

[3]RTEMS is using a single address space for all tasks

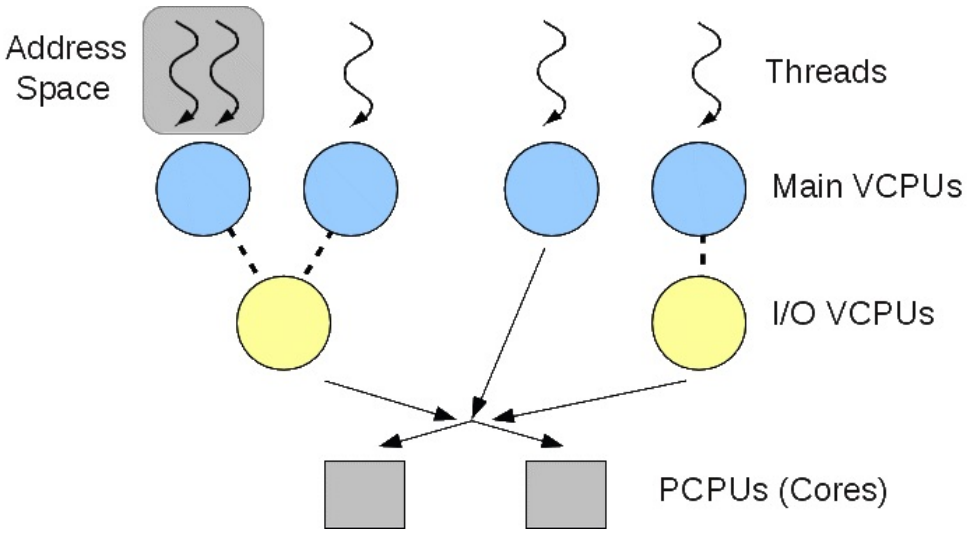


Figure 2.6: Quest Components [41]

Scheduling

The scheduling in Quest depends on a major concept of virtual CPUs. A virtual CPU acts like a physical one for each thread, and it is assigned a time-budget. VCPUS are implemented by sporadic servers. There are two types of VCPUS: 1) Main VCPUs that are responsible of scheduling threads and 2) IO VCPUs that handle IO operations associated with interrupt behaviour. IO VCPUs make interrupt handlers act like tasks in terms of scheduling and priorities. Figure 2.6, from Quest web-page [41] illustrates the relations between VCPUs and other components of Quest.

Virtualization

Quest makes direct use of Intel hardware virtualization to provide fault tolerant system and to isolate each sandbox from one another, thus, it prevents a crash in a sub-system from bringing the whole system down. Quest-V is a multikernel OS (offering virtualization features) on which each core (or several ones) has its own kernel image and hardware resources (sandbox) and can communicate with other cores on chip as a distributed system. There is a monitor associated with each kernel to help making migrations of applications easier by managing extended page-table (EPT) technology. Monitors in Quest-V differ from other hypervisors in that each sandbox has its own monitor that helps bootstrapping the kernels, and it eliminates the cost of VM-exits since the kernel will resume right away. Other hypervisors use a single shared

monitor for all of its VMs.

Quest-V and Linux

Quest-V can communicate with Linux via well-defined communication channels when Quest-V acts as a hypervisor and Linux as a guest OS. Linux and other mixed criticality systems running on top of Quest-V are separated from each other given that each of which runs on its own sandbox. Thus, Linux applications and Quest real-time threads can run on a single system. This is done by making use of hardware virtualization features (Intel VT-x, VT-d, and EPT)

Communication

Quest uses shared memory channels managed by monitors to support inter-sandbox communication. This makes it easy for real-time systems and address space migration between sandboxes to exist. To notify a sandbox with an arrival of a message, Quest uses inter-processor interrupts (IPI) and polling on status bit for a given mailbox. For example IPI is used for recovery purposes to tell other core(s) about a failure of one core. The absence of a global clock and a global scheduler creates new challenges including timing and thread migrations.

Migration

In Quest, both vCPUs and threads can be migrated between sandboxes for load balancing and performance purposes. IPI is used for handshaking between sandboxes to initiate and notify about the end of migration process. Monitors are used to map the source address space to destination address space. Figure 2.7 from [68] shows the Quest migration process.

The main issue with Quest is that it is privately maintained by Boston University, and the open-source version is quite outdated. Also, Quest heavily relies on Intel's hardware virtualization technologies, which makes it very difficult to port to another hardware architecture that does not provide the same features. No formal manuals or specifications are provided for Quest.

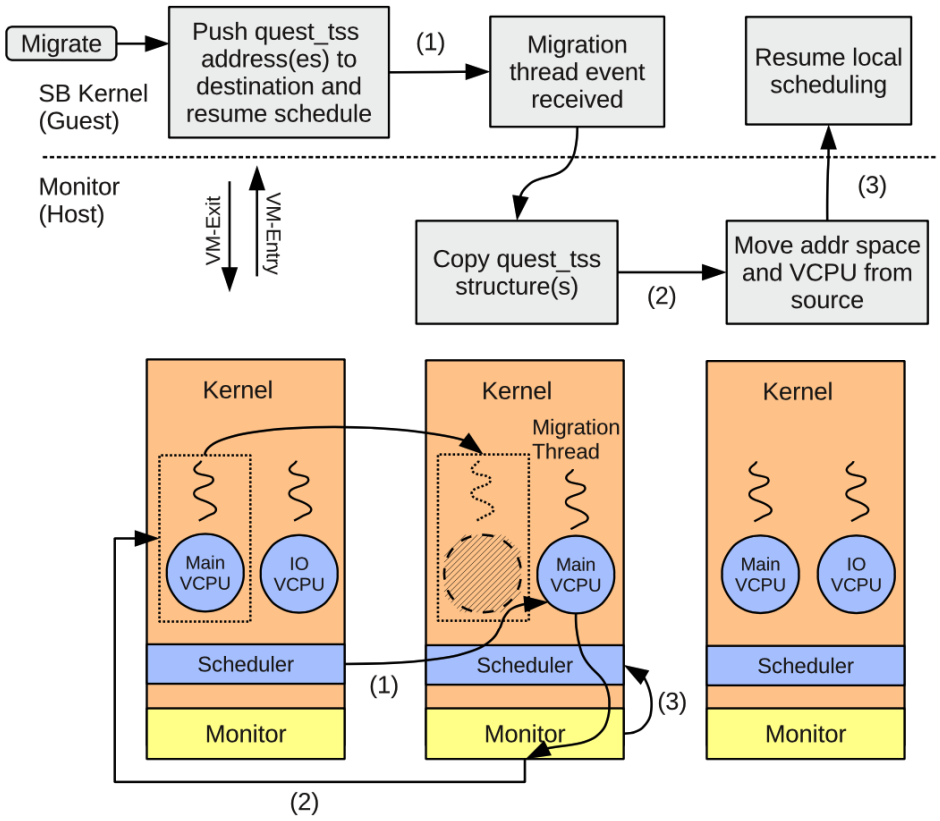


Figure 2.7: Migration Strategy [68]

2.3.5 Corey

Corey [30] is an exokernel OS designed for many-core systems and runs on AMD Opteron and Intel Xeon. It tries to make access to shared data structures as minimum as possible. For example, it bounds accesses to shared memory to only one core. Also, Corey makes use of the many cores to create a set of them only to handle some specific functions/threads. It also sheds lights on the importance of giving applications the rights to control data structures instead of leaving the kernel managing these data structures exclusively without previous knowledge of its application needs.

Corey Abstractions

Corey defines three abstractions to give applications the control over (shared) data, given that they know their sharing needs. These abstraction are:

- Address Ranges
- Kernel Cores

- Shares

Through address ranges, Corey makes it possible for applications to specify some address ranges that can be private and others that are shared for each core to avoid contention. If an address range is private, no contention will happen and this will avoid TLB shootdowns. On the other hand, if an address range is shared, the application specifies which cores use it, and contention is limited to these set of cores only. Figure 2.8 explains how address space ranges work. MCS locks [72] are implemented by Corey to handle synchronization and accesses to critical sections. MCS locks have a great performance benefit over Linux spin-locks especially when the number of cores increases as shown in figure 2.9.

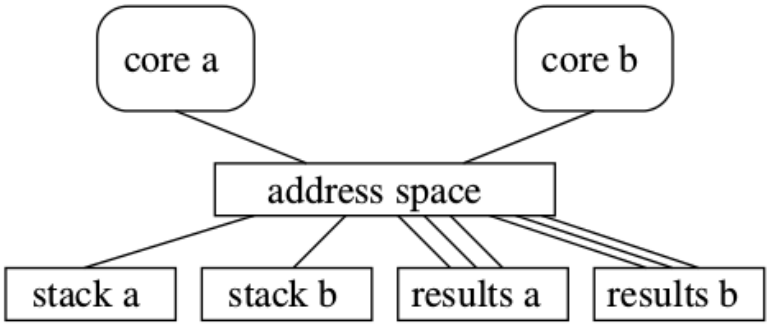
To avoid contention, applications can bind specific kernel functions and data to a given core by making use of *kernel cores*. The core exclusively has access to the code/data of a specific kernel function (e.g. device drivers), and can communicate with other cores requesting a service from it or delivering interrupts by a shared memory IPC. This mechanism enables the kernel core to keep the code/data in its cache and avoid TLB invalidation. No shared data structures or locks are needed this way to access this device as only one core uses it.

Every application can create a *share* that tells the kernel which part of its data can be shared with other cores. These shares make it possible to separate private data from shared ones. For example an application may share memory with other cores by having a shared mapping (page-table).

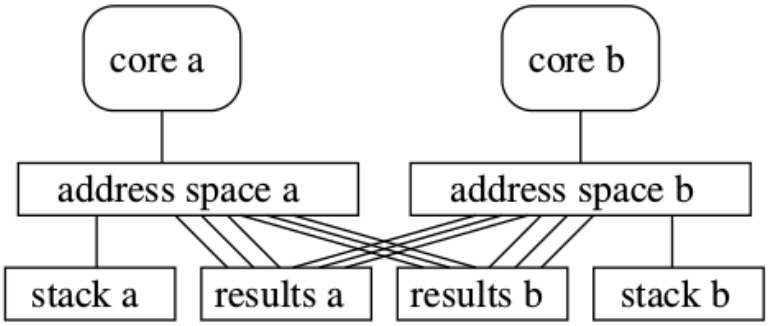
Corey is very good with hardware architectures that provide MMU (including page-based management and TLBs), as well as caches. However, some of its abstractions might not work (well) on architectures that lack such hardware units, or provide other alternative ways of memory management.

2.3.6 FOS

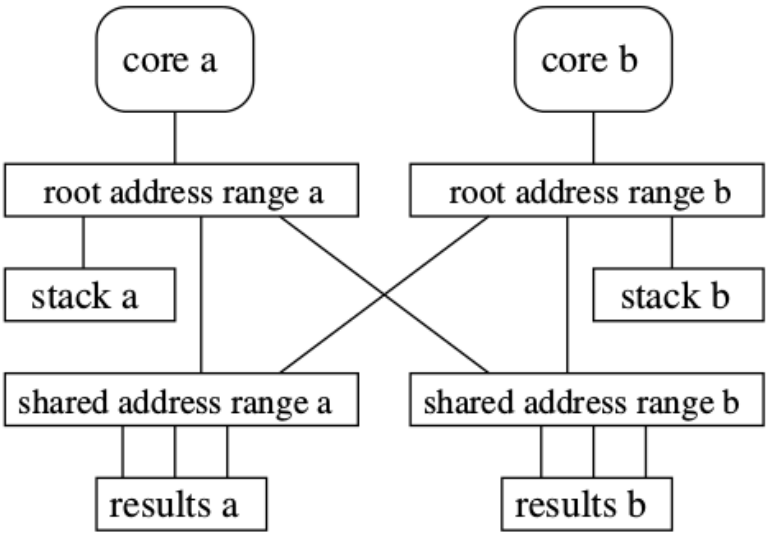
Factored OS (FOS) [103] is a new kernel design developed by MIT. FOS takes OS scalability as its highest prioirty challenge that drive its design aspects. The authors argue that a new OS design is needed to work on future 1000+ core chips, and propose FOS design and implementation as



(a) A single address space.



(b) Separate address spaces.



(c) Two address spaces with shared result mappings.

Figure 2.8: Example address space configurations for MapReduce executing on two cores. Lines represent mappings. In this example a stack is one page and results are three pages. [30]

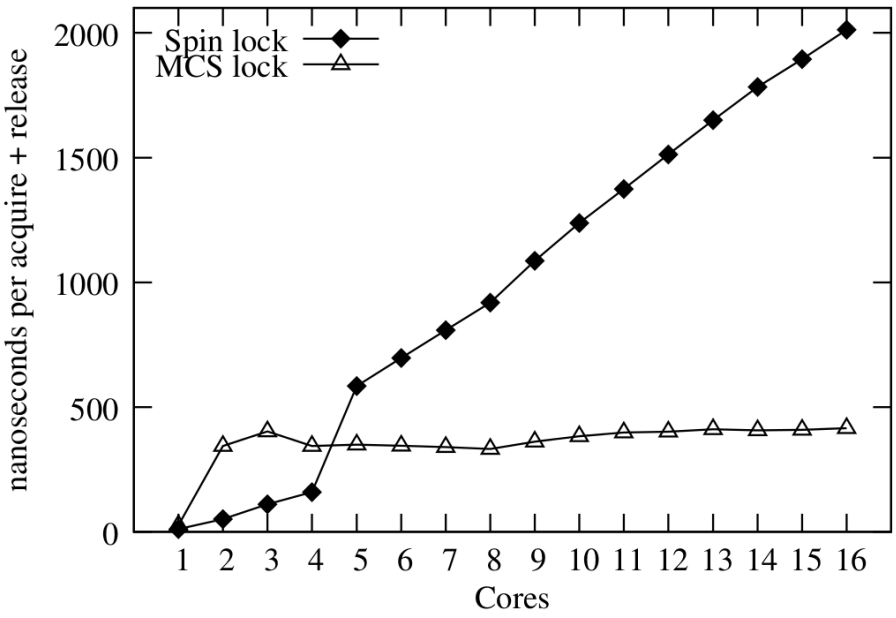


Figure 2.9: Time required to acquire and release a lock on a 16-core AMD machine when varying number of cores contend for the lock. The two lines show Linux kernel spin locks and MCS locks (on Corey). A spin lock with one core takes about 11 nanoseconds; an MCS lock about 26 nanoseconds. [30]

a prototype.

FOS is inspired by scalable internet services and L4 microkernel designs, and it tries to replace the traditional time-sharing-oriented OSs (that were originally designed for uniprocessors) with a space-sharing design where services are pinned to physical cores and requests are made to these services from other application cores. By having exclusively allocated cores for FOS kernel's services, it is possible to avoid contention between applications and kernel services on resources like TLBs and caches, thus, providing a scalable, high performance OS.

FOS has defined its design principles as follows:

- Space multiplexing replaces time multiplexing.
- OS is factored into function specific services.
- Servers collaborate and communicate only via message passing.
- Servers are bound to specific cores.
- Applications communicate with servers via message passing.

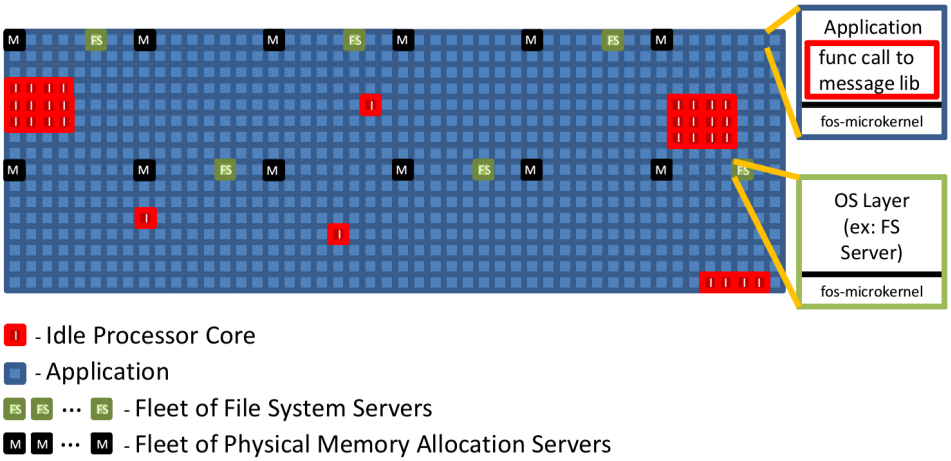


Figure 2.10: OS and application clients executing on the fos-microkernel [103]

- Servers leverage ideas (caching, replication, spatial distribution, lazy update) from internet servers.

The architecture of FOS is shown in figure 2.10. One major similarity between FOS and multikernel design is that both embrace share-nothing (no locks) and message passing techniques.

FOS design is inspired by microkernels design, and it provides system services across one or more cores composing a fleet that acts as a distributed server (i.e. page allocator). Communication is done through message passing technique, which is known for its scalability.

2.4 Operating Systems Design Issues on Multi-core Architectures

With the rise of system on chips, and the need for custom IP cores that are doing some specific functions better than others (for example, GPU chips are better than general purpose CPUs with graphics), both SoC and OSs (including device drivers), have to handle this new heterogeneous architecture model. It is no longer valid that OSs assume to be designed for single core, or even homogeneous multi-core architectures; they have to evolve.

Heterogeneous architecture can be a mix of CPU, memories, GPU, NoC, UART, accelerators, DMA, FPGA, etc. each with its specific func-

tion. In fact, a heterogeneous system can be many CPU cores, with the same ISA/ABI, but different implementations, and features. An example of this is the new RISC-V architecture, with one core acting as a master with 3 levels of privileges that are able to run a complete OS, while there are other application cores that can run only user-level code with only one privilege level. Similarly, Parallella/Epiphany architecture that has a master ARM core(s), as well as FPGA, memory, Epiphany cores (as accelerators), and others.

The main problem is that for each new heterogeneous architecture, the designers are inventing new mechanisms for addressing and communication between such heterogeneous cores. Not only some of these mechanisms need a lot of effort from OSs developers to adapt (typically device drivers), but also some of them may not be valid from some OS design principles and implementation.

There are major aspects of a multi-core SoC to consider when developing OSs:

1. Memory System
2. Virtual/logical Address Spaces
3. Cache Coherency
4. Communication and addressing of other cores
5. IO
6. ISA

2.4.1 Memory System

Memory is a major component (if not the most important one) to consider from both hardware and software sides. Physically, memory aspects including manufacturing technology, number of ports, data rate, alignments, physical structure whether one piece of hardware or distributed ones, protection, all of these aspects affect how a complete system would work. A single change to one of the previous attributes may lead to a complete re-factoring of one (or more) OS components (e.g. memory management library).

To give an example of how memory system affects an OS, let's consider memory protection. Traditionally, almost all of the current OSs

(including the ones mentioned in this chapter) assume that there is a memory management unit (MMU), which is page-based. This means that the smallest memory unit to deal with when applying protection attributes is of a page size, which is typically 4 KiB, and different processors can be configured to support different sizes. Linux, seL4, Quest, and Corey all assume this page-based protection, and they build on it to provide other features like multi-tasking, virtualisation and demand paging for instance. This MMU unit works fine with monolithic and microkernels, but does not do very well with embedded systems, RTOS, and distributed memory systems. The main reasons are the overhead of (n-level) address translation, TLB contentions (cenetralised hardware structure and limited number of entries) and arbitrary page-size granualities. 4 KiB is, in most architecure, the smallest page size, which might be too big for embedded systems with low memory resources and require n-level page tables translation.

RTEMS as an RTOS does need some form of memory protection for some applications, but in most cases, it can not afford the cost of translation and TLB contention that conventional MMU units enforce. Moreover, being deployed in embedded systems, the memory sizes would not be as big as several gigabytes as with desktops. Hence, dealing with 4 KiB page size as the smallest unit causes a huge waste of memory resources. The previous two reasons result in RTEMS discarding using (dynamic) memory protection to best use resources and meet its real-time requirements. Clearly, this affects RTEMS design to exclude dynamic memory management library for most of the supported architectures. Dynamic memory management simply means that RTEMS provides support for users to allocate and enfornce protection attributes on memory regions, and update page tables in run-time. This process is costly as it needs to take into consideration the overhead of updating page tables, consistency of page table entries and TLBs (especially if working in SMP environment). Instead, RTEMS sets up page tables and MMU at start-up and bootstrapping process with fixed mapping and protection attributes.

Physically distributed memory introduces other challenges and complexities but improves scalability. Hardware designers have to provide a standard for this model whether to make it transparent for programmers or to provide consistency models with new (atomic) instructions and requires programmers to ensure data consistency themselves to some extent. OS developers then have to re-design their

shared data structures, taking into consideration the rate of contentions, delays, consistency and correctness. Data structures include page tables, which have to be consistent for the shared pages. Imagine 100 processors are trying to update a shared page table/entry at the same time, this is certainly a bottleneck especially when scalability is a requirement. This is due to the fact that (shared) page tables are centralised shared data structure. Because of this bottleneck, multikernel and FOS that were discussed try to avoid any sharing. Even though, the multikernel has to implement a form of software agreement protocol to maintain consistency of globally-viewed data structures (if any).

2.4.2 Address Space

Address space is the range of addresses (physical or virtual) a process or core can access. It is not only used for memory addressing, but it is used for accessing other cores and memory mapped IO. According to the core's register size, the maximum address space range is specified. For example 32-bit cores can address up to 4 GiB. What range of address space can be accessed by a given core/process is specified by a memory management unit, or arbitrary during hardware manufacturing. An address space can resemble logically adjacent addresses to access physically scattered cores (including distributed memories). From the OSs point of view, an address space term is correlated with a process, giving it an access control to some amount of (virtual) memory, while from IO devices and other cores, it is the range or physical addresses they can access. Register sizes affect the unit of allocation and pointers used within OSs data structures and algorithms.

2.4.3 Cache Coherency

Cache coherency is another big issue for multi-core scalable SoCs. Like with the previous example of 100 processors contending over a page table/entry, the same issue occurs with cache lines, when the 100 processors are contending over this shared line. This is clear with [72] that introduces scalable locks depending on cache coherency protocols. While MCS locks introduced in this paper are currently considered scalable, this may not be the case with 1000+ cores as FOS [103]

predicts, and cache coherency would be a crucial barrier even with a small amount of sharing. Multikernel uses MCS locks as well as cache coherency hardware to communicate between cores. Similarly, RTEMS tries to put some variables within data structures in different cache lines to avoid cache line contention and false sharing. This is clearly not portable (or requires more effort and introduces bugs) to architectures that have different cache line size, or no caches at all. Also caches have been known to affect predictability and determinism requirements.

OS performs context switches frequently between threads. Context switches affect the overall performance, and have many parameters. If an OS supports memory protection and/or cache coherency, a context switch will incur an overhead of changing the page-tables including TLB faults, and cache invalidation operations. Although some hardware solutions are provided, contention occurs due to the shared nature of such structures. That is why FOS suggests to avoid context switches (time sharing) and embrace locality principle; different subsystems (including the kernel) should only care about how to efficiently communicate and the performance of this communication (IPC) component. Figure2.11, provides IPC performance costs of almost the same IPC software implementation but on different hardware implementations. It is noticed that the hardware greatly affects a crucial microkernel component like IPC, ranging from 36 clock cycles for Pistachio IPC on Itanium 2 processor up to 2000 clock cycles for Hazelnut IPC on Pentium 4 processor.

2.4.4 Communication and Addressing

Communication between different cores on a multi-core system is the responsibility of OSs, all of the discussed kernel designs and implementations have a communication library or component. Communication is tightly dependent on hardware features provided for communication like inter-processor interrupts, shared memory, sending packets, etc. The most widely used method of communication is using shared memory mainly because it is simple, and does not require special instructions. RTEMS for example uses shared memory for its multi-processing support. However, shared memory does not do very well when it comes

Name	Year	Processor	MHz	Cycles	μs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium 2	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	Core i7 (Bloomfield) 32-bit	2,660	288	0.11
seL4	2013	Core i7 4770 (Haswell) 32-bit	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

Figure 2.11: IPC costs of different L4 microkernel implementations on hardware architcures [44]

to scalability and performance (taking into consideration the previous issues of cache coherency and page-based memory management). From the hardware side, communication media like buses, network on chips, point to point communication, all affect the way an OS on multi-core chip works. Some of the OSs may choose to use one option to communicate between cores to meet its requirements such as simplicity (shared memory), interrupts (performance), NoCs (scalability), buses, etc. Communication is also a big concern for some kernel designs like L4 microkernels because its IPC component acts as a main pillar of its design and implementation. The ability to easily (and quickly) communicate with other cores hugely affects kernel design decisions and implementations.

2.4.5 IO Management

IO management (in the form of device drivers), differs between different kernel designs in the privileged mode they execute in. Monolithic kernels like Linux include device drivers part of its components which adds to code complexity and introduces more bugs to the kernel, while in exokernel and microkernel designs, IO management is done by user-level device drivers, thus, reducing kernel code size and bugs. Device drivers have to coordinate sharing of its physical resources between different cores and processes. Also, there must be some sort of protection between device drivers, applications, and the kernel itself. A device

driver that can do DMA access can (intentionality or not) corrupt other kernel data structures, or worse, hack the system. Quest-V relies on Intel hardware virtualisation extensions to provide protection between system components and IO. Also even with exokernel design, the implementation should use such hardware security features from the kernel, and leave policy to user-level. So, providing a simple standard IO interface would affect the OS design and implementation complexity, scheduling, security and scalability.

2.4.6 Instruction Set Architecure

ISA is also another parameter for the design and implementation of OSs. Except from RTEMS, all other discussed OSs assume there is at least two levels of hardware privileged modes: user and kernel. Others may provide more levels for use cases like hypervisors or machine emulation. Privilege levels are a way to enforce protection on the CPU level coordinating with an MMU unit for example to separate kernels and applications, and providing security and integrity of a system. An ISA that only provides one level of privilege modes would not be qualified to run a typical OS like Linux or seL4.

In the next chapter, porting RTEMS as an RTOS to Parallella board is discussed. Parallella has 16 Epiphany cores connected via a network on chip and communicating with ARM, FPGA, local and global memory (with no caches) cores. This gives us the opportunity to address the issues of porting an OS that has been originally developed for single-core system, to heterogeneous and SMP embedded system architecture like Parallella, and what are the real-world issues of such a porting process and what are the design issues from OSs point of view.

Chapter 3

RTEMS on Epiphany multi-core NoC

The goal of this chapter is to assess both a monolithic kernel like RTEMS which is in its early stages of supporting SMP, and a new multi-core architecture called Epiphany, and how they fit together. Experience of refactoring RTEMS to work on a multi-core hardware with limited memory resources is reported, as well as how the hardware architecture affects the design and implementation of RTEMS, trading off speed and size.

Three versions of RTEMS were implemented: 1) a squeezed, 2) SMP and 3) factored version. The squeezed version tries to reduce RTEMS size to less than 32 KiB to fit into the fast local memories, thus keeping it monolithic but removing most of its features. SMP version follows the current ongoing development of RTEMS SMP support (and other similar monolithic kernels), assuming shared memory, this design requires using the very slow DRAM off-chip memory while preserving all RTEMS features. So the trade-off is which parts of RTEMS needed to be placed in local memories, and which in the shared external slow memory and how this affects the performance. The factored version tries to convert RTEMS to something like a microkernel by placing the kernel and device drivers on different cores using only local memories to get the benefit of speed, and avoid using the slow shared memory. This attempt was not practical due to the monolithic nature of RTEMS.

3.1 Epiphany Architecture

Epiphany [12] is a multi-core architecture developed by Adapteva. Currently, there are two variants of Epiphany chips: 16 and 64 core chips

(all 1GHz). The architecture is shown in Figure 3.1. Each CPU together with a DMA unit and network controller forms a node, with nodes connected via a 2D mesh networks, together with the memory. The memory, CPU and tools are detailed in the remainder of this section.

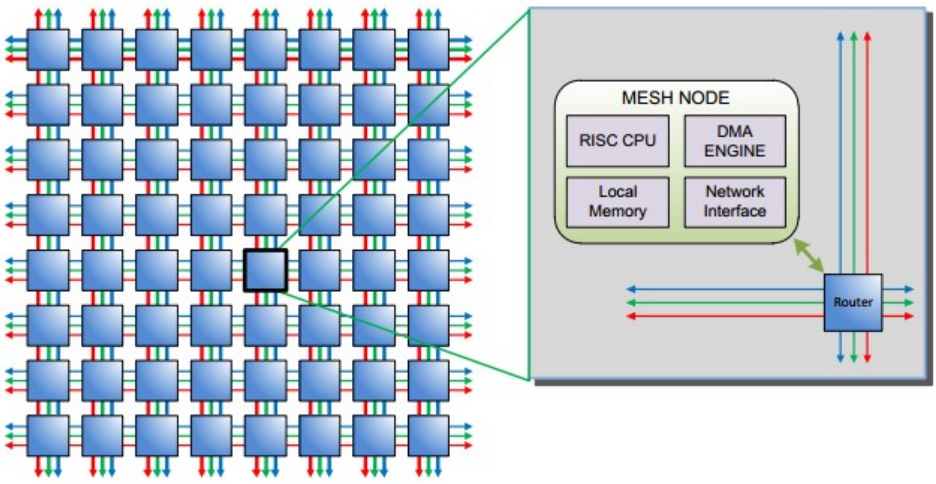


Figure 3.1: *Epiphany Architecture [12]*

3.1.1 eCore CPU

The Epiphany core (eCore) [12] is a pipelined superscalar RISC CPU designed for energy efficiency and real-time. It is a dual issue out of order CPU, with a 9-port 64-word register file, IEEE-754 compliant FPU, interrupt controller and memory protection unit.

The Instruction Set Architecture (ISA) has a variable length instructions (16 and/or 32 bit instructions). It is the responsibility of the compiler to decide which instruction version to generate depending on the instruction being executed and its operands. This is done for efficient code density optimization. Instructions are categorized into: integer operations, floating point operations, branching, load/store, data movement (between registers and immediate values only) and program flow instructions. Loads and stores are the only way to access the memory. There are different addressing modes that can be used with load/store instructions like displacement addressing, index-addressing, and post-modify addressing.

Each eCore has an interrupt controller supporting nested and prioritized interrupts. 10 different kinds of exceptions are supported, including sync/start, memory protection faults, two timer interrupts, message in-

terrupt, two DMA interrupts, WAND interrupts, user interrupts and software exceptions (FP operations, invalid instruction and unaligned memory access).

Timer interrupts can be used for scheduling purposes, eg. countdown timer for deadline monitoring. Timers can also act as event counters – eg. for profiling, idle clock cycles, number of valid integer instructions, number of valid FP instructions, dual issue instructions, five different stalls events depending on the reason of the stall and finally mesh traffic counter.

The WAND interrupt enables synchronization between workgroups of CPUs – a multicast hardware synchronisation barrier. Once all CPUs have set their WAND bits, then a WAND interrupt is generated to all CPUs in the workgroup.

Message interrupts can be used as inter-processer interrupts in OSs, and it is used in RTEMS part of the SMP implementation as shown in later sections.

The Application Binary Interface (ABI) gives some rules for Epiphany programmers that must be followed to guarantee a correct execution. For example, it lists what general purpose registers must be saved by the caller and others that the callee has to save during a subroutine call. Also, the ABI states that the stack is growing down, and the start address of the stack should be aligned to 8 bytes. Parameter passing, data types and register usage are part of the ABI.

3.1.2 Memory Architecture

eCore have a 4GB address space consisting of 32-bit words (little endian). Memory is accessed using bytes, halfwords, words or double-words according to the load/store instruction type. Software exceptions are caused by accessing a wrongly aligned address, eg. accessing a double word using a store-double instruction with an address not aligned to 8 bytes.

Figure 3.2 shows a memory map for 64 core Epiphany chip. Each core has its own local on-chip memory, whose address space ranges from 0x00000000 to 0x000FFFFF (1 MB). This is divided into 4 internal memory banks (each of which is 8 KB, with total space of 32 KB), memory mapped registers and a reserved area. The address space of each core (including memory mapped registers) can be globally accessed by all other

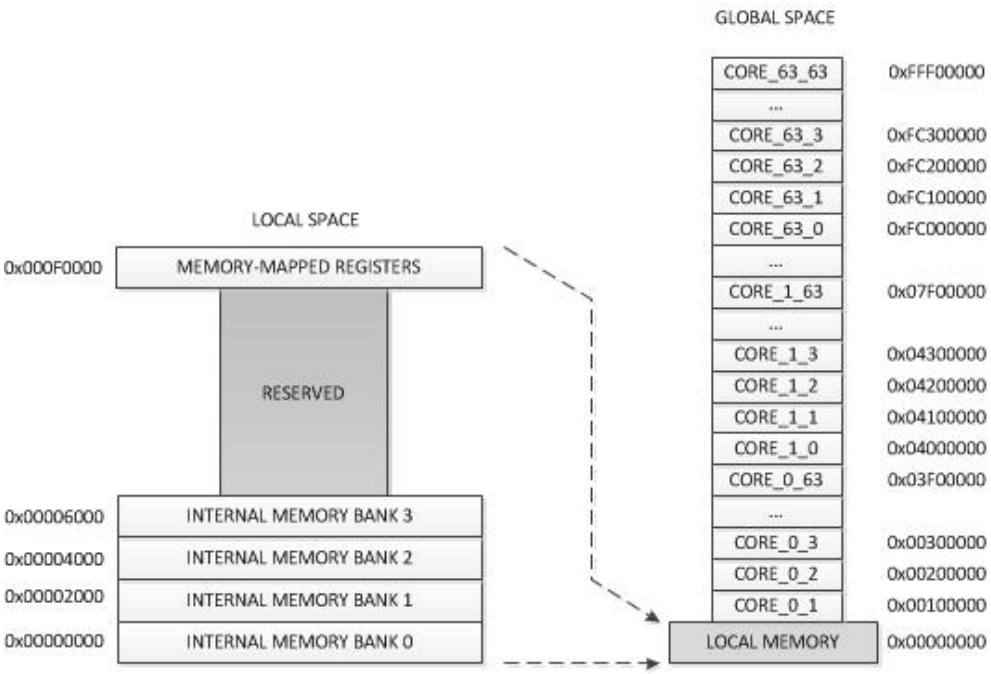


Figure 3.2: *Epiphany Global Address Map [12]*

cores – a 12-bit core ID is used in the most significant bits of the address. Note that external memory is not shown, being above the local space in the map, but dependent upon the actual architecture of the board containing the Epiphany chip.

Memory accesses can be normal read/write memory transactions, or access another core's registers (general and/or special purpose). Local memory accesses take 0 cycles, and occur in strict memory order. Accessing non-local memory is relatively costly, with weak memory order.

3.1.3 eMesh network

Epiphany has a scalable 2D mesh network (eMesh) that connects each node directly with its neighbours and operates at 1GHz frequency. Scalability is limited by address space size, eg. 32-bit address space can accommodate upto 4096 nodes (4 GB divided by 1 MB) – a 64-bit address space can accommodate considerably more.

The eMesh network consists of three separate networks:

1. cMesh – serves on-chip write transactions;

2. rMesh – for non-blocking read transaction;

3. xMesh – serves off-chip memory transactions that go beyond the chip.

On-chip writes are 8 bytes/cycle, with off-chip write transactions 16x time slower. Hence, tasks should try to communicate with each other on the same chip. The networks support direct interactions between cores by simply writing to a core's global address space. The edges of the network connect the chip to the outside world like another Epiphany chip(s), shared external memory, IO peripherals, etc.

3.1.4 SDK Environment

Adapteva provides eSDK (Epiphany Software Development Kit), including GNU tools, a multi-core GDB debugger and a functional simulator. GCC is built with the newlib embedded library. The e-lib supplied library acts as a run-time library, and includes interrupt handling, core identification, timer management, DMA handling and synchronization functions. The e-hal library runs on the host side and provides functions to control and communicate with Epiphany chip, loading programs, resetting the system and configuring/managing the behaviour of the Epiphany chip.

3.2 Porting RTEMS SMP to Epiphany

This section describes the main contribution of this chapter, namely the port of RTEMS and RTEMS SMP support to the Epiphany architecture. The target for the port is the Parallella board, model P1602 (for embedded applications). This has an Epiphany 16-core CPU (E16G301), with a Xilinx Zynq Dual-core ARM A9 XC7Z020 host CPU.

3.2.1 Toolchain and Parallela Board Setup

The toolchain used was GNU based, including binutils, GCC and GDB (specific build `epiphany-rtems4.11-*`). GCC is built with an RTEMS specific `newlib`, together with start-up code, system calls and BSPs for bare board ports. We also note that recently there has been a new approach developed for building the toolchain – RTEMS Source Builder (RSB).

As noted above, the Parallela board contains both host (ARM) and Epiphany CPUs. To utilise the board effectively for experimentation

Linux is installed on the ARM A9 as host. The RTEMS port (see next section) can then be loaded onto Epiphany in two ways:

1. `e-server` – for debugging `e-server` opens ports for every Epiphany core that a GDB client can attach to, load, execute and/or debug programs on;

2. `e-hal` – the `e-hal` library API has functions to interact with the Epiphany chip, including local memory access to load an RTEMS image.

By default, the Linux image provided by Parallella community has all the Epiphany tools installed along with `e-hal` and `e-lib` libraries, making programming the Epiphany relatively straightforward.

3.2.2 RTEMS Porting Process

Porting a new CPU architecture (like Epiphany RISC CPU) to RTEMS has to go through many stages. First, there must be a toolchain that helps compiling/building RTEMS kernel and applications. RTEMS uses GNU toolchain like binutils, GCC, and GDB. The resulting toolchain programs should be something similar to epiphany-rtems4.11-*, where Epiphany is the architecture, rtems4.11 is the RTEMS release number and finally the name of the program (gcc, g++, obdjump, nm, etc). It only differs a little from the original elf toolchain used to build bare-metal applications. Since RTEMS is targeting real-time embedded applications, GCC is built with newlib as an embedded C library instead of GCC library. Bare-metal elf toolchains that use newlib library normally rely on libgloss for providing start-up code, system calls and BSPs, however, RTEMS does not use libgloss as it provides its own implementation of the previously mentioned libgloss components. Recently, there has been a new system that builds the toolchain (for supported RTEMS architectures) from the source code; this system is called RTEMS Source Builder (RSB), and it is now the main tool for building the toolchains.

RTEMS understands how the new architecture works internally via the Abstract Binary Interface (ABI). A new directory with the name of the architecture is added to *cpukit/score/cpu*; this directory contains the definitions and/or configurations of the architecture required by RTEMS. Key definitions are held in *cpu.h* – this defines which direction the stack grows, endianness, disabling/enabling interrupts, exception

control, structures to be filled in and/or restored during context switches or ISR handling. An assembly implementation for the thread context initialisation, context switch and ISR handler are included. An RTEMS port to a new architecture also requires a BSP, containing drivers for console, clock, timers, DMA etc.

This part of the code would be shared with all BSPs that have this CPU architecture. For example, there is only one ARM directory at cpukit/score/cpu, while there are many ARM BSPs like real-view, Raspberry Pi and Beagle boards that share the same code of ARM CPU. The most important file at the newly added directory is cpu.h which contains code defining how RTEMS will deal with this architecture. It defines which direction the stack grows, endianness, disabling/enabling interrupts, exception control, structures to be filled in and/or restored during context switches or ISR handling. Then an assembly implementation for the context switch and ISR handler should be provided, along with some code that initializes thread contexts during task creation.

For a new architecture to be tested, there must be at least one BSP for it. BSPs contain code for peripherals like console driver, clock driver, timers, DMA, etc. The peripherals are part of the board that embeds the CPU architecture. Different BSPs can have different peripherals. As we will see later, there is a BSP called Parallella that depends on the Epiphany RISC CPU architecture. There is a one-to-one mapping between a hardware board and a software BSP as well as between a CPU ISA and RTEMS cpukit/score/cpu.

An initial port of RTEMS to a new architecture can be tested by using the functions in the BSP, with default test cases (eg. hello world, ticker). This ensures that the new architecture port and BSP are basically correct. There are more than 500 tests that cover almost all of the features that RTEMS may have. A new framework called RTEMS Tester, is being developed for automating running all of the 500 tests and get some results/numbers of passed/failed tests. Figure 3.3 gives an overall flowchart of the porting process of a new architecure to RTEMS ecosystem.

3.2.3 Porting RTEMS to Epiphany

In this section, the steps of porting RTEMS to Epiphany with the Parallella board as the first BSP used for verification are listed. The Parallella

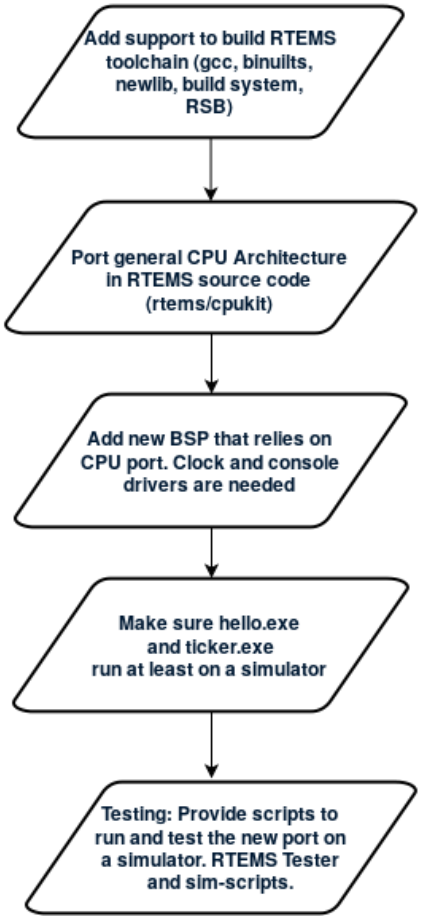


Figure 3.3: *RTEMS Porting Process*

board model used is P1602 (for embedded applications) that has a Xilinx Zynq Dual-core ARM A9 XC7Z020 host processor, Epiphany 16-core CPU (E16G301), 1 GB DDR3, and other peripherals.

First, a Linux distribution has to be installed on the Zynq chip which is provided by Parallella community and can be found on their website with detailed instructions how to burn the Linux image to the SD card and run it. The RTEMS executable can then run on the Epiphany chip from Linux on the Zynq chip. There are two ways to load/run RTEMS on the Epiphany chip, one way can be used for debugging using e-server that opens ports for every Epiphany core that a GDB client can attach to, load, execute and/or debug programs at, and the other way is to use the e-hal library API that has a lot of functions to do almost every vital operation with the Epiphany chip. Both methods can be used to run RTEMS, but C programs that use e-hal to load and manage/monitor RTEMS are more efficient and have more control. For example, it's easier to load a single RTEMS image to the 16 Epiphany cores and start them with a single line of code rather than opening 16 GDB terminal clients for each core (other than the e-server). By default, the Linux image provided by Parallella community has all the Epiphany tools installed along with e-hal and e-lib libraries, so it's ready for users to start programming Epiphany once they run Linux.

As discussed earlier, a toolchain must exist to build programs for our target—Epiphany. Almost any UNIX platform can be used to build the toolchain. Linux was used during the porting process. The toolchain includes binutils, newlib and GCC. Except for binutils, all other tools (in source code format) are checked out from the head repositories. binutils is cloned from Adapteva GitHub repository. A few set of patches has been applied to all of the previous sources to make them recognize RTEMS at the stage of configuring and building these tools. The configuration line for these tools should have epiphany-rtems4.11 at a target option. The resulting set of tools are all prefixed with epiphany-rtems4.11-* (which may be changed in future RTEMS releases). GCC is built with newlib discarding all libgloss code.

`cpu.h`

`cpu.h` (in `cpukit/score/cpu/epiphany`) contains almost all of the important machine definitions. In terms of Epiphany the important characteristics are:

- *stack* – Epiphany (eCore) stacks grown down, and are initialised at a high address.
- *interrupts* – disable and enable functions are provided in that file, making use of `gid` and `gie` Epiphany assembly instructions for global interrupt disable and enable respectively.
- *thread context* – data structure defined to hold all eCore context for context switching and ISR handling.

A related file (`epiphany-utility.h`) contains some important definitions for registers, IO memory mapped addresses and for mapping Epiphany core IDs to RTEMS CPU IDs and *vice versa*, as well as functions to get the currently executing core ID.

Context Switch and Interrupts

Context switch code is located in `epiphany-context-switch.S`, with `epiphany-exception-handler.S` containing the code for the `ISR_Handler` assembly function which is the core of interrupt handling.

Interrupt handling proceeds in the following manner:

1. Space is reserved in the interrupted task stack to save a `CPU_Exception_frame` context. Currently all of the 64 general purpose registers are saved along with `status`, `config` and `iret` special registers.
2. If SMP is enabled, another is_executing boolean variable has to be added to the context control structure to indicate whether the heir (to-be-restored) thread is executing on another processor or not, and wait until it no longer executes on another processor, then sets it. This process must be done as an atomic operation.
3. Increment nesting level and disable thread dispatching. At this point, a decision must be made if it is needed to switch the stack to RTEMS software interrupt stack depending on the nesting level.
4. Jump to the user C handler. The user C handler can be installed dynamically. NB there are default handlers that are set by the BSP at the start-up code.

5. After returning from the C handler, a check to see if a thread dispatch is necessary is made, and if so, it jumps to `_Thread_Dispatch` to select a higher priority thread.

6. The final action is to restore the context of, and jump to, the next thread by returning from the interrupt.

A context switch is essentially the same as the above, except that it does not involve executing of an interrupt handler.

Drivers

Three other drivers are required for the port:

1. *Clock driver* – required for scheduling purposes. This includes `epiphany_clock_initialize`, `epiphany_clock_at_tick`, and `epiphany_clock_handler_install`. Within this driver `epiphany_clock_initialize` installs RTEMS `Clock_isr` function address at corresponding `timer0` entry of the C handlers vector table, then it sets up the `timer0` registers and starts it. Function `epiphany_clock_at_tick` is called from `Clock_isr` at every clock interrupt occurrence and it resets the `timer0` value and runs it again.

2. *Console driver* – this driver is a way to print out data. This driver uses a shared-memory approach, communicating to Linux running on the ARM host via a shared buffer (and control bits) with Linux polling and printing when data is available to print out.

3. *Timer benchmark driver* – helps with profiling and timing performance analysis.

3.2.4 RTEMS SMP Implementation on Epiphany

Implementing RTEMS SMP on Epiphany requires initially a way to convert Epiphany core IDs, to RTEMS linear IDs. Epiphany uses 12-bit IDs – 6-bits for column and 6-bits for row. For example, a core on the 32nd row and the 8th column has an ID of 0x808. Conversion of Epiphany IDs to RTEMS linear IDs is achieved using (for the 16 core chip) [1]:

[1] Alternatively a more expensive switch statement could be used, but was felt not to be scalable as the number of cores increases.

```
movfs    r17, coreid
mov      r19, #0x003
mov      r20, #0x0F0
and      r19, r17, r19
and      r20, r17, r20
lsr      r20, r20, #4
add      r17, r19, r20
```

This is used within `start.S` to decide whether the executing core is the boot CPU and to index `_Per_CPU_Information[]` table to get the proper stack address for the secondary processor(s). Within the port, core ID 0x808 (row 32 and columns 8) is mapped to RTEMS CPU ID 0. The `ISR_Handler` also uses the above to index `_Per_CPU_Information[]` table with the current RTEMS CPU ID. The `_Per_CPU_Information` entry holds data like stacks, interrupt level, and dispatching control.

Within `cpukit/score/cpu/epiphany` the new `epiphany-atomic.c` file is added – NB currently no available GCC atomic operations for Epiphany. Within `epiphany-atomic.c` a fetch-and-increment operation on a static global variable (the one lock) is implemented.

Within `cpukit/score/cpu/epiphany` file `epiphany-smp.c` holds code for low-level inter-CPU interrupts. Here `_Epiphany_Send_interrupt` takes the Epiphany core ID and the type of interrupt as arguments to construct the global address of the targeted core's ILATS register (used to set interrupts for a core) and set the corresponding bit according to the type of the interrupt. Function `_Epiphany_Send_interrupt` calls `_CPU_SMP_Send_interrupt` with an interrupt type of `SMP_MESSAGE` which maps to the message interrupt.

At the BSP layer, a new `bspsmp.c` file is added that contains `bsp_start_on_secondary_processor` function which is called from `start.S` if the CPU is a secondary one. This function installs `bsp_inter_processor_interrupt` (that calls the generic `_SMP_Inter_processor_interrupt_handler function`) at the indirect interrupt vector table for this secondary core, and disables interrupts before changing its state to become `PER_CPU_STATE_READY_TO_START_MULTITASKING` and waiting for the boot processor to give it the permission to start.

		Local Memory Allocation			
		Whole RTEMS	ISR_Handler + Stack	ISR_Handler	Nothing Local
Execution Time	ISR_Handler	234	5275	57345	101696
	ISR_Handler + Clock_ISR	564	303915	406017	450367
Memory for Applications (bytes)		5280	31280	31280	32096

TABLE I.

		Local Memory Allocation			
		Whole RTEMS	Context Switch + Buffers	Context Switch	Nothing Local
Execution Time	_CPU_Context_Switch	283	16918	68230	98454
	_CPU_Context_Switch + _Thread_Dispatch	619	N/A	N/A	156238
Memory for Applications (bytes)		5280		31536	32096

TABLE II.

Figure 3.4: RTEMS on Epiphany Timing and Memory Analysis. Whole RTEMS column is the squeezed version where there is no use of the external shared memory. All RTEMS instances are placed in a local memory on each core. All of the other columns are of the SMP version where some parts of each RTEMS instance (mentioned in the column's title) are placed in each core's local memory, and the remaining (shared) parts are placed in the shared external DRAM memory. Execution time is given in cycles.

3.3 Performance Analysis of RTEMS port on Epiphany

In this section we give timing measurements of context switch, interrupt handling and some scheduling functions for the squeezed and SMP versions. The squeezed version only uses local memories, while SMP version uses both local and external shared memories. These are measured for different code placement strategies – ie. as the Epiphany core local memory is limited to 32Kb, we have to allocate code (and buffers etc.) across local and external memory, with the timing dependent upon allocation. Clearly, if more RTEMS is placed into local memory, then there is less space for application code, and `vice versa` – although the trade-off for memory allocation between RTEMS and the application remain for future work.

Figure 3.4 shows the timings (in cycles) for different allocations of RTEMS functionality to memory:

- *Whole RTEMS (squeezed version)* – all RTEMS placed into local memory (via source code and compiler optimisation for space).
- *Nothing local (SMP version)* – only start code and interrupt vectors placed into local memory.
- *ISR_ Handler + Stack (SMP version)* – only start code, interrupt vectors, `ISR_Handler` and stack in local memory.

- *ISR_ Handler (SMP version)* – only start code, interrupt vectors and`ISR_Handler` in local memory.

- *Context Switch + Buffers (SMP version)* – only start code, context switch and context buffers in local memory.

- *Context Switch (SMP version)* – only start code and context switch in local memory.

As can be seen in the tables, the fastest is for all of RTEMS to be placed in local memory, although this corresponds to the minimum amount of free memory for applications[2].

As mentioned previously, one Epiphany core local memory is only 32KB. We worked on two main versions of RTEMS: normal version and squeezed version. The normal version places only the start code (.start section) that is responsible for low-level initialization and interrupt hooks, in local memory and all other code sections are placed on external memory including the stack, heap, .text and .data sections. The squeezed version aggressively reduces the whole RTEMS size to fit into less than 32KB local memory by means of compiler's optimization flags and source code hacking. As you may have guessed, the squeezed version is much more faster than the normal one.

As you can see in figure3.4, the squeezed version that places the whole RTEMS in a local memory is extremely fast compared to other versions.

We note that since `Clock_isr` (and its call tree) is a large piece of kernel code, execution time rises significantly when not in local memory (see figure3.4) – hence it appears a good trade-off to place `Clock_isr` and `ISR_handler` in local memory, as they are executed frequently every 10ms to handle the default clock interrupt in RTEMS. As shown in figure3.4 placing the interrupt stack in local memory also improves execution time.

Table2 in figure3.4 shows context switch and dispatch timings. `_Thread_Dispatch` is called from `ISR_Handler` if a new higher priority task has been scheduled (eg. potentially called from `Clock_isr` for periodic thread release), and eventually calls `_Context_Switch`. Note that timings for `_Context_Switch` and `_Thread_Dispatch` are marked as N/A in the table, as they are essentially the same as nothing local (unless the whole of RTEMS is in local memory) – this is due to

[2]Note that Adapteva are considering extending local memory to 64Kbytes for future eCore and Epiphany architectures.

`_Thread_Dispatch` calling a lot of other routines which are in external memory.

Timings for SMP features, including message passing and inter-CPU interrupts remain for future work. Note that current observations suggest inter-core interrupt latency is almost the same as with `ISR_Handler` plus C user SMP handler.

3.4 Conclusions

Many core chips have become popular during the last decade, trending towards massive multi-core / NoC architectures [103]. To respond, RTOSs have to be re-designed to scale alongside the architecture scaling, noting that synchronization between CPUs becoming increasingly important.

This chapter has considered the porting of a conventional monolithic RTOS design, namely RTEMS, to the Epiphany NoC architecture. This architecture has only limited local memory, hence the issue of how much of the RTOS can be placed into local memory, rather than external memory, is important. It is noted that the relative costs of accessing external memory to local memory suggest that getting the allocation of RTOS to local and external memory is crucial to overall system performance. Further investigation of this balance is required, with only basic consideration (of key functionality) given in the chapter. One key conclusion though of this chapter is that placing the whole of RTEMS into local memory is not practical, for 32 KiB local memories.

Further work will investigate the true costs of SMP communication (via shared memory and inter-CPU interrupts). We noted that initial observations suggested that RTEMS SMP is efficient, although this needs to be quantified fully.

To conclude the experience of porting RTEMS to Epiphany and related RTEMS design decisions, the issue points mentioned in section 2.4 are discussed here.

First, the most important observation gained from this experience is the memory structure within a multi-core chip. OSs have a lot of data structures, and they need to be laid out and accessed efficiently in a scalable way. This is the responsibility of both kernel developers and hardware designers. The Epiphany architecture has two extreme types of memories: fast small local memories, and slow big external memory. Clearly with a considerable effort only on the kernel structure part, it

is not enough to get the best use of such architecture mainly because of its memory structure. The hardware has to get involved and amended to help achieving an overall optimal multi-core system. The good aspect about memory system in Epiphany (according to RTEMS), is that there is no MMU unit. Thus, the port does not have to do any static initialisation of MMU, or suffer from translation, TLB or fragmentation overheads. However, a slightly more sophisticated OS like seL4 would not simply work there, because of its needs to memory protection, privilege levels and more memory footprint. The lack of cache units was also a big win from RTEMS perspective to maintain its predictability requirements and simple code base. Not to mention the scalability boost gained by getting rid of non-scalable caches.

Second, Epiphany is very good when it comes to its address space model. From the Epiphany chip, every core and IO device can be simply accessed using addresses. This greatly simplifies how different (heterogeneous) cores communicate with each other. Such a flat address space enabled the port to simulate a console driver using only shared memory.

There was a final attempt to get RTEMS working on only local memories (16 * 32 KiB) by re-designing it in a microkernel way with only one kernel instance on one core, and other device drivers and applications on other cores. Unfortunately this was not applicable for the following reasons: 1) Run-time-wise, RTEMS is inherently a monolithic kernel, there is no IPC layer like with microkernels between its different subsystems, 2) the result of the building process of RTEMS is a single image (even on SMP it assumes a shared memory address space) containing the kernel, drivers, and applications, so it was not even possible to split up its components during the building process, 3) Epiphany architecture does not provide any protection/isolation between cores and IO, so maintaining security on such platform is an open issue.

Chapter 4

seL4 microkernel on RISC-V Hardware Architecture

The objective of this chapter is to investigate the hardware requirements to run a more sophisticated OS (than RTEMS) like seL4 microkernel, which needs to enforce security, simplicity and scalability as its design principles. seL4 requires more memory footprint, hardware protection features and IPC implementation layer, thus Epiphany was not a suitable hardware platform candidate. RISC-V ISA is a new open-source hardware specification, which provides the previous hardware requirements to run seL4 microkernel. Moreover, RISC-V specification stands out between other hardware architectures for being completely open-source and defining how an implementation may add new hardware features in a standard way according to its requirements. This allows implementing new research ideas on both OS and hardware architectures, not being restricted by one another.

Because of its simplicity and minimality, seL4 (as a microkernel) can act as a hypervisor that can be quickly ported to new (research) hardware architectures, while application layer can still be (relatively) hardware agnostic and compatible. Application layer may involve a complete monolithic server (like L4 Linux) with a very little effort compared to porting Linux itself to a new (perhaps exotic) hardware architecture. This way, new hardware features can be assessed (i.e new hardware security features or memory management mechanisms) without having to re-factor thousands (or even millions) lines of code of affected OS components (like memory management) while preserving application-level compatibility, and getting the advantage of reusing a lot of other conventional/legacy applications including monolithic kernels and their rich

device drivers repositories.

4.1 Introduction

In order to investigate new solutions in both hardware and software for the previous issues, seL4 microkernel has been ported to RISC-V hardware architecture. Both are completely open-source, allowing us to modify and add new features flexibly. In this chapter, the RISC-V architecture is introduced as well as the porting process of seL4 microkernel to it. The focus is on assessing seL4 (as a microkernel design) and RISC-V (as a new open-source ISA) features and how they fit together. No performance analysis is done for some reasons: 1) RISC-V is an ISA specification rather than an implementation, the chapter is mostly concerned with the ISA only, 2) because RISC-V is open-source, different implementations can trade-off and add their own features according to their requirements (power efficiency, size, performance, scalability, etc), 3) because RISC-V is new, few (mature) open-source implementations actually exist currently, the most reliable ones are Spike simulator and FPGA-based Rocket Core which were used while porting seL4 and are not quite feasible to get performance analysis for.

The reason for choosing seL4 is that it is currently considered the state-of-art microkernel, that is simple, powerful and formally verified (has no bugs). Moreover, being a microkernel, it has to deal with memory management (MMU), and its subsystems communicate via message passing channels. This allow us to target broader challenges of memory management and protection as well as scalability, and such issues could not be addressed within RTEMS that only works in privilege (kernel) mode with no notion of user/kernel levels, and hence all service invocations are just direct function calls with passing parameters and sharing memory rather than message-passing-oriented. Indeed, this is a current bottleneck for RTEMS SMP support. Furthermore, RTEMS barely provides (dynamic) memory protection and isolation between threads. So, seL4 acts as an optimal OS for NoC-based SoC (given that it is inherently using hardware message passing), allowing us to address the memory protection challenges within NoC/SoC.

RISC-V [101] is a new open-source Instruction Set Architecture (ISA) designed by UC Berkeley. It is mainly introduced for educational and

research purposes, and to avoid the patent issues involved with commercial processors like Intel, ARM and MIPS. Giving that RISC-V is just an architecture specification, it allows any one to have their own hardware (microarchitecture) implementation, but they all must comply to the original ISA specification. That is, an implementation can be concerned with power consumption, code density, scalability, area or any other trade-offs.

Currently there are some RISC-V cores implemented by different parties, some of them are open-source and others are not. In the following sections, RISC-V user and privilege level specifications are introduced.

4.2 RISC-V User-Level ISA - Version 2.0 [101]

RISC-V ISA is divided into base integer and extension ISA. The base integer is mandatory for any RISC-V implementation, and it acts as the core of the ISA. There are two versions of the RISC-V base integer ISA: 32 and 64 bit, with the feasibility of future 128 bit ISA. These are called RV32I, RV64I and RV128I versions. An implementation can support RV32I or RV64I or both of them.

Beyond the base integer ISA, there could be standard, but optional, extensions like multiplication and floating point units. There is a separate Privilege-mode ISA specification for OSs discussed later. The user-level ISA was basically designed to be modular and simple, and not to rely on any customized hardware features like caches, in/out-of-order execution, thus, allowing an implementation to take place with its own extensions if required.

The base ISA instruction encoding is fixed 32-bit length, with the possibility of adding variable length instructions. There is also a 16-bit compressed ISA but it is optional. In the following section, 32-bit ISA is described.

4.2.1 32-bit RISC-V Base Integer ISA

RV32I ISA is a load-store architecture that has 32 registers (x0-x31). It is little-endian and assumes byte-addressable memory. Instruction encodings are divided up into categories as seen in figure 4.1

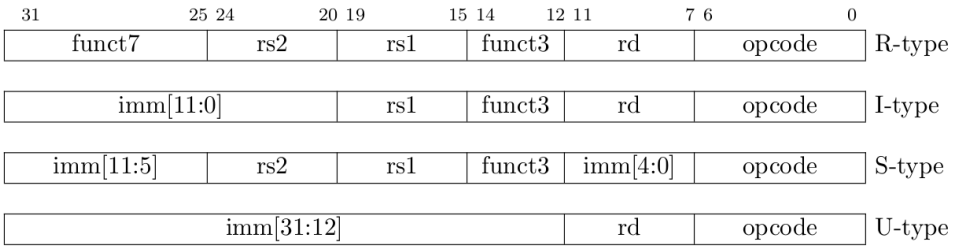


Figure 4.1: RISC-V RV32I Instruction Encoding [101]

R-type encodes instructions that operate on registers and store the result in a register also. Registers' addresses are fixed across different instruction encodings. Examples of R-type instructions are: add, and, or, xor, all with three operands (rs1, rs2 and rd).

I-type encodes instructions that has one of its source operands in a register (rs1) and the other is an immediate encoded part of the instruction taking a room of 12 bits. To ease sign extension decoding, MSB of any immediate encoding is always placed in instruction[31] bit. Examples of I-type instructions are: addi, andi, xori, and also has three operands with rs2 replaced with an immediate value.

S-type instructions are usually used for encoding conditional branch instructions with rs1 and rs2 source operands to be compared and 12-bit immediate as the branch signed offset giving a range of 4 KiB from the current pc in either direction.

Finally, U-type instruction encodes unconditional jumps with 20-bit signed offset from pc with 1 MiB range.

Besides the previous instructions, there is a *fence* instruction to allow the programmer to manage concurrency within a relaxed memory model. Concurrency can take place between concurrent RISC-V threads or IO devices for example.

4.2.2 System Instructions

There are system instructions used to change the RISC-V mode from user-level to privilege-level to request a service from an OS for example. ECALL and EBREAK are used for trapping and debugging purposes. Furthermore, to help with scheduling and profiling, there are system instructions to handle cycle and time counters.

4.3 RISC-V Privilege-level ISA

RISC-V Privilege-level ISA involves every state and operation that are beyond User-Level ISA and typically provides the features for an OS to execute. At the time of writing this thesis, the RISC-V privilege-level specification is in a draft [100] form that will most likely change in the future. The components that are involved part of seL4 port are discussed here.

4.3.1 RISC-V Privilege Modes

There are currently four RISC-V privilege modes: user, supervisor, hypervisor and machine modes. Machine mode is the highest privilege level, giving access to all hardware resources, and it is mandatory and must be implemented in any RISC-V compliant core, while the others are optional. To provide applications with protection, a minimal RISC-V core would support both user and machine modes. Supervisor mode provides page-based protection as most OSs expect. Hypervisor mode helps with virtualisation, however it is not documented nor it has been implemented yet. In the following section machine and supervisor modes are discussed in more details.

RISC-V Machine Mode

Machine mode is the highest RISC-V privilege mode and is the first mode to be entered on power reset. An application executing in machine mode can access any register state, including lower privilege registers. Machine mode registers are prefixed with *m* letter.

mcpuid register gives information about the features the underlying RISC-V provides like which ISA does it provide (RV32I, RV64I or any other extensions). It also specifies which privilege modes are there part of this hardware implementation.

Machine Status Register *mstatus* is the most important register in Machine mode that controls the RISC-V core and monitors its behaviour. It manipulates interrupts, privilege modes, virtualisation management and disabling/enabling other features like floating point operations.

The 5-bits VM entry within the mstatus register represents how virtualisation management is done during the current execution time. The different VM management modes are shown in figure 4.2

Value	Abbreviation	Modes Required	Description
0	Mbare	M	No translation or protection.
1	Mbb	M, U	Single base-and-bound.
2	Mbbid	M, U	Separate instruction and data base-and-bound.
3–7	*Reserved*		
8	Sv32	M, S, U	Page-based 32-bit virtual addressing.
9	Sv39	M, S, U	Page-based 39-bit virtual addressing.
10	Sv48	M, S, U	Page-based 48-bit virtual addressing.
11	Sv57	M, S, U	Reserved for page-based 57-bit virtual addressing.
12	Sv64	M, S, U	Reserved for page-based 64-bit virtual addressing.
13–31	*Reserved*		

Figure 4.2: Encoding of virtualization management field VM[4:0]. [100]

M, S and U refer to Machine, Supervisor and User modes respectively. Mbare mode in entered on reset with no protection or translation.

Different exceptions and interrupts can occur in machine mode and RISC-V recognizes the cause of the exception by reading the *mcause* register. Figure 4.3 shows different exception types.

RISC-V Supervisor Mode

Supervisor mode is the proper mode that contemporary Unix-based OSs and microkernels like seL4 expect. It provides page-based translation and protection by a memory management unit (MMU) that involves page tables and entries. There are few page-based modes each with its page (table) and address space granularity.

Sv32 provides a 32-bit address space covering 4 GiB, with both 4 MiB mega pages and 4 KiB pages granularity laid out in a one or two level page tables respectively. It works with RV32 ISA. Sv32 system is discussed here as seL4 port has been ported for it, and also other Sv39 and Sv48 systems follow the same terminology.

The formats of virtual/physical addresses as well as page table entries are shown in figures 4.4, 4.5 and 4.6. VPN is 20-bit virtual page number that is translated to 22-bit Physical Page Number (PPN). The page offset stays fixed and is not translated.

V bit refers to a valid Page Table Entry (PTE) when set. R and D bits specify whether the corresponding page has been read or modified (dirty), and these might be managed/used by the OS. Type bits encode the access control attributes of the page and whether it is a leaf PTE or a pointer to the next page table level. Possible PTE types are provided in figure 4.7

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Environment call from H-mode
0	11	Environment call from M-mode
0	$\geq$12	*Reserved*
1	0	Software interrupt
1	1	Timer interrupt
1	$\geq$2	*Reserved*

Figure 4.3: Exception Codes [100]

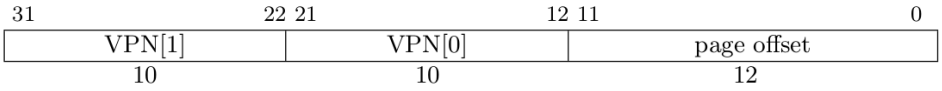


Figure 4.4: Sv32 Virtual Address Format [100]

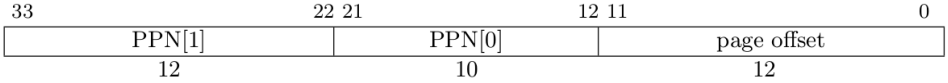


Figure 4.5: Sv32 Physical Address Format [100]

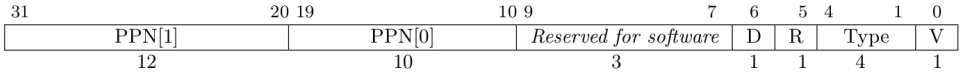


Figure 4.6: Sv32 Page Table Entry Format [100]

			Supervisor			User		
Type	Meaning	Global	R	W	X	R	W	X
0	Pointer to next level of page table.		—					
1	Pointer to next level of page table—global mapping.	•						
2	Supervisor read-only, user read-execute page.		•			•		•
3	Supervisor read-write, user read-write-execute page.		•	•		•	•	•
4	Supervisor and user read-only page.		•			•		
5	Supervisor and user read-write page.		•	•		•	•	
6	Supervisor and user read-execute page.		•		•	•		•
7	Supervisor and user read-write-execute page.		•	•	•	•	•	•
8	Supervisor read-only page.		•					
9	Supervisor read-write page.		•	•				
10	Supervisor read-execute page.		•		•			
11	Supervisor read-write-execute page.		•	•	•			
12	Supervisor read-only page—global mapping.	•	•					
13	Supervisor read-write page—global mapping.	•	•	•				
14	Supervisor read-execute page—global mapping.	•	•		•			
15	Supervisor read-write-execute page—global mapping.	•	•	•	•			

Figure 4.7: Sv32 Page Table Entry Type Encodings [100]

Figure 4.8 is a flowchart demonstrating how page translation and protection is done in Sv32 mode. *sptbr* register holds the address of the first-level page table (as known as page directory).

Similarly Sv39 and Sv48 work with RV64 ISA, providing 39-bit and 48-bit virtual address space respectively, with three level page tables.

4.4 seL4 on RISC-V

Porting seL4 to RISC-V requires a knowledge of both seL4 microkernel and RISC-V design/implementation. The project succeeded to perform a complete port of seL4 microkernel that enables running a simple OS on top of it. Currently it can run on Sv32/RV32, Sv39/RV64 on both Spike (the main RISC-V simulator), and also on Rocket Chip (FPGA). There have been some implementation trade-offs regarding the project, described below.

4.4.1 seL4/RISC-V Port Details and Trade-offs

32-bit or 64-bit? Both! seL4 only supports 32-bit targets currently, on the other hand, RISC-V has been focusing on 64-bit implementations right from the start with a little support for 32-bit; UC Berkeley team has only 64-bit Rocket chip and there is no 32-bit hardware implementation so far (except for some simple educational repos). Luckily, Spike has recently supported 32-bit mode (with a new –isa flag). It is

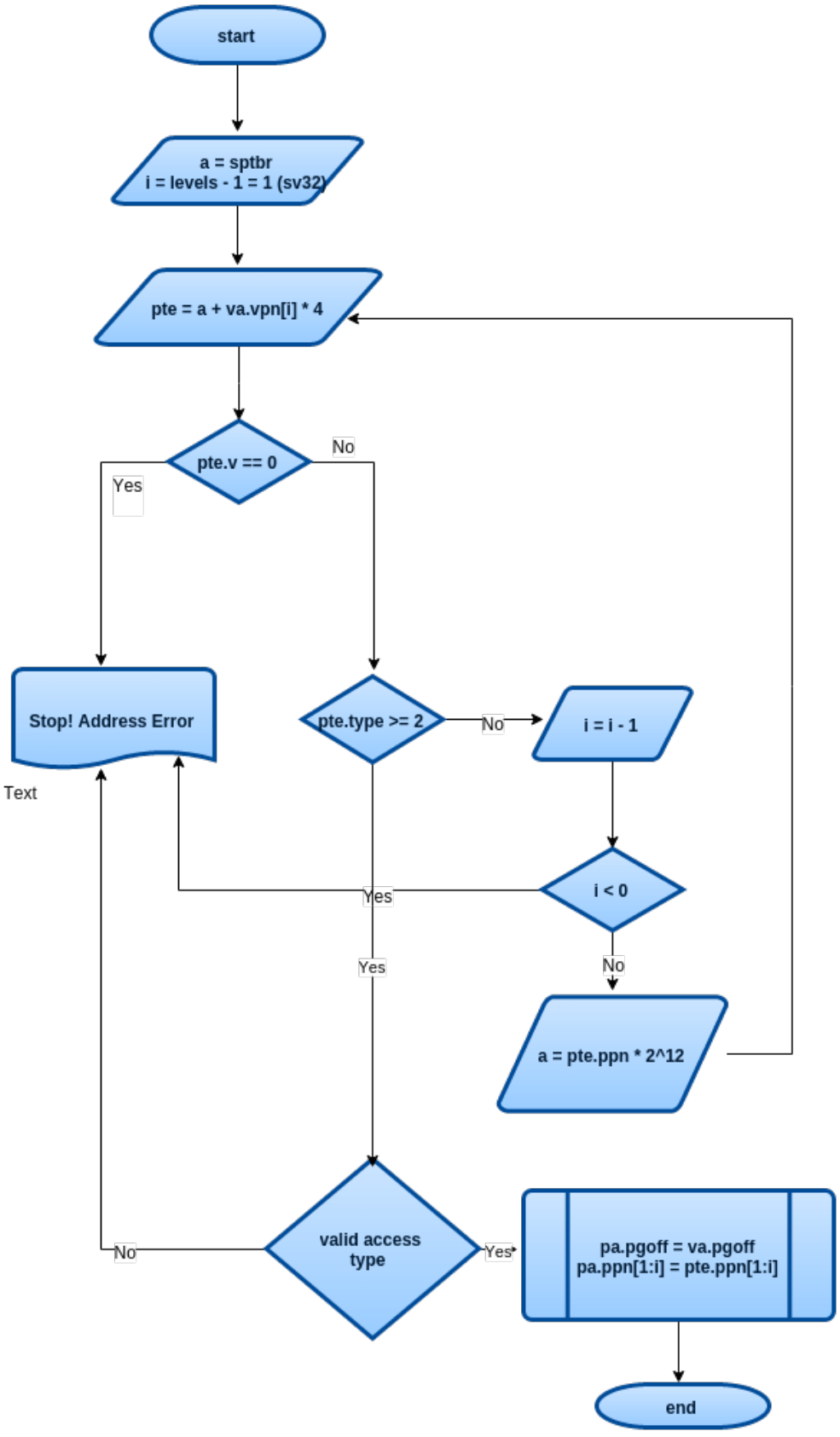


Figure 4.8: Sv32 Page Translation Process.

easier to port seL4 for 32-bit architecture trying to follow/imitate the existing ARM/IA-32 ports. A complete 64-bit implementation would be more challenging as most of the seL4 data structures and scripts assume 32-bit environment. The port successfully runs on Spike/RV32 simulator. In order enable it running on real hardware (64-bit Rocket Chip), the page-tables and memory management related data structures were laid out to comply with Sv39/RV64 system. All other instructions and data structures are kept in 32-bit formats as seL4 was originally designed for.

Working in which mode?

The latest privileged specification introduces 4 modes that RISC-V software can run in. Conceptually, seL4 might run in any of the three privileged modes separately, or even two or three of them simultaneously. Figure 4.9 shows the possible seL4, guest OSs and applications configurations regarding to which modes they can run at.

The number of which modes to run seL4 microkernel in was narrowed down to two by the fact that there is no hypervisor implementation yet. These two modes are: machine (M-mode), and supervisor (S-mode) modes. The M-mode supports physical access control and Base-and-Bounds checking, i.e. no mapping or address translation, only S-mode does. seL4 microkernel on the other hand expects that it would run in an address-translation-based mode, and would map its kernel image, IPC buffers, bootframe and other areas of memory during bootstrap. So we followed the current seL4 ports to work in S-Mode.

Loading the image(s) and mapping pages

The bare seL4 system basically consists of: 1) the kernel image, 2) applications. Current ARM and IA-32 seL4 ports differ in the way they load the kernel image and applications. Since IA-32 port can boot in multikernel mode, it loads the images in way similar to grub. So the kernel is the first part that takes control of the physical resources, and it loads/maps the application images itself. The ARM port behaves differently in that it archives the kernel and application images in cpio format. There is a separate elfloader tool that reads the ELF images from the cpio archive, loads it to the available physically-adjacent memory, sets up

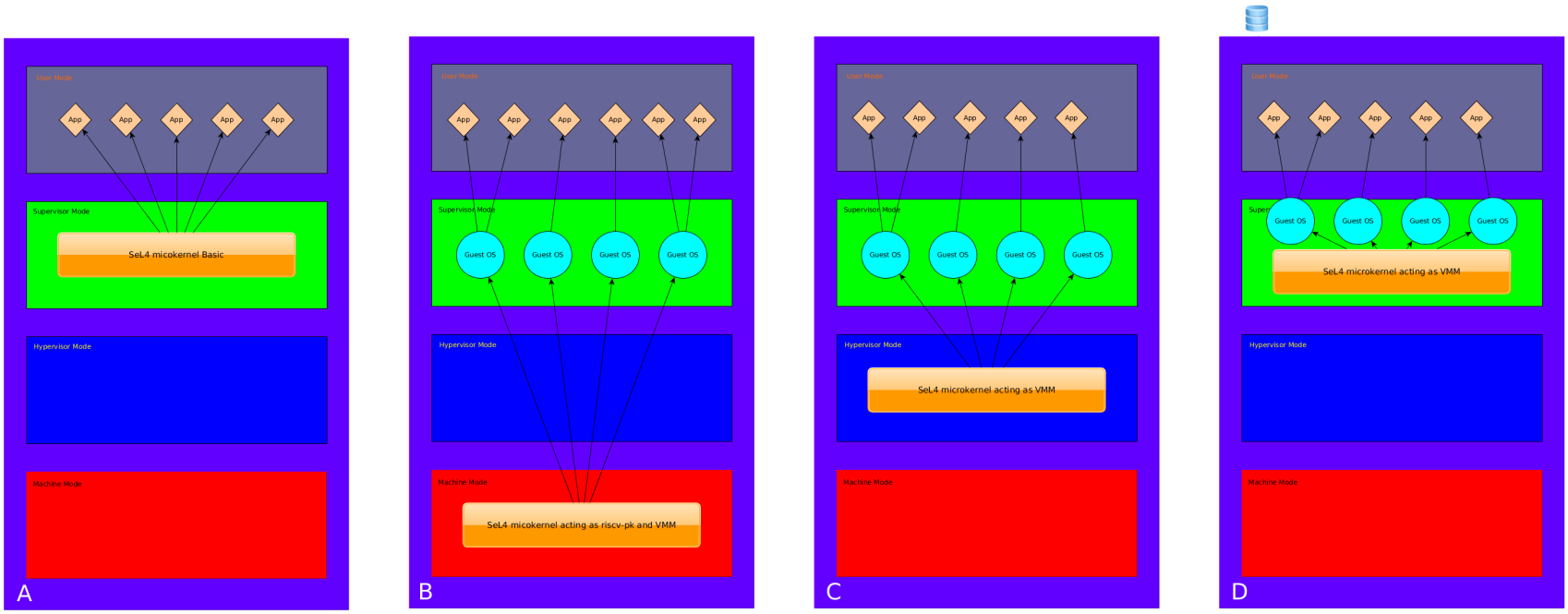


Figure 4.9: Possible RISC-V Modes That seL4 Can Run In.

the VM environment and finally maps the ELF images according to their ELF's section VMAs. Hence, the elfloader is the first to take control of the physical resources, and then it passes control to the kernel (which works in a VM environment right from the start) with some information passed to it about the loading addresses of the kernel image itself and the user image(s). The final image for the seL4/ARM system then contains: 1) elfloader tool, 2) libelf, 3) libcpio, 4) kernel image and 5) user applications. We followed the ARM port as it is more hardware agnostic, and as a start the RISC-V wouldn't need to support multikernel mode.

Other seL4 components involved

seL4 microkernel itself needs other tools and libraries to work with. Such tools are elfloader, building/configuration systems and other user-space utility libraries. The port involved working on:

libmuslc: libelf depends on libmuslc. I performed a very basic port of musl c library to RISC-V architecture, enough to build it successfully and produce the .a library.

libelf: This one is portable and architecture-independent. It has to be included part of the elf loading process.

libcpio: like libelf, libcpio is also architecture-independent and is used to read the cpio archive containing the kernel image and user images.

elfloader: This tool is developed by seL4 team for the ARM port, I had to port it to RISC-V. It has to work in M-mode and it is acting as riscv-pk, that is, any system calls from seL4 microkernel are redirected to elfloader code, which handles it and returns (apart from its main purpose which is loading the kernel/user images). elfloader currently only supports write and exit system calls (to be able to get some printf output and exit the spike simulator).

seL4 microkernel: The project is mainly about the seL4 microkernel. The port basically followed ARM port and even a lot of code is copied from it. seL4 microkernel runs in S-mode right from the start as mentioned previously. Some architecture-level capability data structures had to be modified according to the RISC-V ISA, and the low-level RISC-V VM handling code is now implemented to map the kernel image, kernel frames, initial task and user images properly.

Build system: The build system for seL4 projects is the Linux Kconfig/Kbuild build system. The existing Kconfig/Kbuild files had to be modified to allocate a new entry for RISC-V architecture with

a new Spike platform. New riscv defconfig, project-riscv.mk, makefiles and other files were added to enable building a complete seL4/RISC-V system (elfloader, libcpio, libelf, seL4 mircokernel and user image) like in seL4tests project and other seL4 projects.

4.4.2 Simple Operating System Running on seL4

seL4 microkernel runs in privilege mode while device drivers and applications run in user-mode. To prove the port is reliable, and to put new message passing and scalability issues under consideration, an interesting use-case such as running an OS on top of seL4 would be useful. This also might allow heterogeneous OSs to run on heterogeneous cores, with seL4 acting as a hypervisor, and this is an active area of both research and industrial development currently.

What is SOS

Simple Operating System (SOS) is a server running on top of the seL4 microkernel. The SOS server is expected to provide a specified system call interface to its clients (Specified in libs/libsos/include/sos.h). SOS is used part of an advanced OS course offered by University of New South Wales and currently only runs on Sabre Lite ARM-based board, seL4 on RISC-V can be considered the second supported platform for SOS, and the first all-open-source seL4 system, providing that seL4 (and its components), RISC-V ISA, Spike simulator (and Rocket-Chip on FPGA) are all open-source. The SOS framework is described in figure 4.10.

The components shown in the figure are:

Harware: The hardware described in our case is the RISC-V platform.

seL4 microkernel: This is the seL4 RISC-V port of the kernel (and the third port after IA-32 and ARM). It provides the functionalities needed to run our SOS project in the form of memory management, scheduling, IPC, etc.

SOS: a stub OS running on top of seL4 microkernel. It is intended to be developed and enhanced by students and/or people who are interested to learn about seL4. SOS initializes a synchronous endpoint capability for its clients/applications to use for communication. Interrupts are delivered using an asynchronous endpoint (seL4 has two types of endpoint capability: synchronous and asynchronous).

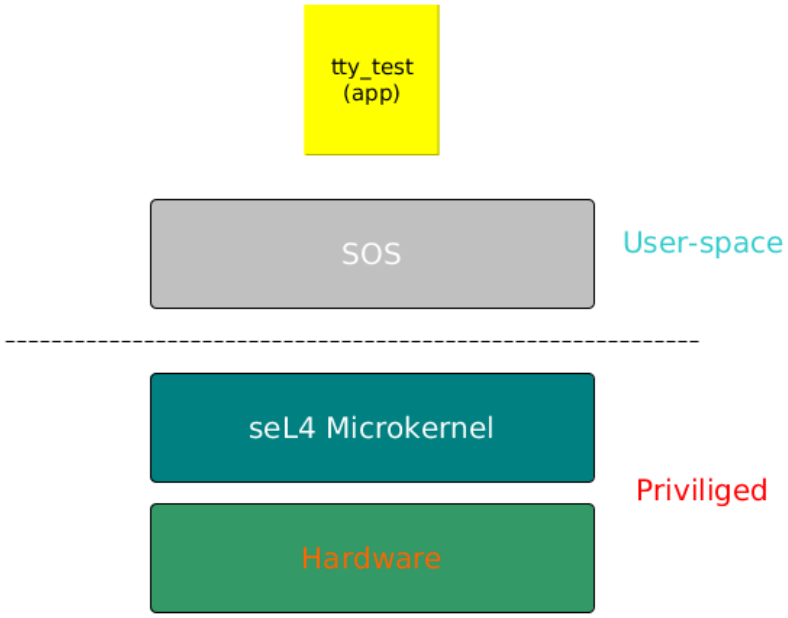


Figure 4.10: SOS Framework

tty test: It serves as a simple application running on top of SOS that simply prints out a hello word message. The application level would need to issue system calls to SOS using the seL4 endpoint capabilities. An example of such a system call is tty test application requesting (from SOS) some data to be printed out. SOS on the other hand monitors the system call requests from its clients (using seL4Wait system call), serves it, and sends replies.

What is needed to support running SOS

Other than the seL4 microkernel internals, almost all of the current seL4 user-level libraries had to be supported to build SOS and its applications. To be able to build/run SOS, the following components are involved:

seL4 microkernel: it now supports memory management capabilities, context switch, traps from user applications, and a lot more architecture-dependent functions were implemented.

libseL4: This is the user-level library for applications to deal with seL4 microkernel via system calls. It defines the format of the system calls, kernel objects definitions, user-level context and it exposes them all to the user.

libmuslc: The C library that seL4 and its libraries depend on. It has been ported to RISC-V part of this project, and now it is working pretty fine as expected.

libsel4muslcsys: A minimal muslc implementation for the root task to bootstrap, it provides stdio related system call handlers and it is part of the bootstrap procedure of the root task, defining the system call table and entry point for muslc-based applications.

libplatsupport: Some platform related functions (BSP) for seL4 supported platforms. For example serial driver initialization and console driver functions for a given board are provided there. libsel4platsupport depends on it. I had to add Spike platform with very basic implementation just to get over build dependencies.

libsel4platsupport: For RISC-V it has to be ported to provide the bootstrapping and the exe entry point `__sel4_start` for the root task. It gets the boot frame address from the seL4 microkernel, constructs the stack vector as muslc expects, and then jumps to the normal muslc `_start` entry, enabling it to populate the libc environment's data structures with its details, initializes TLS, files and stdio handlers, etc. Finally the muslc task bootstrap procedure jumps to the user's main() function, or the root task, which in our use case is SOS.

libcpio: used by SOS to parse the cpio archive, searching for user binaries.

libcpio: This one is used by SOS to parse the ELF binaries extracted from the cpio archive. Hence SOS can read the ELF's section headers, and do the loading/mapping consequently.

libsel4cpace: a library provided to abstract away the details of seL4 CSpace management, this library had to also be ported for RISC-V. It is used by SOS to construct tasks' CSpace.

mapping: SOS comes with mapping.c file that is needed in conjunction with elf.c to load/map the user ELF binaries. It is ported to RISC-V and it invokes the newly provided RISC-V system calls like `seL4_RISCV_Page_Map` and `seL4_RISCV_PageTable_Map`.

Other libraries (like libmuslcsys and libsel4vka) had to be modified to be aware of the new RISC-V architecture and just modified to be built, again to get over other required libraries dependency.

Figure 4.11 shows the run-time steps for a complete SOS/seL4 to execute until it reaches the highest level application of the stack (tty application).

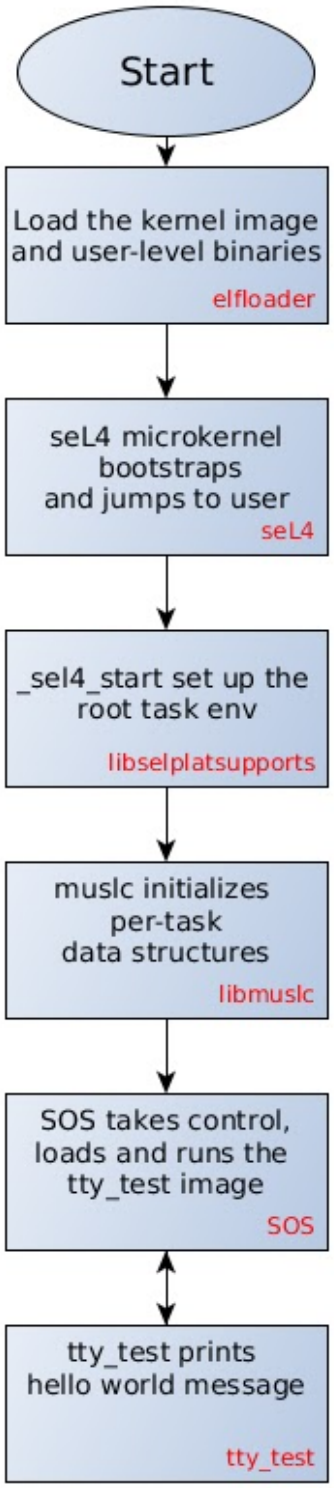


Figure 4.11: seL4/SOS bootstrap procedure

4.5 Conclusion

seL4 microkernel is more sophisticated than RTEMS. It relies on message-passing to communicate between its different subsystems, which makes it scalable, and a better fit to run on NoC-based SoC. A complete seL4 project would involve seL4 microkernel and many other tools and libraries (there were over 20 repositories that were involved to run SOS on seL4). This makes it far bigger than an RTEMS systems. However, thanks to its modular design and message passing mechanism that microkernels in general agree on, it is possible to scatter seL4-based system across distributed cores that can be in a form of memories, heterogeneous cores, accelerators, etc. That said, the small seL4 microkernel can run on one or more (RISC-V) cores, and its libraries and applications might run on other, possibly heterogeneous, cores that perform better on, and all of these subsystems communicating with each other via a standard (message-passing) interface. This way, it is no longer needed to constrain a system to run a specific OS (that might not perform well with some application requirements) and to run it on a homogeneous SMP (single ABI/architecture) like with RTEMS on Epiphany NoC. We now have the freedom to modify both software (seL4 as a hypervisor and its applications that can be other OSs) and hardware (RISC-V based SoC with other pluggable heterogeneous cores and NoCs).

Now to link with issues mentioned in section 2.4. Regarding the memory system, it is a very conventional one with MMU and 4 KiB pages for both 32 and 64 bit address spaces. This makes seL4 (and Linux) happy enough to run there, but does not give an advantage over any other architecture, instead it inherits the page-based issues discussed in the section 2.4. Perhaps this can be enhanced in the future due the flexibility of RISC-V architecture. Cache coherency is irrelevant since it is not yet part of RISC-V standard, but an implementation can choose whether to implement it or not. The good thing is that RISC-V architecture can be extended to support other new features (given that it is mainly a research open-source architecture) and memory models as required.

The most irritating thing about porting seL4 to RISC-V (Rocket Core and Spike), is its Host Target Interface (HTIF) interface. It is a totally new interface, that needs programming and special handling to construct and send packets with format a host device would understand. This needs support from both a target OS (to construct commands and data

packets and send them, which requires M-mode instructions), and the host device (to receive packets, interpret them and handle commands). Luckily, HTIF is a good example of how a new core (and IO device) interface should not be implemented. Moreover, RISC-V does debug/IO access using special purpose CSRxx instructions, which is not portable, neither easy to adopt, given that there is no current documentation for this interface.

Finally, RISC-V does not provide (enough) specifications neither seriously consider the multi-core heterogeneous issues and how cores communicate, at least not yet. Also, there is no implementation or specification details of the RISC-V hypervisor mode, which could be of a big benefit for seL4 given that microkernels have been successfully used as secured software hypervisors.

Chapter 5

Operating System Support on Multi-core Architecture

5.1 Discussion

This chapter concludes the research, programming and development efforts for this thesis, and according to that, it provides the requirements for both OSs and (multi-core) hardware architectures, and finally it suggests solutions for such requirements.

The overall development effort includes:

1. RTEMS port to Epiphany, MicroBlaze, OpenRISC and RISC-V architectures both SMP and shared memory multiprocessing.
2. seL4 microkernel Port to RISC-V architecture.
3. RISC-V Vscale core integration to OpenRISC wishbone-based FuseSoC.

As noticed in the previous chapters, the trend is moving to heterogeneous multi-core architectures, but each vendor (or organization) design their own customized hardware that communicates in a different way than one another, whether from software or hardware perspectives. Moreover, vendors most likely provide their own toolchain, and their own device drivers sometimes enforcing license issues to them.

This makes it harder for OSs to be ported to such new architectures. An example of this is an attempt to merge the work of RTEMS (open-source) port to MicroBlaze (by Xilinx), and the main issue that

prohibited this merge is the commercial licence that Xilinx enforces to its software code.

The opposite case is clear with RISC-V, which is all open-source, and although it is a new architecture, there have been many people working on OS ports to RISC-V. Currently there are: Linux, FreeBSD, seL4 and RTEMS ports. seL4 and RTEMS ports are part of the effort of this thesis.

This thesis suggests (from OS point of view) that (heterogeneous) multi-core hardware architectures should be:

1. Easy to program.
2. Portable (takes less effort to add OS support for it).
3. Scalable.
4. Communication-transparent.
5. Flexible.
6. Secure.
7. Easy to identify/address.

Easy to program The problem with some heterogeneous architectures is that they provide their own special purpose instructions, registers or interfaces to handle inter-core communications. This does require documentation from the architecture vendors and software developers are required to understand and grasp how to write low-level code (at the assembly level) that manages the cores. Moreover, it requires support from the toolchain mainly the linker and compiler (in the case of GNU, this is binutils and GCC).

Examples of this problem are the MicroBlaze's Fast Simplex Link (FSL) and UC Berkeley's host-target interface (HTIF) for their RISC-V SoC implementation. FSL core comes with documentation, but still the programmer has to read and understand how it works from MicroBlaze and other connected cores, as well as the programming interface. The problem with RISC-V is even worse. It does not provide any documentation, except for some code examples. And from such examples, the programmers have to know how to construct a packet to be sent to the host using a system call via a usage of some special purpose

registers like (mtohost, mfromhost), and magic memory that contains the data packet(s). Clearly this was a downside for people who wanted to use the Rocket Core (the UC Berkeley implementation of RISC-V), with HTIF. This made other projects like lowRISC to work on removing such target-dependent interface. Similarly, I worked on integrating the VScale/RISC-V core to OpenRISC-based projects, removing any use of HTIF interface and only relying on wishbone interface. This actually enabled a RISC-V to make use of many other open-source cores that are wishbone compliant, and supported by other boards.

Portable (takes less effort to add OS support for it) This is tightly correlated to the previous requirement. If there is a standard way for cores to address each other, this will save programmers a lot of time reading the documentation and programming the hardware. In fact portability can be as easy as just copying C code for one driver that works with some OS on a given CPU, and pasting it to a totally different OS that executes on another CPU.

An example of this is the NS16550 UART interface. The exact same driver for one RTEMS BSP can be used for another RTEMS BSP with another CPU. In fact, when I was implementing the UART driver for seL4 port to RISC-V/QEMU, I just copied the source code of the driver from RTEMS/OpenRISC port to seL4. This is certainly not the case with HTIF interface, or similar SoCs that include MicroBlaze. Both need a considerable effort, logic, and code size to implement a console driver.

Scalable Scalability issues with mutli-core systems have different parameters and aspects from hardware and software sides. Shared data structures and shared physical memory are great bottlenecks with scalability. The shared data structure issue is somehow solved by factoring the OSs as discussed with FOS, or keeping each core's data structures local with no sharing and only message passing to preserve consistency as with Multikernel. While this is a good solution and improves scalability issue, the hardware implementation is still a problem. Hardware caches that are used with Barrelfish port in x86-based architecture are restricting the boost of the multikernel design.

Communication transparent Programmers do not have to worry about what communication fabric is used to connect different cores whether it is a network on chip, bus or direct links. However, it may sometimes be useful to actually control the communication medium as discussed in [78] where software is controlling network on chips, and with Raw microprocessor [93] where software controls the crossbar. Still, security and privilege level should be considered when it comes to OS management, but the good idea is to embrace exokernel and microkernel models by exporting such new hardware features to applications for reasons like that applications better know their needs, and it is better to keep the kernel small and simple (and verified like with seL4).

Flexible One of the major problems that OSs are restricted to currently (even the research ones) are the limitation of some hardware features that were originally developed for uniprocessor systems and evolved from there. A clear example of this is cache units. Caches are huge bottleneck for scalability especially when the number of cores increases. This is mainly due to the requirement of achieving consistency between cached versions on different cores of a shared data structure. Even multikernel design which embraces no sharing and locality of data structures is still being restricted to using caches even if they do not need it. Furthermore, Barrelfish is using cache lines for communication between different kernels on different nodes. Other examples of making use of caches are scalable locks that try to be placed on different lines of caches not to be contended over, replaced or false shared. Similarly, another restriction is relying on virtualisation and protection features (i.e. page-size, MMU, Intel's VTx) of a given hardware like with Quest-V on Intel.

The previous examples not only need considerable amount of understanding and implementation effort from the programmer, but also are simply non portable. A software implementation that assumes some very specific cache features like coherency protocol or cache line size will simply fail when it moves to another different architectures with minor differences (i.e. cache line size). OSs should not rely on these kinds of features at all, rather, it can provide access control only, and leave the policy to applications. From the hardware side, architecture designers should not force programmers to use some features like caches, MMU, etc, at least to provide the option to simply disable them.

Secure Security is of major concern for OSs. From exokernel to monolithic, all agree that the kernel at least should guarantee some form of security between applications with each other and applications and the kernel. Providing such security/protection scheme can be supported by software-only libraries and mechanisms, or hardware enforcement. Most current hardware architectures do provide help for software to enforce protection. This can be a form of different privilege levels for each software type (kernel, application, hypervisors, etc), MMU, and/or even some new hardware capability implementation like CHERI [102]. Security and protection can even be of physical separation like what FOS suggests that each core holds its own kernel/application, without messing with other cores. Capability software for access control works great like discussed earlier with seL4; it works to the level where seL4 has been totally formally verified being the first world's OS kernel proven to have no bugs. While software-only capability model works well, it can work better with hardware-capability implementation like CHERI, improving performance, hardware security enforcement, and formal verification.

Currently memory management units are mostly providing page-based translation/protection. This might not be a good idea wasting memory for applications that want smaller protection granularity. Also, an RTOS like RTEMS with small memory footprints and performance requirements would not work well with 4 KiB page-based protection. In fact, most RTEMS targets just abandon using MMU because it can not afford the overhead of: n-level page-table translations, few TLBs entries with costly operations, cache (being non deterministic), and fragmentation caused when there is a need for protection over small amount of memory less than the smallest page granularity offered by the hardware.

CHERI provides some hardware solutions for the previous issues, but it has not been deployed by OSs that really need it like RTEMS and seL4, not yet.

Identification/addressing It is not a good idea that for each core in a heterogeneous system, there is a different way to identify and address it. A standard way for addressing other cores in the system is preferred, regardless of the type of the core whether it is CPU, FPGA, GPU, UART, etc. Thus, dealing with other cores as *black boxes* with shared standard interface would hugely ease the identification, addressing and communication processes that are managed by the kernel. Furthermore, the

same exact code can be used for addressing different cores by providing different parameters (i.e. core ID).

5.2 Proposed Solutions And Future Work

Using shared address space (not just for memory)

Shared address space is not just for memory. It can be used to address other cores including IO memory mapped devices. The downside of this is the waste of some address space bits. However, with 64-bit registers/addressing, especially with embedded systems, this would not be an issue. Both Tilera and Epiphany, as scalable multi-core chips, provide 32 and 64 bit address spaces respectively, with only the former having a cache unit and virtual addressing.

There are many advantages of using memory mapped cores (let's not call it devices any more). Rather than using special instructions and registers for controlling other cores, just use addresses. This is the case with Epiphany and FuseSoC (OpenRISC-based) currently, and porting RTEMS and seL4 to them was much more easier than RISC-V with HTIF for example. Thus, controlling other cores and communicating with it would not need any special instructions. A clear example of this is copying the UART driver of RTEMS on OpenRISC to seL4 on QEMU/RISC-V, without changing any code (except the base address). This greatly helps with easy-to-program, and portable issues.

Partitioned Global Address Space (PGAS) is a promising subset programming model of this idea. The RTEMS port on Epiphany implicitly utilised such a model but from OS side. As discussed in chapter 3, RTEMS made use of the local memories for some of its critical data/code sections like context switch and interrupt handling, while it placed other code and data structures on other remote memories (both remote fast memories of other core's local memories and external off-chip DRAM).

The Heterogeneous System Architecture (HSA) foundation proposes some standards for heterogeneous systems but their effort focuses on parallel computing (CPU and GPU). As with this thesis suggests, unedified address space is embraced. However, HSA still assumes virtual memory

management, page-tables, and cache coherency which may not be very efficient with scalability and real-time requirements.

Giving each core a memory mapped area part of the address space that has a base address, and bound is enough. This area frame can contain control registers, memories, interrupt control registers, buffers, etc. An overall target-related files can be then imported for both hardware and software systems, including base addresses for each core and its size. To use such cores it will be as easy as just including such a single header file with base addresses and use shared drivers, either existing part of the OS library, or provided by the core's vendor.

A case study for that might be converting RTEMS shared memory driver that is used by some targets to support multiprocessing (different RTEMS objects executing on different nodes/cores), into hardware implementation (memory mapped) for accessing another core (perhaps part of the communication medium like NoC or wishbone bus). Hence, the same exact code would not be changed, while the hardware implementation does. This way, the same code might work with shared memory, NoC, or buses.

Standard Interface (cores dealt with as black boxes)

From the OS side, the single address space and memory mapped cores can be considered a standard interface. From the hardware side, some standard interface might be a good idea to just plug-in new cores. This idea has proven its validity when I removed the HTIF interface of the VScale/RISC-V core and wrapped it up with the standard wishbone interface that FuseSoC/OpenRISC uses. It benefited from many other open source cores that can be easily plugged there.

The interface does not have to assume any core attributes, core designers have to just wrap their cores with such interface and it would be ready to be integrated part of other SoCs.

It then would not matter what the interconnection fabric is (from the core point of view), whether it is a bus, network on chip, or even a cloud. The interconnection medium itself would do the job of routing and serving the request. This way, a single core can be plugged to different network on chips, buses, or whatever, given that all of them comply with some interface standard like wishbone or AXI.

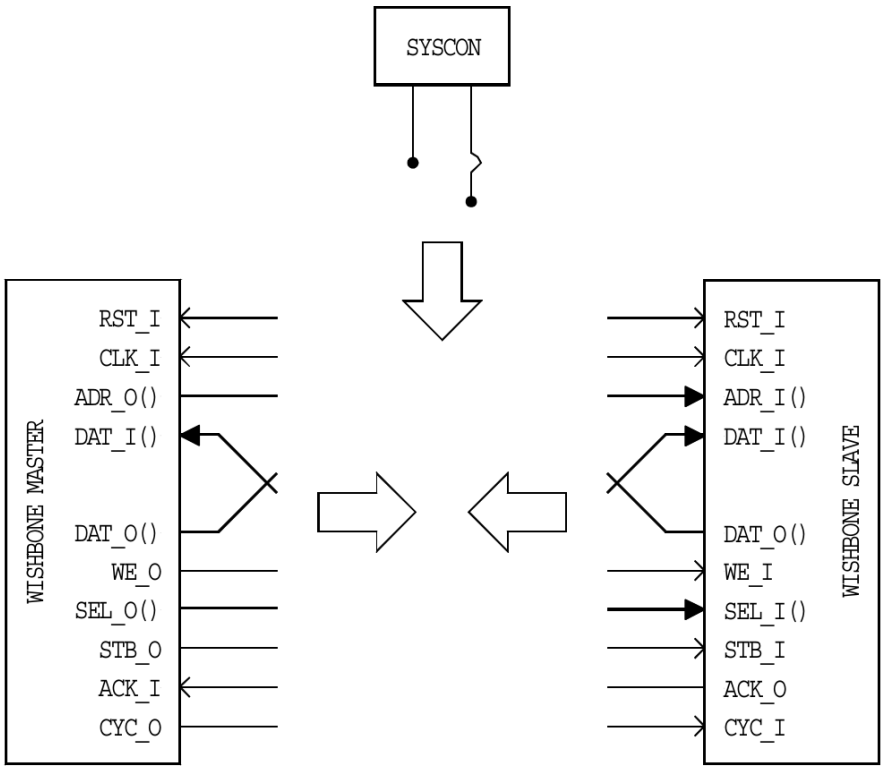


Figure 5.1: Wishbone Interface [57]

Capabilities for access control of heterogeneous cores

Capabilities for access control receieved attention again with microkernels like Fiasco.OC and seL4. It also helped with seL4 verification, proving that it has no bugs. Capabilities can be of any type. For example in seL4 there are frame, IPC, untyped memories and endpoints capabilities. For heterogeneous system, new capabilities types can be added if needed, giving an access control for one application running on one core to have access to another core (including control register, data, memory, etc). The capability type can be mapped to the core's type (CPU, UART, local memory, etc), or just using frame capabilities if such cores are memory mapped.

The above idea has actually been implemented above the seL4/RISC-V port, giving the root-task a frame capability by which it can manage booting/off-loading RTEMS (and even another seL4 kernel instance) to run on another RISC-V core.

CHERI on the same track, embracing capability model, is a hardware implementation of capability model based on MIPS processor. Both seL4 and CHERI would be a best fit, reducing the software code for managing

capabilities in seL4, improving performance, and enforcing hardware security by features provided by CHERI, along with byte granularity, and formal verification goals that both share.

Protection can be enforced then using MMU (like with seL4 frames), or either base and bound methods like with CHERI and machine mode on RISC-V.

Space and Power-aware Scheduling

In the near future, we might have a system where the number of cores would exceed the number of required processes; in this case there would not be a need for costly operations like context switching, MMU/page-table (or capabilities) that enforce security/virtualization, instead security would be enforced by physically separating cores. This model could be early noticed with RTEMS on Epiphany with 16 cores given that RTEMS is simple enough that it does not need MMU, and 16 tasks are more than enough for it.

Research then will focus on how these physically separated cores share data and/or physical resources as well as how they communicate with each other. As FOS predicts, the big issue at this time will be how to do *space scheduling* (rather than the conventional time-sharing scheduling on a single CPU) by dynamically assigning one or more (heterogeneous) cores to a process to satisfy its requirements, including location, distance, power management, communication medium and/or types of the cores. An example of this is to place seL4 microkernel on one core, and an application that handles matrix calculations (GPU device driver) on another core. Both have to be close to each other, and somehow the communication should be deterministic if it is a real-time application. Another example which has been already implemented as a prototype part of this thesis is running seL4 microkernel on a supervisor core, while off-loading RTEMS to another real-time friendly core.

The scheduler (whether it is implemented in user-mode, kernel-mode, or even on hardware) can be extended to handle further operations like load balancing, routing, and power management (voltage control or enabling/disabling cores for example). This will require the hardware to provide new structures for OSs (schedulers) to monitor and handle such operations. MMU protection would not make much more sense there, rather, capabilities can take place to enforce access control of which/how

application/core can access other cores or data structures.

Chapter 6

Conclusion

Computer architecture is already taking serious first steps to the new era of multi-core SoC, that would absolutely need both hardware and software cooperation. With this new architecture in mind, it would require some amount of compatibility support for software that assumes powerful features originally developed on uniprocessor systems. New challenges have been already triggered while trying to adopt such concepts, along with ensuring scalability of both the hardware and software, other new features may even be invented.

This thesis addresses the design and implementation issues of OSs on multi-core architectures, both theoretically and by attacking such issues by getting hands-on experience of porting different kernel designs to multi-core hardware architectures.

The literature review chapter lists the evolution of hardware architectures from uniprocessor to multi-core processors, along with other scalability related features like cache coherency and memory model. Then OS kernel designs and examples were discussed, and how they perform on different hardware architectures. Each OS design has advantages and disadvantages, so there has been a long-term trade-off between different kernel designs like microkernels and monolithic kernels. Some of them do well on given hardware architectures, but others do not. For example, microkernels are scalable on many-core chips, but has poor performance on one or few cores compared to monolithic kernels, while monolithic kernels overtake microkernels on a single or few cores. Security, real-time, virtualisation, scalability, performance, minimality and safety-critical requirements are other examples of such trade-offs between different kernel designs. Since the concern of this thesis is multi-core chips (and scalability), examples of OSs running on SMP and

multi-core chips were discussed next like RTEMS, seL4, Quest and FOS, and the trade-offs are discussed in more details there.

In order to to have an actual real-world experience of the OSs challenges on multi-core chips, chapter three reports the porting process of RTEMS (SMP) as a real-time OS to Epiphany multi-core chip. RTEMS SMP support is currently being developed, so this allowed us to do some analysis of the challenges and design decisions taken when adding support for originally uniprocessor OS, to run on SMP and multi-core hardware. This chapter discussed SMP related data structures involved within RTEMS that are target-independent as well as how the Epiphany and Parallella architecture features fit with RTEMS requirements. Such features are like addressing modes, communications between (heterogeneous) cores and most importantly the memory model. Three versions of RTEMS implementations were discussed, trading off kernel design, performance, size and features.

Because microkernels are superb when it comes to scalability, chapter 4 takes another L4 microkernel called seL4, which is considered the most secure state-of-art microkernel currently, as an experimental kernel design to run on a new research hardware architecture called RISC-V. The experience of porting seL4 to RISC-V gave us the insight of what are the basic requirements of seL4 as a secure microkernel that it needs from a hardware architecture like RISC-V. Such requirements are like: different privilege modes, page-based protection and high performance support from hardware to implement the most crucial component of microkernels–IPC.

Finally chapter five gives some general requirements of OSs and multi-core architectures followed by some proposed solutions that may do better than the existing discussed ones, citing some new research hardware implementation like capability-based hardware that might replace some hardware features like MMU, and do even better especially with the forthcoming many-core chips. Also chapter five credits the good (and bad) hardware design and implementation decisions experienced during porting RTEMS and seL4 to RISC-V/Spike, Epiphany/Parallella, MicroBlaze and OpenRISC/FuseSoC hardware architectures, including address space, addressing and communication between cores, simplicity of

programming and scalability issues.

Abbreviations

SMP Symmetric Multiprocessing

OS Operating System

SOS Simple Operating System

RTOS Rea-time Operating System

FOS Factored Operating System

RTEMS Real-Time Executive for Multiprocessor Systems

UMA Uniform Memory Access

NUMA Non-Uniform Memory Access

ccNUMA Cache Coherent Non-Uniform Memory Access

NoC Network on Chip

MPSoC Multiprocessor System on Chip

MMU Memory Management Unit

TLB Translation Lookaside Buffer

PTE Page Table Entry

GPU Graphics Processing Unit

CHERI Capability Hardware Enhanced RISC Instructions

DSM Distributed Shared Memory

IPC Inter Process Communication

UART Universal Asynchronous Receiver/Transmitter

IO Input Output Devices

ISA Instruction Set Architecture

libOS Library Operating System

ASID Address Space Identification

ABI Application Binary Interface

EDF Earliest Deadline First

BSP Board Support Package

GCC GNU Compiler Collection

TCB Task Control Block

DMA Direct Memory Access

IPI Inter Processor Interrupt

FP Floating Point

SDK Software Development Kit

HTIF Host Target Interface

Bibliography

[1] CoreLink System Memory Management Unit. http://www.arm.com/products/system-ip/controllers/system-mmu.php. Accessed: 2015-07-21.

[2] Desktop Boards - Compatibility with Intel® Virtualization Technology (Intel® VT). http://www.intel.com/support/motherboards/desktop/sb/CS-030922.htm. Accessed: 2015-07-21.

[3] lowRISC Project. http://www.lowrisc.org. Accessed: 2015-07-23.

[4] OPEN SOURCE RESEARCH PROCESSOR. http://parallel.princeton.edu/openpiton/. Accessed: 2015-07-23.

[5] PULP - an Open Parallel Ultra-Low-Power Processing-Platform. http://iis-projects.ee.ethz.ch/index.php/PULP. Accessed: 2015-07-23.

[6] Real-Time Linux Wiki. https://rt.wiki.kernel.org/index.php/Main_Page. Accessed: 2015-07-21.

[7] This is the third public release for the OS Abstraction Layer library from NASA/GSFC Code 582. http://opensource.gsfc.nasa.gov/projects/osal/OS_Abstraction_Layer_Release_Notes.txt, 2007.

[8] NASA GFSC selects OAR for RTEMS Engineering and Analysis again!!! http://www.rtems.com/node/34, 2014.

[9] RTEMS 4.10.99.0 On-Line Library. http://docs.rtems.org/doc-current/share/rtems/html/, 2014.

[10] Trygve Aaberge. Analyzing the performance of the epiphany processor. 2014.

[11] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[12] Adapteva. Epiphany architecture reference, 2015.

[13] Sarita V Adve and Mark D Hill. Weak ordering—a new definition. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 2–14. ACM, 1990.

[14] Alexandra Aguiar and Fabiano Hessel. Virtual hellfire hypervisor: Extending hellfire framework for embedded virtualization support. In *Quality Electronic Design (ISQED), 2011 12th International Symposium on*, pages 1–8. IEEE, 2011.

[15] James Anderson, Philip Holman, and Anand Srinivasan. Fair scheduling of real time tasks on multiprocessors. *Handbook of scheduling: Algorithms, Models and Performance analysis*, pages 31–1, 2004.

[16] James H Anderson and Anand Srinivasan. Early-release fair scheduling. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 35–43. IEEE, 2000.

[17] Thomas E Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6–16, 1990.

[18] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[19] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[20] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[21] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *rtss*, pages 119–128. IEEE, 2007.

[22] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[23] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[24] Michał Bejda. Efficient code placement management for epiphany architecture chips. Master's thesis, Jagiellonian University, 2003.

[25] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[26] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598. IEEE, 2008.

[27] Nikhil Bhatia. Performance evaluation of intel ept hardware assist. *VMware, Inc*, 2009.

[28] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[29] Gedare Bloom and Joel Sherrill. Scheduling and thread management with rtems. *ACM SIGBED Review*, 11(1):20–25, 2014.

[30] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[31] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *OSDI*, volume 10, pages 86–93, 2010.

[32] Bjorn B Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 292–302. IEEE, 2013.

[33] Björn B Brandenburg, John M Calandrino, and James H Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Real-Time Systems Symposium, 2008*, pages 157–169. IEEE, 2008.

[34] Alan Burns and Andy Wellings. Real-time systems and programming languages.

[35] Alan Burns and Andy J Wellings. A schedulability compatible multiprocessor resource sharing protocol–mrsp. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 282–291. IEEE, 2013.

[36] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. Litmus^ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 111–126. IEEE, 2006.

[37] Daniel Cederman, Daniel Hellström, Joel Sherrill, Gedare Bloom, Mathieu Patte, and Marco Zulianello. Rtems smp for leon3/leon4 multi-processor devices. *Data Systems In Aerospace*, 2014.

[38] Matthew Chapman and Gernot Heiser. vnuma: A virtual shared-memory multiprocessor. In *USENIX Annual Technical Conference*, 2009.

[39] Tilera Corporation. Tilera Has Solved The Multi-Processor Scalability Problem. `http://www.tilera.com/`, 2014.

[40] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were

afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48. ACM, 2013.

[41] Boston University Department of Computer Science. Quest. `http://www.questos.org`, 2015.

[42] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11, 2007.

[43] PF Dutot, G Mouni'e, and D Trystram. Operating systems internals and design principles. *5th International Edition, Prentice Hall*, pages 189–203, 2004.

[44] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150. ACM, 2013.

[45] Todd A Ely, Courtney Duncan, E Glenn Lightsey, and Andreas Mogensen. *Real time Mars approach navigation aided by the Mars Network*. Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2006.

[46] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.

[47] ETHz. Pulp - an open parallel ultra-low-power processing-platform, 2015.

[48] Jose Flich, Samuel Rodrigo, Jose Duato, T Sodring, Å G Solheim, Tor Skeie, and Olav Lysne. On the potential of noc virtualization for multicore chips. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 801–807. IEEE, 2008.

[49] S. Huber G. Bloom, J. Sherrill and C. Johns. *Structure of the RTEMS Real-Time Operating System*. CRC Press, Taylor & Francis, 2015.

[50] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. *Memory consistency*

and event ordering in scalable shared-memory multiprocessors, volume 18. ACM, 1990.

[51] Robert P Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112. ACM, 1973.

[52] Zonghua Gu and Qingling Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization. 2012.

[53] Gernot Heiser. Many-core chips—a case for virtual shared memory. In *Proceedings of the 2nd Workshop on Managed Many-Core Systems (MMCS)*, page 4. Citeseer, 2009.

[54] Gernot Heiser. Virtualizing embedded systems: why bother? In *Proceedings of the 48th Design Automation Conference*, pages 901–905. ACM, 2011.

[55] Gernot Heiser and Ben Leslie. The okl4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.

[56] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[57] Richard Herveille et al. Wishbone system-on-chip (soc) interconnection architecture for portable ip cores. *Revision B*, 4, 2002.

[58] Philip Holman and James H Anderson. Adapting pfair scheduling for symmetric multiprocessors. *J. Embedded Computing*, 1(4):543–564, 2005.

[59] John Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, Devon Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[60] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on

arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261. IEEE, 2008.

[61] Asif Iqbal, Nayeema Sadeque, and Rafika Ida Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2110, 2009.

[62] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[63] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[64] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[65] James Laudon and Daniel Lenoski. The sgi origin: a ccnuma highly scalable server. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 241–251. ACM, 1997.

[66] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. *The directory-based cache coherence protocol for the DASH multiprocessor*, volume 18. ACM, 1990.

[67] 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.

[68] Ye Li, Eric Missimer, and Richard West. Predictable migration and communication in the quest-v multikernel. *arXiv preprint arXiv:1310.6301*, 2013.

[69] Jochen Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.

[70] lowRISC. Tagged memory and minion cores in the lowrisc soc. 2015.

[71] Dave McCracken. Posix threads and the linux kernel. In *Ottawa Linux Symposium*, page 330, 2002.

[72] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[73] John M Mellor-Crummey and Michael L Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *ACM SIGPLAN Notices*, volume 26, pages 106–113. ACM, 1991.

[74] John M Mellor-Crummey and Michael L Scott. Synchronization without contention. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 269–278. ACM, 1991.

[75] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *INTEGRATION, the VLSI journal*, 38(1):69–93, 2004.

[76] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[77] Rishiyur S Nikhil and Kathy R Czeck. Bsv by example. *CreateSpace, Dec*, 2010.

[78] Vincent Nollet, Théodore Marescaux, Diederik Verkest, Jean-Yves Mignolet, and Serge Vernalde. Operating-system controlled network on chip. In *Proceedings of the 41st annual Design Automation Conference*, pages 256–259. ACM, 2004.

[79] Michael Ott, Jaroslaw Zola, Alexandros Stamatakis, and Srinivas Aluru. Large-scale maximum liklihood-based phylogenetic analysis on the ibm bluegene/l. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 4. ACM, 2007.

[80] Gary Plumbridge, Jack Whitham, and Neil Audsley. Blueshell: a platform for rapid prototyping of multiprocessor nocs and accelerators. *ACM SIGARCH Computer Architecture News*, 41(5):107–117, 2014.

[81] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[82] The RTEMS Project. RTEMS Real Time Operating System (RTOS). `https://devel.rtems.org/wiki/Developer/SMP`, 2014.

[83] The RTEMS Project. RTEMS SMP– Status of effort. `https://devel.rtems.org/wiki/Developer/SMP`, 2014.

[84] Xen Project. RT-XEN. `http://www.xenproject.org/directory/directory/projects/92-rt-xen.html`, 2015.

[85] Zoran Radović and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 241–252. IEEE, 2003.

[86] Mendel Rosenblum. Vmware's virtual platform™. In *Proceedings of hot chips*, volume 1999, pages 185–196, 1999.

[87] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[88] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[89] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175–1185, 1990.

[90] Roy Spliet, Manohar Vanga, Bjorn B Brandenburg, and Sven Dziadek. Fast on average, predictable in the worst case: Exploring real-time futexes in litmusrt. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 96–105. IEEE, 2014.

[91] Udo Steinberg and Bernhard Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.

[92] Andrew S Tanenbaum. *Distributed operating systems*. Pearson Education India, 1995.

[93] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25–35, 2002.

[94] Joe Touch, Yu-Shun Wang, and Venkata Pingali. A recursive network architecture. *ISI, Tech. Rep*, (2006-626), 2006.

[95] Francisco Triviño, José L Sánchez, Francisco J Alfaro, and José Flich. Network-on-chip virtualization in chip-multiprocessor systems. *Journal of Systems Architecture*, 58(3):126–139, 2012.

[96] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando Martins, Andrew V Anderson, Steven M Bennett, Alain Kägi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[97] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 11. ACM, 2011.

[98] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. *Operating system support for improving data locality on CC-NUMA compute servers*, volume 31. ACM, 1996.

[99] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[100] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanović. The risc-v instruction set manual volume ii: Privileged architecture version 1.7. 2015.

[101] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Technical report, DTIC Document, 2014.

[102] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie,

Simon W Moore, Steven J Murdoch, and Michael Roe. Capability hardware enhanced risc instructions: Cheri instruction-set architecture. *University of Cambridge, Computer Lab., Tech. Rep. UCAM-CL-TR-850*, 2014.

[103] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[104] Richard West, Ye Li, and Eric Missimer. Quest-v: A virtualized multikernel for safety-critical real-time systems. *arXiv preprint arXiv:1310.6349*, 2013.

[105] Wikipedia. Altix — wikipedia, the free encyclopedia, 2014. [Online; accessed 24-July-2015].

[106] Wikipedia. Moesi protocol — wikipedia, the free encyclopedia, 2015. [Online; accessed 24-April-2016].

[107] Wikipedia. sel4 website, 2015. [Online; accessed 24-July-2015].

[108] Wikipedia. Monolithic kernel — wikipedia, the free encyclopedia, 2016. [Online; accessed 23-April-2016].

[109] Wikipedia. Rtems — wikipedia, the free encyclopedia, 2016. [Online; accessed 17-April-2016].

[110] M Yang. *CAL code generator for epiphany architecture*. PhD thesis, Master Thesis, Halmstad University, 2013, in preparation, 2014.