Real-Time I/O System for Many-core Embedded Systems

Zhe Jiang

Doctor of Philosophy

University of York
Computer Science
August 2018

Abstract

In modern real-time embedded systems, time predictability is vital. This extends to I/O operations which require predictability, timing-accuracy, enhanced performance, scalability, parallel access and isolation. Currently, existing approaches cannot achieve all these requirements at the same time. In this thesis, we propose a framework of hardware-implemented real-time I/O virtualization system to meet all these requirements simultaneously — **BlueIO**.

BlueIO integrates the important functionalities of I/O virtualization and low layer I/O drivers (achieved via Virtualized Complicated Device Controller (**VCDC**)), as well as a clock cycle level timing-accurate I/O controller (i.e. GPIO Command Processor (**GPIOCP**)). BlueIO provides this functionality in the hardware layer, supporting abstract virtualized access to I/O devices from the software domain. The hardware implementation includes I/O virtualization and I/O drivers provide isolation and parallel (concurrent) access to I/O operations and improves I/O performance. Furthermore, the approach includes GPIOCP to guarantee that I/O operations will occur at a specific clock cycle (i.e. be timing-accurate and predictable).

This thesis proposes the design and implementation of BlueIO, together with its components — GPIOCP and VCDC. It is demonstrated how a BlueIO-based system can be exploited to meet real-time requirements with significant improvements in I/O performance and low running cost on different OSs. The thesis presents a hardware consumption analysis of BlueIO, in order to show that it linearly scales with the number of CPUs and I/O devices.

Finally, the thesis proposes a scalable real-time hardware hypervisor termed BlueVisor, which is built upon proposed modules. BlueVisor enables predictable virtualization on CPU, memory, and I/O; together with fast interrupt handling and inter-virtual machine communication. BlueVisor shows that the approaches towards I/O proposed in this thesis can be applied and expanded to different architectures, whilst maintaining required properties.

Contents

List of Figures

List of Tables

Acknowledgement

It would not be possible to complete my Ph.D. study and to finish this dissertation without the help and support of the people around me.

Firstly, I would like to present my deepest appreciation to Prof. Neil Audsley for his persistent guidance, help and care for the past four years. Without him, my research work and this Ph.D. thesis would not be even possible. In addition, I would like to thank Prof. Andy Wellings for his assessments and comments during my whole Ph.D. study. He is always nice and patient to point out my weakness and encourage me to carry on my study. From him, I learned not only the right attitude not only on research but also on work and life. In addition, I would like to thank Dr. Ian Gray for his guidance and help duration my Ph.D. study.

I would also like to thank my parents for their unconditional support and continuously encouragement, which allows me to focus on my research and gives me the strength to overcome the difficulties and challenges from both my study and daily life.

The appreciation also goes to the members in the Real-time Systems group for their brilliant advice to my research and their encouragements. From my point of view, this the best research group in this world. I will be always proud of the group.

Finally, many thanks also go to my friends: Dr. Yunfeng Ma, Dr. Xinwei Fang, Dr. Haitao Mei, Dr. Shuai Zhao, Mr. Xiaotian Dai, Mr. Dizhong Zhu, Ms. Yanting Dai and Ms. Qianhan Xu — for blowing my mind with the discussion of research questions and support me when I run to troubles.

Declaration

I declare that all work contained within this thesis is a result of my own investigations, except where explicit attribution has been given. The content of some of the chapters has already been published within the following publications:

- Zhe Jiang, and Neil C. Audsley. GPIOCP: Timing-accurate general purpose I/O controller for many-core real-time systems. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017. [120]
- Zhe Jiang, and Neil C. Audsley. VCDC: The Virtualized Complicated Device Controller. *Euromicro Conference on Real-Time Systems (ECRTS)*, 2017. [72]
- Zhe Jiang, Neil C. Audsley and Pan Dong. BlueVisor: A scalable real-time hardware hypervisor for many-core embedded systems. *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018. [73]
- Zhe Jiang, Neil C. Audsley and Pan Dong. BlueIO: A scalable real-time hardware I/O virtualization system for many-core embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*. (Accepted to be published) [121]

  This work has not previously been presented for an award at this, or any other, University.

Chapter 1

Introduction

Recently, *embedded systems* have become widespread, e.g. in transportation systems, medical systems, mp3 players, telephone switches, etc. It is estimated that more than 99% of microprocessors are used for embedded systems [74,86]. By definition [115], an embedded system is:

> a collection of programmable parts surrounded by Application-Specific Integrated Circuits (***ASIC***s) and other standard components, that interact continuously with an environment through sensors and actuators.

In embedded systems, some component systems have bounded time constraints. Often, these time constraints have to be guaranteed. Representative examples of such systems are flight controllers in aircraft, braking controllers in cars and train control systems. In these systems, an input stimuli must receive a response before a given deadline, because any deadline miss may cause a catastrophic failure, even death. These systems are called *real-time systems*. The definition of a real-time system in [44] is:

> a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment.

In architectures of embedded and computer systems, due to the recent breakdown of Dennard scaling [58], system designers have been not been able to improve system performance by increasing processor frequencies directly. Instead, in order to maintain expected year-on-year performance increases

(also known as Moore's Law [99]), designers have turned to increase the number of cores on one chip. Nowadays, an eight-core processor is commonplace, and the number of cores on one chip is continually increasing, e.g. Knight's Landing is a commercial 72-core processor proposed by Intel [9], and Parallela has even proposed a 256-core processor [12]. Therefore, in order to efficiently achieve increased computational ability, the platforms of modern real-time systems have been moved from single-core systems to multi-core and many-core systems.

1.1 Input and Output Systems (I/O Systems)

Not only in real-time systems, but also in all embedded and computer architectures, an Input and Output system (***I/O system***) is vital, as:

- The I/O system extends the functionalities of the whole system [74];
- The I/O system provides interactive interfaces between the embedded and computer architectures and the outside world [115].

An I/O system is composed of I/O devices (peripherals), I/O controllers, and I/O drivers (specific details are described in Section 2.2).

1.2 Performance Features

In real-time systems (even single-core systems), I/O performance is a major system bottleneck [37, 72]. This mainly results from the significantly slower processing speed of normal I/O facilities compared to CPUs. This may result in the performance reduction of the whole system [72].

However, when it comes to real-time systems with more than one core, the bottleneck of I/O performance is magnified, mainly resulting from processor scheduling and contention over I/O resources. For example, in a traditional bus-based multi-processor system (e.g. an AMBA High-performance Bus-based system [45]), if an I/O operation is requested by a user application, the system has to deal with scheduling between the cores in each processor, as well as I/O resource scheduling between different processors.

This leads to the first research question:

> *Research Question 1: How can I/O performance in real-time systems be enhanced by an increased number of cores? (compared to a traditional system)*

In order to answer the question, an expected I/O system requires both enhanced I/O performance and good scalability, which are referred to as Performance Features in this thesis.

1.3 Real-time Features

As described in [44], satisfying timing constraints is a basic requirement of an I/O system in real-time systems:

- Predictability [104]: I/O operations have to be predictable in order to ensure a timely reaction when a critical situation occurs, e.g. the braking operations of a car are always required to be handled within a hard deadline [104].
- Timing-accuracy [120]: In real-time systems, I/O operations are often required to be timely — occurring at a specific clock cycle.[1] This feature is vital for both I/O devices and the whole system. Specifically, on the I/O side, timing-accurate I/O operations achieve accurate control over I/O devices. For example, the accuracy of the motor controls in a 3D printer determines the accuracy of the final printed product [49, 56, 107]. When it comes to the system side, I/O operations are often requested repeatedly and frequently with other system instructions. Frequent missing of clock cycles between I/O operations will damage the predictability of the whole system. Therefore, timing-accuracy is a vital feature for I/O devices, and even the whole system.

As defined in [106], predictability means:

[1] In this thesis, all I/O devices share a single synchronization clock source with the whole system. For example, if the frequency of the system clock is 100 MHz, the granularity of a clock cycle is 10 ns.

> It should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made, e.g., concerning failures and workloads. In other words, predictability is always subject to the underlying assumptions being made.

As defined in [120], timing-accuracy is presented as:

> If an operation occurs absolutely at the expected time, this operation is totally timing-accurate.

However, achieving predictability of an I/O system is always challenging [44], mainly resulting from the extremely uncertain execution time of I/O operations on various I/O devices, and the transmission latencies of I/O operations from a user program to a targeted I/O device. Specifically, in standard computer and embedded architectures, even in a single-core system, latencies caused by device drivers and application process scheduling make predictable and timing-accurate I/O operations problematic, often leading to a dedicated CPU for the I/O application, or to that application being made the highest priority. However, neither solution is scalable nor does it offer good predictability, timing-accuracy and resource use [120].

In multi-core and many-core systems, these issues are compounded. Whilst an application can invoke an I/O operation accurately via the interrupt of a high-resolution timer (e.g. the nanosecond timer provided by an RTOS [22, 23]), the transmission latencies from a CPU to an I/O controller can be substantial and variable due to the communication bottlenecks and contention. For example, in a bus-based many-core system, the arbitration of the bus and the I/O controller may delay the I/O request. For a Network-on Chip (***NoC***) architecture, the arbitration of on-chip data flows across the communications mesh will also increase latencies. More details are introduced in Section 2.2 and Section 2.3

This leads to the second research question:

> *Research Question 2: Apart from performance features, how can the predictability and timing-accuracy of I/O operations in multi-core and many-core real-time systems be guaranteed?*

In this thesis, we classify ***predictability*** and ***timing-accuracy*** as ***Real-time Features***.

1.4 Protection Features

In practice, hard real-time systems are often associated with safety-critical systems [39, 44, 106], since any deadline miss may cause catastrophic failure of the whole system.

In safety-critical systems, in particular for I/O operations, isolation is another vital feature. Specifically, with the number of cores increased on one chip, I/O operations may be required to occur at the same time. For example, multiple motors in a 3D printer are required to be controlled simultaneously, in order to achieve efficient and precise controls on the nozzle [122]. However, it is common for different applications to try and access the same I/O device simultaneously, and even worse, a side channel may damage access attempts from the other cores.

In this situation, an I/O system should simultaneously enable ***parallel accesses*** and ***isolation*** of I/O operations. In this thesis, we term these two features ***Protection Features***.

1.4.1 Virtualization Technology

Currently, virtualization technology is the most widely used technology [57,82] to achieve protection features (e.g. [82] [57] [66] and [109]). By definition [64], virtualization technology presents:

> A framework or methodology of dividing the resources of a computer or an embedded system into multiple execution environments, by applying one or more concepts or technologies such as

hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others.

In a virtualization system, each independent execution environment (also known as a virtual machine (***VM***)) enables a guest operating systems (***OS***) to run logically isolated, which means I/O operations requested from different VMs can never affect each other [33, 102, 113]. At the same time, the I/O operations are also prevented from being affected by other VMs, even if the VMs break down [113].

Moreover, other reasons for widespread use of virtualization in real-time systems are the superior benefits brought to the whole system, including increased resource use, reduced volume and cost of hardware and load balance [33, 102].

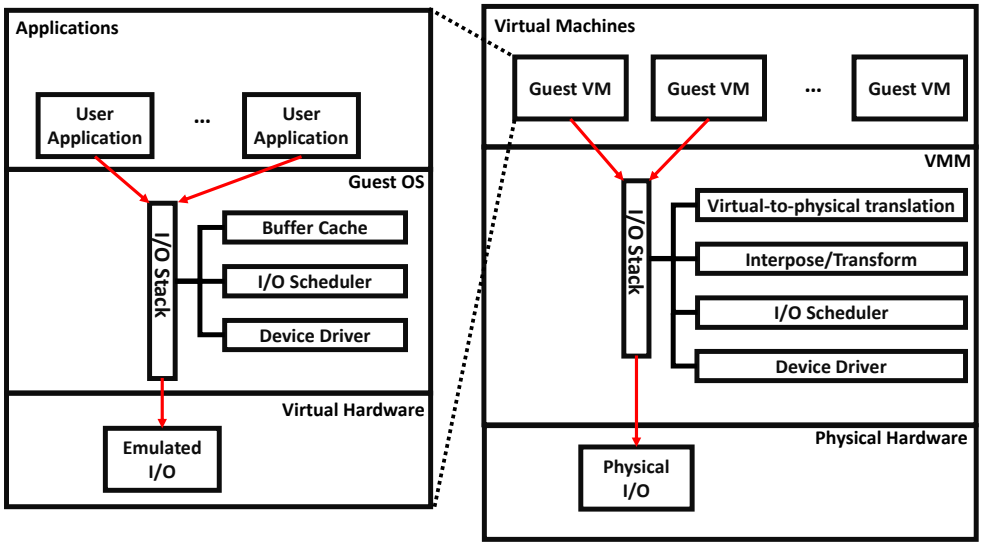


Figure 1.1: Flow of I/O Request in Traditional Virtualization System

However, virtualization technology (I/O virtualization) involves complicated I/O access paths (i.e. indirection and interposition of privileged instructions, see Figure 1.1) and complicated shared I/O resource management (i.e. scheduling and prioritization) [72, 102]. These two issues significantly conflict with the performance features (performance and scalability) and real-time features (predictability and timing-accuracy) [72,102]. Eliminating the issues and fitting virtualization technology to real-time systems is challenging.

Since virtualization technology relies on hardware support [33], today's chip manufacturers have promoted different hardware assists in order to simplify the complicated I/O access paths and assist complicated shared I/O resource management [33,94]. Intel's Virtualization Technology for Directed I/O (***VT-D***) [68], which can provide direct I/O access from guest VMs, is an example of this. The IOMMU [39] is applied to commercial PCI-based systems to offload memory protection and address translation, in order to provide fast I/O access from guest VMs. These commonly used hardware-assisted I/O virtualizations have successfully reduced the performance loss caused by complicated I/O paths, and complex shared I/O resource management, in traditional virtualized systems. However, they cannot improve I/O performance or guarantee real-time features (predictability and timing-accuracy) [33, 102, 120, 122]. For example, in [90], a hardware-based I/O virtualization approach using memory-mapped I/O, MMU and IOMMU is proposed, which achieves maximally only 73.10% of the normal DMA write data rate with no improvement on performance features or real-time features.

In order to improve real-time features, a number of real-time virtualizations have been proposed. For example, RT-Xen [116] integrates real-time scheduling theories with Xen [25] and instantiates a suite of fixed-priority servers (e.g. Deferrable Server), which is able to provide effective real-time scheduling to a guest Linux OS within a 1ms quantum. This gives good predictability, but no improvement on timing-accuracy. Similarly, Kiszka [76] proposed improvements in predictability regarding KVM, without any improvement in timing-accuracy. Generally, current approaches cannot satisfy the requirements of both performance and real-time. (Note that, more approaches are reviewed in Sections 2.4.4 and 2.4.5.) Therefore, current I/O virtualization cannot be directly applied to real-time systems. This leads to the third research question:

> *Research Question 3: How can performance features and real-time features for I/O systems be achieved when I/O virtualization is deployed (to achieve protection features)?*

To sum up, in real-time arenas, the following features are required by an I/O system simultaneously:

- **Performance features:**

  - Enhanced I/O performance;
  - Scalability.

- **Real-time feature:**
  - Predictability;
  - Timing-accuracy.

- **Protection feature:**
  - Parallel accesses;
  - Isolation.

As mentioned in Section 1.1, the I/O system is one of the most vital parts of embedded and computer architectures. Therefore, research in the area of an I/O system have to be associated with a complete system, which leads to the fourth research question:

> Research Question 4: How to integrate the ready-built I/O system to the complete system with the expected features inherited?

1.5 Hypothesis

Virtualization technology (i.e., I/O virtualization) has been shown to be a useful technique for achieving protection features to I/O operations (i.e. parallel access and isolation). However, the deployment of I/O virtualization introduces complicated I/O access paths and complex shared I/O resource managements, which leads to decreased I/O throughput, worsen scalability, more complicated timing analysis, and decreased timing-accuracy compared to a non-virtualized system — significantly conflicting to the performance and real-time features. Therefore, current I/O virtualization cannot be directly applied to real-time systems.

The hypothesis of the thesis is that:

Effective real-time I/O and Virtualization can be achieved by moving Virtualization, I/O drivers and I/O operations into hardware.

The thesis will show that moving the virtualization layer and I/O drivers from software layer to hardware layer significantly increases I/O performance compared to traditional virtualized and non-virtualized systems. Also, it will show that a programmable I/O controller contained in the virtualization system permits applications to instigate complex sequences of I/O operations at an exact time (the output values can be both static and dynamic), so achieving timing-accurate and predictable I/O operations with I/O virtualization.

Moreover, The design of the real-time I/O virtualization system is generic, which can be ported to different platforms with a scaled number of processors and I/O devices. Therefore, it can be directly applied to a real-time system, with the inherited performance features, real-time features and protection features.

1.6 Success Criteria

To facilitate the assessment of the work proposed in this thesis, a set of success criteria (SC) are given. In order to support the thesis hypothesis given in Section 1.5, the following need to be developed:

- SC-1: A virtualized complicated I/O controller that moves the functionalities of I/O virtualization and I/O drivers from the software layer to the hardware layer, which increases I/O throughput compared to both traditional virtualized and non-virtualized systems – performance features;

- SC-2: A timing-accurate I/O controller that can permit user applications to instigate complex sequences of I/O operations (with both static and dynamic output values) at an exact time, which achieves timing-accurate and more predictable I/O operations compared to traditional systems, which are real-time features (verified by experimentation).

- SC-3: A real-time I/O virtualization system built on SC-1 and SC-2 that can simultaneously support timing-accurate and predictable virtualized I/O operations with increased I/O throughput compared to traditional virtualized systems. The design of the I/O virtualization system can be scaled with a different number of processors and I/O devices;

- SC-4: A complete virtualization system built on the ready-built real-time I/O virtualization system (in SC-3), which inherits the expected real-time features (in SC-2) and performance features (in SC-1). The integration work verifies the design of the real-time I/O virtualization system (in SC-3) in an architecture agnostic way.

1.7 Structure

The thesis is structured as follows:

- **Chapter 2** Reviews the background and related research of the thesis. Firstly, real-time systems, I/O systems and I/O systems in multi-core and many-core architectures are reviewed. Then, the literature related to achieving the expected features is reviewed, including virtualization technology, timely I/O controllers and implementation platforms. At the end of this chapter, the current problems which the thesis looks at solving are given.

- **Chapter 3** Firstly describes the system context of the research. It then introduces the six expected features of a real-time I/O system and their corresponding evaluation metrics, used in the following chapters.

- **Chapter 4** Proposes a hardware-implemented I/O virtualization system called Virtualized Complicated Device Controller (***VCDC***). It permits user applications to access and operate I/O devices directly from a guest VM, bypassing the guest OS, the VMM, and low layer I/O drivers. This achieves significant performance improvements (i.e. I/O performance and scalability), containing shorter I/O response time, greater I/O throughput and less on-chip communication overheads. This is verified by evaluations in Section 4.3. This chapter provides the solution to research question 1.

- **Chapter 5** Proposes a resource efficient programmable I/O controller, termed the GPIO Command Processor (***GPIOCP***). It enables applications to instigate complex sequences of I/O operations at an exactly specific clock cycle, thus achieving real-time features (i.e. predictability and timing-accuracy) which are verified by evaluation in Section 5.4. This chapter provides the solution to research question 2.

- **Chapter 6** Proposes a real-time I/O virtualization system integrating GPIOCP and VCDC, termed BlueIO. The evaluation results in Section 6.4 demonstrate that BlueIO inherits the benefits brought by GPIOCP and VCDC real-time features (i.e. timing-accuracy and predictability) and performance features (i.e. enhanced I/O performance and scalability). Furthermore, due to the employment of I/O virtualization, significant protection features (i.e. parallel accesses and isolation) are also brought. This chapter demonstrates the solution to research question 3.

- **Chapter 7** Proposes a scalable real-time hardware hypervisor for multi-core and many-core embedded architectures, termed BlueVisor, which is built on GPIOCP, VCDC and BlueIO. BlueVisor enables predictable virtualization on CPU, memory and I/O, as well as fast interrupt handler and inter-VM communication. The establishment of BlueVisor aims to show our methodologies can be applied and expanded to different architectures and platforms, with maintained features on real-time, performance and protection as evidenced by the evaluation results in Section 7.3

- **Chapter 8** Draws the final conclusions and summarises future work.

Chapter 2

Literature Review

This chapter introduces the background and literature related to this thesis, which is divided into four major parts:

Firstly, in Sections 2.1 to 2.3, the background material related to the thesis is reviewed, which includes real-time systems, I/O systems and I/O systems in multi-core and many-core architectures. Secondly, in Sections 2.4 and 2.5, the research on achieving performance features, real-time features and protection features for I/O systems is reviewed. Thirdly, in Section 2.6, an overview of implementation fabrics that are commonly used for embedded systems is given. Finally, Section 2.7 concludes with existing problems which are expected to be solved in the thesis.

2.1 Real-time System

The literature in real-time systems is broad. This section mainly presents a top-level view in order to place our work in context. More details are given when work is used later in the thesis. In this section, we review the basic classifications of real-time systems, and two classes of approaches commonly used to measure predictability in the systems.

2.1.1 Classifications

As introduced in [44], real-time systems are split into hard real-time systems, firm real-time systems, and soft real-time systems. Common definitions of the terms are [44]:

- **Hard real-time system**: Where it is absolutely imperative that the system reacts within its given time frame, else there may be disastrous consequences.
- **Firm real-time system**: Where the deadline may be missed occasionally, but there is no benefit from it being late.
- **Soft real-time system**: Where the deadline may be missed occasionally, or a service can occasionally be delivered late.

In firm real-time systems and soft real-time systems, there may be an upper limit on the times of tasks missing deadlines. In addition, some systems may have both soft real-time and hardware real-time requirements [44]. For example, a communication system may have a soft real-time deadline of 30ms for optimal signal processing; at the same time, a hard real-time deadline of 500ms also guarantees completion of basic communication.

In this situation, it is vital to determine the range of time in which the task executes in order to show that the task is able to meet the timing constraint. While determining the range of execution time, a commonly used technique in both academia and industry is predicting the Worst Case Execution Time (**WCET**) of the task.

2.1.2 Deriving Worst Case Execution Time (WCET)

As reviewed in [114], commonly used methodologies of achieving the WCET of tasks can be mainly classified into static analysis and measurement-based analysis.

2.1.2.1 Static Analysis

Static analysis is an offline methodology, attempting to analyse and calculate the WCET of tasks via modelling the target architecture [47]. As described in [54], static analysis has three steps:

- **Flow analysis**: Reconstructing all the possible paths through a process, via the source and the final output.
- **Global low-level analysis**: Computing the factors may affecting tasks on a specific machine via its global constructs (e.g. memory accessing).

- **Local low-level analysis**: Computing the same factors as above, but localised to a single task and code segment (pipeline).

In order to ensure the analysis is accurate, it is important to establish an extremely accurate model for the architecture, including processor, memory etc. In practice, if the architecture is simple, the model can be established easily (e.g. Intel 8080 [2] uses a fixed number of clock cycles to execute different instructions). When it comes to complicated modern architectures, accurate modelling becomes almost impossible because of a mass of unpredictable factors, e.g. cache, memory, etc. For example, in the modern Intel x86 [24] system architecture, different models of memory and cache may cause uncertain timing variances which are very difficult to predict. As described further in Section 2.3, the unpredictability of I/O also increases the difficulty of static analysis.

These factors result in issues with pessimism in static analysis. Because it is very hard to assert conditions on the state of the system, static analysis has to assert worst-case conditions. As an example, worst-case I/O access latencies are normally assumed and worst-case transmission latencies between processors and I/Os have to be assumed as well. Therefore, without a large number of assumptions, it will be very difficult to determine the WCET in practice.

2.1.2.2 Measurement-based Analysis

As it is different to rely on an extremely accurate model of the architecture (static analysis), measurement-based analysis adopts the behaviour of the system itself to measure the execution time [114].

As with static analysis, measurement-based analysis starts from reconstructing a flow graph of the program. Specifically, in the flow graph, a program is re-constructed into a number of basic blocks. In the measurement, the tool executes the program many times under a set of inputs and environment conditions. It then records the duration of each block, deriving a distribution of probabilities. After this, the execution time of each block can be combined into the flow graph. If the basic blocks are connected to a sequential function, the execution times of each block are summed. If the basic blocks are connected to a parallel function (e.g. in a switch construct), the maximum execution time of all parallel blocks will be chosen. The combination of all these basic blocks forms an estimate of the WCET of the program.

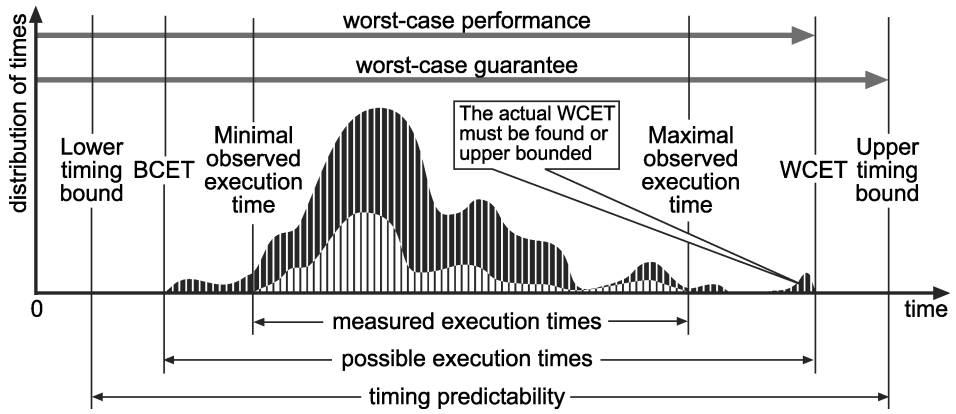


Figure 2.1: Graphical view of the execution times of a task, along with the relevant bounds [114]

In order to make the analysis sound, the worst-case path of the program and the worst-case conditions of the system have to be accurately found. Figure [114] demonstrates an example where the meanings of the items are as follows:

- **Measured execution times**: Observed maximum/minimum execution times for the program. These could not be accurate, since the actual maximum/minimum execution times are very difficult to observe.
- **BCET/WCET**: Actual best and worst-case execution times.
- **Upper/lower timing bound**: Best and worst case execution times with a safety margin added.
- **Possible execution times**: The set of all possible execution times for all different program paths, inputs and initial hardware conditions.
- **Timing predictability**: The possible range of execution times after the safety margin has been added.

In Figure 2.1, the maximal observed execution time is lower than the WCET, which is under a different condition set. As described in [114], there are two solutions to eliminate the gap. The first solution is observing the flow graph, and ensuring full program coverage has been achieved. If this cannot be ensured, a block of code with an extremely high actual execution time may be missed. The second solution is testing all possible inputs and executing

the code with a sufficient number of iterations, in order to ensure all possible system states can be tested.

Even though measurement-based analysis is simpler than static analysis techniques, it is much more difficult to assert measurement-based analysis is sound, resulting from the unpredictability of system components, including I/Os, caches, external memory etc. In the following sections 2.2 and 2.3, we will specifically introduce the unpredictability caused by I/Os, and analyse the unpredictability in different types of system architectures.

2.2 Input and Output Systems (I/O Systems)

In computer and embedded architectures, input and output (***I/O***) systems transfer information between the main memory and the outside world [53]. An I/O system is composed of I/O devices (peripherals), I/O controllers and I/O drivers (carrying out the I/O request(s) through a sequence of I/O operations). Figure 2.2 illustrates an I/O system in a traditional single-core bus-based system, e.g. AHB Bus. In the figure, the shaded blocks represent the components of the I/O system.

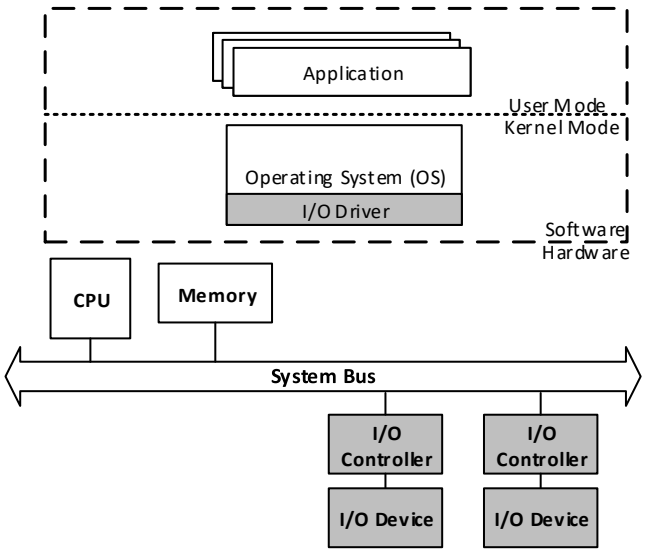


Figure 2.2: Structure of I/O System in a Conventional Bus-based System

This section is divided into two parts. Specifically, in Section 2.2.1 to Section 2.2.3, we introduce the basic idea of the three components in I/O systems. In Section 2.2.4, we discuss the reasons that I/O systems affect the

performance and real-time features of the whole system.

2.2.1 I/O Devices

In computer and embedded architectures, I/O device are computing facilities to provide user input/output, data storage and retrieval, network access capabilities, etc [53]. In order to make the controls and developments of different I/O devices to be general, classification of I/O devices is vital. In the thesis, we describe four types of commonly used classification for I/O devices.

According to the functionalities [92], I/O devices can be classified as:

- **Input devices**: Input information from a user to a computer, e.g. mouse and keyboard.
- **Output devices**: Output information from a computer to a user, e.g. screen and speaker.
- **Input and output devices**: Have functionalities of both input and output, e.g. network.
- **Storage devices**: Are used to store information, e.g. disks.

UNIX has proposed a widely used classification method of I/O devices: "depending on types of transmission, I/O devices are divided into character devices and block devices" [108] . Specifically, if the hardware device is accessed by a stream of data, it is a character device (e.g. keyboards and UART). Otherwise, if the device is accessed randomly (non-sequentially), it is a block device (e.g. disk) [108].

As described in Section 2.1.2, the WCET of a program is always varied due to the specific hardware in both static analysis and measurement-based analysis. Therefore, even with the same system architecture and the same software, if an I/O device is replaced, the WCET of the program may be different. In general, the number of I/O devices increases the likelihood of the program's WCET being affected.

Furthermore, I/O devices are also impact on the performance features of the whole system, because I/O devices are much slower than processors. Consider a common input device, a keyboard. Typing at 120 words per minute is equivalent to 10 characters per second, or 100 milliseconds between each character. A processor running at 2 GHz can execute approximately 200

million instructions during that time. If a blocking I/O operation happens, the processor will suffer from significant performance degradation.

2.2.2 I/O Controllers

In computer and embedded architectures, in order to handle I/O devices sufficiently and effectively, the following requirements are necessary [92]:

- Individual addressing of each device;
- Allowing devices to initiate communication with the processor;
- Method for transferring the bulk of data between I/O devices and memory;
- Consistent method for programs to handle I/O from extremely different devices.

All these requirements suggest that it is not practical to connect the I/O devices directly to the processor. Each device or class of devices should have its own hardware interface connected to the processor [92] — interface module or I/O controller. Therefore, instead of handling thousands of different I/O devices, programs are only required to handle dozens of interface modules (I/O controllers). In Figure 2.2, an I/O controller acts as a direct interface between the system bus and the controlled I/O device. Figure 2.3 illustrates an example of general-purposed I/O controller.

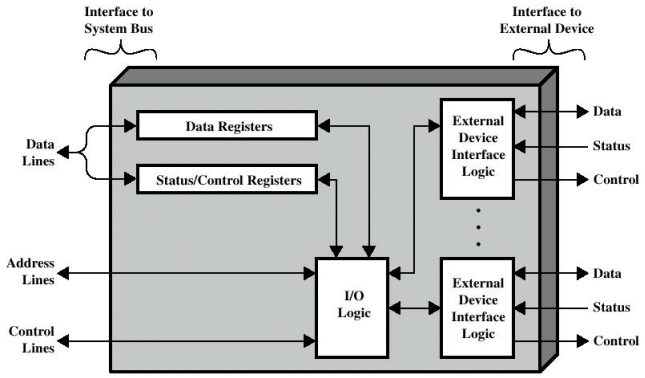


Figure 2.3: A General-purposed I/O Controller [67]

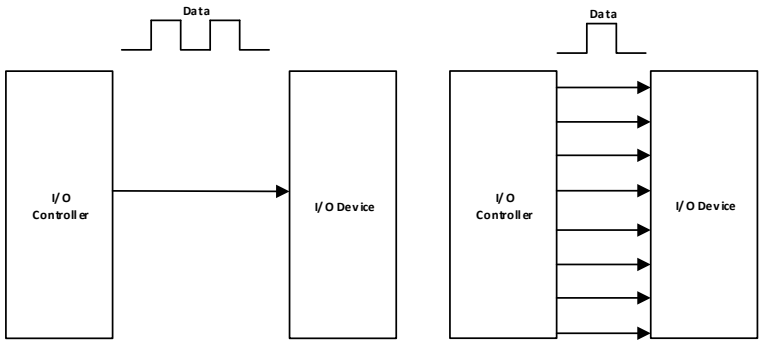
As shown in Figure 2.3, a general-purpose I/O controller has two communication interfaces, which are physically connected to a bus and I/O devices respectively:

- **Bus-side Interface**: Is responsible for buffering data to be transferred from the processor to an I/O device and allowing the processor to control the I/O device and read its status.
- **I/O-side Interface**: Is in charge of communicating with I/O devices, including data, status, control etc.

An I/O controller enables the following functionalities:

- Accepting requests from the processor to control and perform I/O operations on the connected device(s);
- Controlling and managing the connected I/O device(s);
- Buffering received data until it is transferred to memory or the connected I/O device(s);
- Directly transferring data between I/O devices and memory.

Different I/O controllers have been widely adopted in different situations. According to the types of communication interface, I/O controllers can be classified as serial I/O controllers or parallel I/O controllers. Figure 2.4 demonstrates an example of these two types of I/O controllers.



(a) Serial Controller (b) Parallel Controler

Figure 2.4: Two types of I/O Controllers

As shown in Figure 2.4, the difference between serial I/O controllers and parallel I/O controllers is the transmission method of transmitted data. Specifically, via a parallel I/O controller employed, each bit (of the transmitted data) has a single wire devoted to it and all the bits are transmitted at same time. Conversely, via a serial I/O controller employed, the bits are transmitted as a series of pulses. One of the typical parallel I/O controllers is the HDMI controller, and one of the typical serial I/O controllers is the SPI controller.

Adopting I/O controllers brings the following advantages to the whole system:

- Simplifying the interfaces between I/O devices and processors;
- Providing consistent I/O handling methods for processors across the different I/O devices;
- Allowing parallel control of different I/O devices.

However, even though timing constraints are required in both types of I/O controllers (e.g. SCLK line in SPI controller), the real-time features cannot be guaranteed. Specifically, I/O controllers cannot ensure an I/O operation occurs at a specific clock cycle, which means the timing-accuracy of an I/O operation cannot be guaranteed by an I/O controller. Moreover, even though an I/O operation can be completed within a predictable timing variance, it cannot eliminate the unpredictability caused by I/O devices (see Section 2.2.1) or buses (see Section 2.3), etc.

2.2.3 I/O Drivers

An I/O driver (also known as a device driver) is a program that operates a particular I/O device by controlling the connected I/O controller [92]. The I/O driver provides a software interface to I/O devices, which enables operating systems (***OSs***) and other programs to access hardware functions without knowing specific details (e.g. type of the I/O controller). Specifically, if a process requests an I/O device, the invoked I/O driver will translate this high layer request to serial specific instructions on the I/O controller, then the I/O controller will operate the target I/O device.

The main purpose of I/O drivers is to provide abstraction by acting as a translator between a hardware device and the programs or OSs that uses it [92]. In practice, programmers are able to write high-level application code

independently of the specific hardware the end-user is using. For example, a high-level application for interacting with a serial port may simply have a function called “printf”. At a lower level, the function “printf” calls an I/O driver function “SendToUart” to achieve data sending via a particular serial port controller installed on a user’s computer. The I/O drivers controlling a 16550 UART are totally different from the drivers controlling an FTDI serial port converter, but each hardware-specific I/O driver abstracts these details into the same (or similar) software interface (see Figure 2.5). Note that, this example deals with synchronous I/O, even though the thesis considers both synchronous and asynchronous I/Os.

As demonstrated in Figure 2.5, the specific I/O drivers of 16550 UART

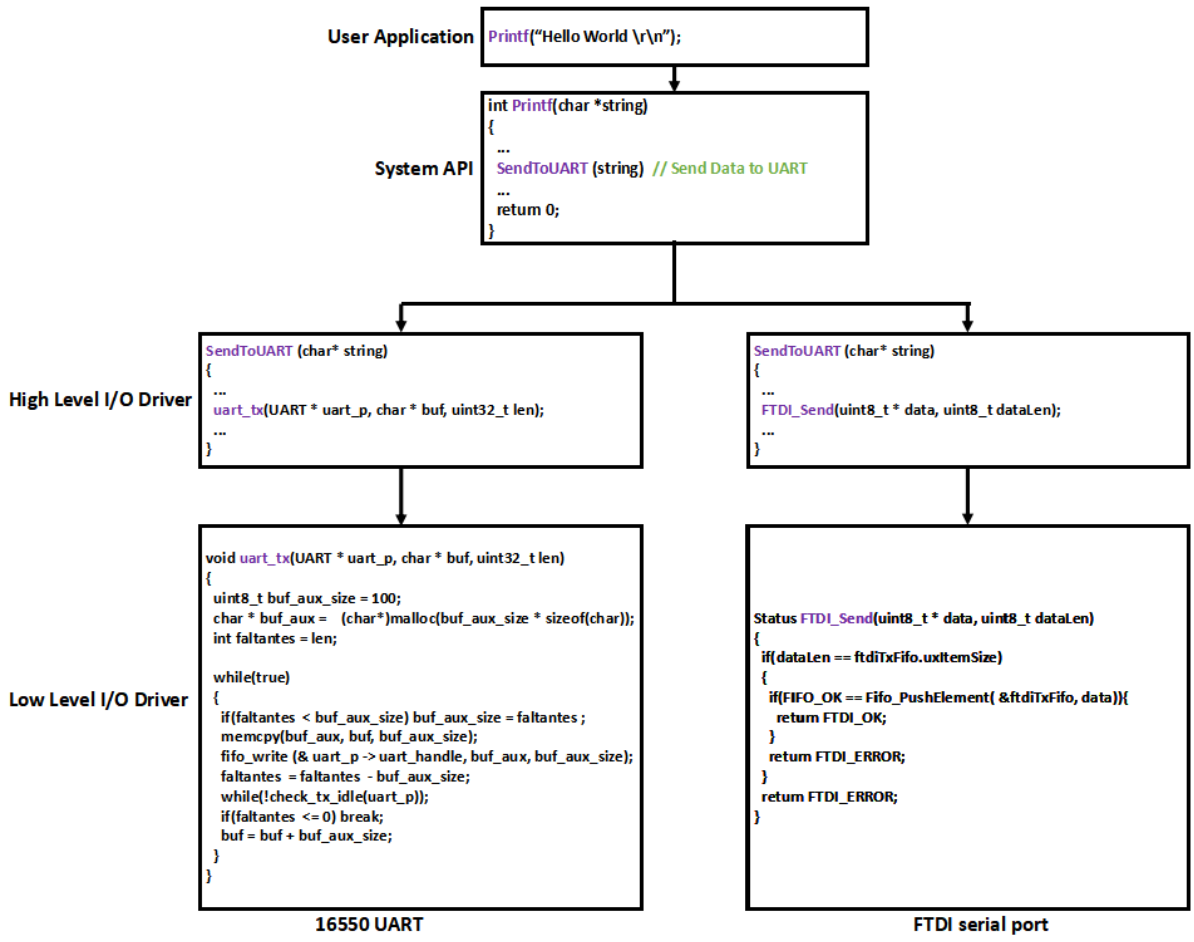


Figure 2.5: Send “Hello World” from High Layer Application to the I/O devices

and FTDI serial port converters are totally different. However, the interfaces abstracted to the programs at the high level are the same. Specifically, no matter which UART controller is used in the system, high-level programs can use a unified interface such as
"**printf ( )**" to send data.

However, having multiple different I/O drivers in a system significantly impacts real-time I/O features. Specifically, as described in Section 2.2.1, a number of different I/O devices results in a large amount of I/O drivers in the system. The execution times of these introduced I/O drivers are different, and hard to unify. Therefore, the predictability and timing-accuracy of I/O operations may worsen. Furthermore, the different I/O drivers also lead to extra software overhead to the system, which can affect overall system performance. In this research, software overhead is measured using memory footprint, see Section 6.4.1 and 7.3.1.

2.2.4 Conflict to Performance and Real-time Features

As introduced at the beginning of Section 2.2, an I/O system is mainly composed of three modules: I/O devices, I/O controllers, and I/O drivers. Furthermore, as described in Sections 2.2.1 to 2.2.3, interference between I/O devices can detract from the performance and real-time features of I/O systems, and even the whole system.

2.2.4.1 Conflict to Real-time Features

As reviewed in Section 2.1.2, the methods of achieving WCET are typically classified as static analysis and measurement-based analysis. Of these two methods, flow analysis is compulsory (reconstructing all the possible paths of an operation). Therefore, in order to estimate the WCET of a complete I/O operation, it is necessary to calculate the WCET consumed on its corresponding driver, controller and device.

Currently, because the number of I/O devices is increasing, it is almost impossible to achieve the WCET of executing a specific I/O operation on all kinds of I/O devices. Specifically, even in a system with the same architecture and software, if the involved I/O device is replaced, the WCET of the same I/O operation may suffer from significant variance. With I/O drivers, this issue is magnified. Therefore, in order to drive such different I/O devices,

various I/O drivers are added to systems. However, the WCETs of executing all these different drivers cannot be unified or predicted.

Moreover, latencies caused by device drivers and application process scheduling make timing-accurate I/O control problematic, often leading to a dedicated processor for the I/O application or that application being made the highest priority, neither solution is scalable or offers good resource utilisation. With RTOS employed, e.g. [22] [14], an application can invoke an I/O operation accurately via the interrupt of a high-resolution timer (e.g., the nanosecond timer), the transmission latencies from a processor to an I/O controller can be substantial.

Transmission Latency v.s. Jitter

Note that in this research, transmission latencies cannot simply be treated as jitter.

In real-time systems, jitter is defined as the worst-case time a task can spend waiting to be released after arrival [36]. However, the latency may be have a number of reasons, such as application scheduling, core scheduling, processor scheduling, bus scheduling, bus contention, I/O contention, etc. If we just treat the latency as jitter, the timing analysis may become easier, but it may make the timing analysis more unreliable.

The only situation that we can treat the transmission latency as jitter is when the I/O operation is requested in a single-core system with constant output values. However, this thesis mainly focuses on general systems, which is not only the special situation.

2.2.4.2 Conflict to Performance Features

In order to support as many I/O devices as possible, a large number of I/O drivers have to be integrated into a system, e.g. in OSs [101]. The integration of so many I/O drivers results in two main drawbacks to performance features:

- **Significant software overhead**: I/O drivers have to be loaded in to memory while running, which consumes a large amount of memory footprint and significant processor overheads.
- **Longer response times of I/O operations**: The operations of I/O drivers in OSs increase the response times of I/O operations, which reduces I/O throughput significantly.

In order to alleviate these issues, the micro-kernel proposed by Regnecentralen [81] allows I/O drivers to be optionally added into an OS kernel depending on actual requirements. The deployment of a micro-kernel efficiently reduces the software overhead and increases the I/O performance compared to a traditional system. Moreover, exo-kernel [55] has completely removed I/O drivers from the OS kernel (kernel space) and allowed developers to write I/O drivers in user programs (user space). Compared to micro-kernel based systems, user applications with exo-kernel based systems are able to optimise the I/O drivers according to particular requirements, which achieves less software overhead and better I/O performance. Neither method completely eliminates software overhead nor achieves enhanced I/O performance.

When it comes to multi-core and many-core systems, these issues are magnified even further in real-time features and performance features . In the following section, we introduce systems evolved from single-core to many-core. This is followed by specific reasons why the issues are magnified.

2.3 The Move to Multi-core and Many-core

Currently, due to the breakdown of Dennard scaling [58], it has become difficult to continuously increase clock speed/frequency of processors. Furthermore, as feature sizes become smaller, so do the interconnecting wires. Because the resistance of a wire is inversely proportional to its size, and the capacitance is proportional to its length, the capacitor time constant dictates the maximum wire length, as wires are made thinner ($T = RC$). (Note: T implies time constant; R implies resistance; and C represents capacitance [58]). Therefore, the area of a chip which is reachable in a single clock cycle [34] quickly makes large circuits at a high clock speed unfeasible.

Instead of using the conventional method of increasing processor frequency in order to meet the expected year-on-year performance increase, i.e. Moore's Law [99], designers have started to increase the number of cores on one chip.

In this section, we review the two kinds of classic multi-core and many-core system architectures: bus-based multi-core systems (in Section 2.3.1), and NoC-based many-core systems (in Section 2.3.2). Section 2.3.3 then introduces and analyses real-time and performance features of the I/O systems in these systems.

2.3.1 Bus-based Multi-core System

With the number of processors on one chip increasing, different processors have to deal with the communication and operations between both shared memory and I/O devices. In typical embedded systems, these communication operations are normally achieved via adopting a shared bus, e.g AHB bus [45] (see Figure 2.6).

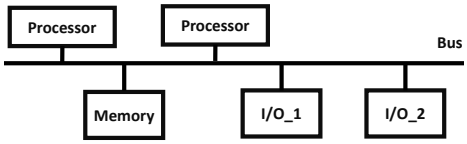


Figure 2.6: An Example of a Shared Bus

However, classical shared buses suffer as the number of cores continues to increase. Specifically, in a system with a shared bus, once a processor accesses an I/O device, the whole bus will be locked. Therefore, other processors cannot initiate a request, even if the requested destination is a completely different I/O device.

In order to alleviate this issue, crossbar interconnects has been proposed. Specifically, crossbar interconnects connect all processors and I/O devices through a set of switch-boxes, and employ dedicated links which replace a shared bus, e.g. AXI interconnect [30] (See Figure 2.7).

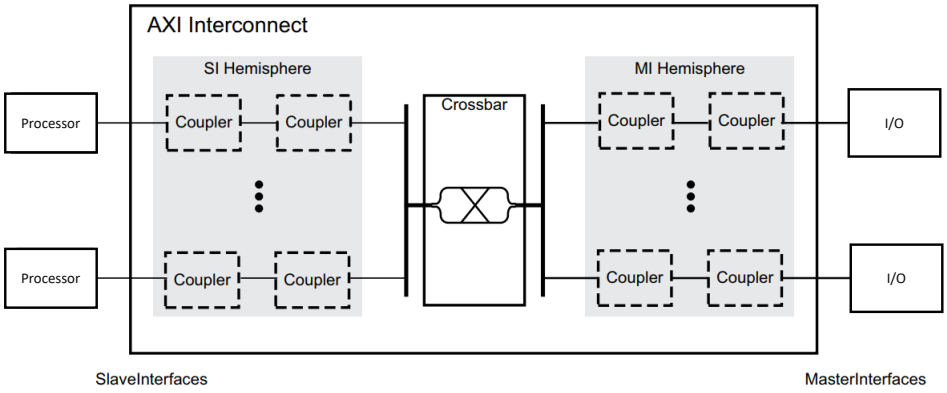


Figure 2.7: An Example of Crossbar Interconnects (AXI Bus) [30]
SI = Slave Interface; MI = Master Interface

With crossbar interconnects, multiple transactions are able to occur at the

same time, which provides great improvements on system performance. However, extra hardware consumption is also generated, due to the requirement for a large number of switches, which may damage the maximum possible clock frequency. In a digital design, the maximum clock frequency is inversely proportional to the design area [32, 112].

The issue with scaling of processors has led to system designers moving to network-on-chip (NoC) based approaches [95].

2.3.2 NoC-based Many-core System

The NoC architecture consists of three basic components (link, network interface and router). An example of NoC architecture with 3*3 topology is shown in Figure 2.8. It can be seen that communication between IPs is completed by a set of routers which are linked together through physical links. The connections between IPs and routers are provided through Network Interfaces (NIs).

Different from bus-based multi-core systems (including crossbar interconnects systems), NoC-based many core systems attach each processor to a small network router (see left part of Figure 2.8), then each message is encapsulated into a network packet and routed over the network based upon some routing scheme (see right part of Figure 2.8).

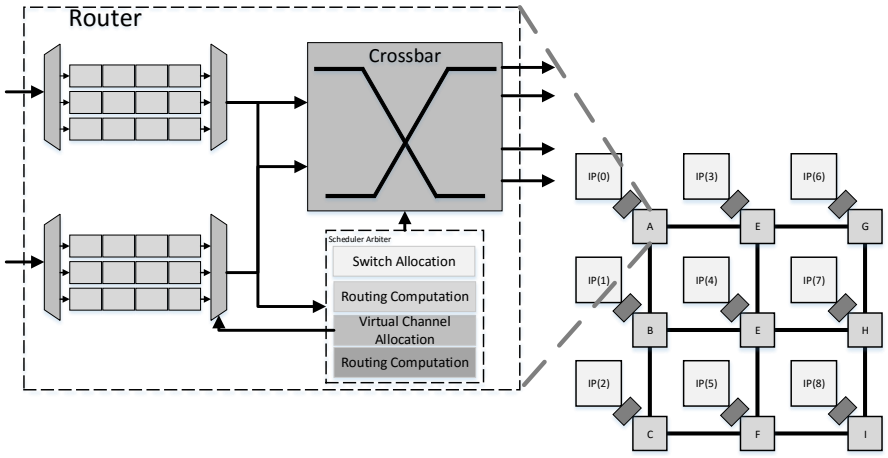


Figure 2.8: Examples of NoC and Router Architectures
A - G: Routers; IP: Intellectual Property core

In a NoC, packets are routed in any direction at each router to their des-

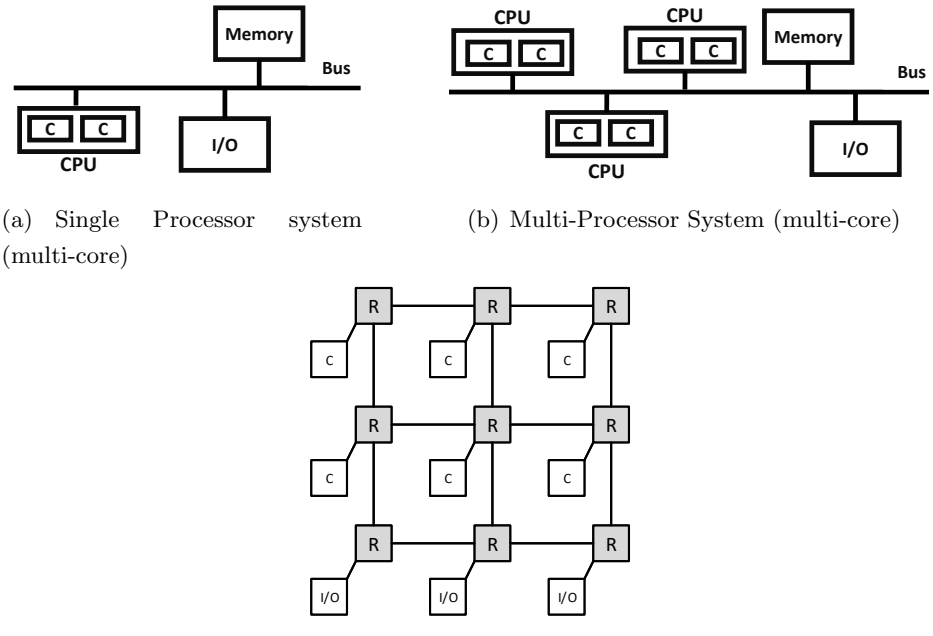
tination. Commonly, they will be routed in an X-Y fashion, where they will first be routed to the correct column within the network, then either up or down the column to their target [43, 63, 88] are examples of current widely used NoCs.

Compared to bus-based multi-core systems, overheads resulted from communication have been distributed from a single bus or a big switch to multiple independent routers in a NoC, which has successfully solved the bottleneck of clock frequency resulting from the scaling of processors.

2.3.3 I/O Systems in Multi-core and Many-core Systems

In multi-core and many-core systems, the most significant challenges associated with I/O systems are loss of performance and lack of real-time features, resulting from the complicated I/O resource management, i.e. scheduling and prioritisation [72].

Because this section focuses on I/O resource managements, we only classify systems into single processor systems (multi-core), multi-processor systems (multi-core) and many-core systems (NoC-based), see Figure 2.9



(a) Single Processor system (multi-core)

(b) Multi-Processor System (multi-core)

(c) Many-core System

Figure 2.9: Structure of Multi-processor and Many-core Systems with I/O devices. C - Core, R - Router/Arbiter

- **Single processor (multi-core) system (Figure 2.9(a))**: User applications can normally request and operate I/O devices by modifying memory-mapped registers. The overhead of I/O resource management mainly comes from the scheduling of the processor — deciding which core has priority to access the I/O device. This procedure is normally handled by an OS.

- **Multi-processor (multi-core) system (Figure 2.9(b))**: Apart from processor scheduling, contention over I/O devices is unavoidable when a shared I/O is to be accessed. To solve the issue of I/O contention between processors, hardware mutexes are normally added in multi-processor systems, which causes extra hardware overhead as well as high bus workload (frequent communication is required between processors and hardware mutex).

- **Many-core System (Figure 2.9(c))**: All arbitration between cores is controlled by the system arbiter (e.g. the routers in a NoC-based system), therefore processor scheduling is not required. However, many-core systems still suffer from I/O contention when different cores need to access I/O devices at the same time.

In general, complicated I/O resource management has main three drawbacks for the system as a whole:

- **Significant system overhead**: Processor scheduling is mostly implemented at the software level, and I/O contention is mostly handled at the hardware level, which both consume significant system overhead.

- **I/O operations with bad timing features**: The complexity of I/O management makes I/O operations difficult to predict, and timing-accuracy cannot be guaranteed.

- **Bad scalability**: With the number of cores and processors increasing in a system, the of resource management workload will be also increased, which causes more serious performance reduction of the whole system.

As described in Section 2.3.2, NoC is the main trend of many-core architectures. Therefore, we review the I/O systems in popular NoC architectures.

Related approaches for I/O systems over many-core NoC architectures can be divided into those that use a standard architecture (as demonstrated

in Figure 2.9(c)) and those that introduce a dedicated unit for handling I/O operations. Technologies such as Programmable Logic Controllers (***PLC***s) and associated I/O controllers are out-of-scope. The following reviews the standard architectures of I/O systems in NoCs. Note: dedicated units will be reviewed in Section 2.5.

Typical NoC based architectures (i.e. Figure 2.9(c)) that have been implemented in silicon contain integrated I/O devices connected to the edge of the mesh, e.g. Tilera's TILE64 [17] and Kalray's MPPA-256 [48]. Specifically, the TILE64 requires processors within the mesh to instigate I/O operations, with a shared I/O controller passing the operations to the actual I/O devices – hence significant latencies will occur between I/O command instigation and actual I/O occurring, which detracts from predictability and timing-accuracy. The MPPA-256 provides 4 I/O subsystems, with I/O operations instigated by the processor passed to the Resource Manager (***RM***) cores within one of these I/O systems, depending which device is required. The MPPA-256 RM cores are essentially Linux based processors controlling many devices (although real-time OS RTEMS [14] can also be used). Hence, predictable and timing accurate controls (real-time features) of many external devices connected to the GPIO are not possible. In addition, neither approach is resource efficient, as independent processors are required for I/O controls.

2.3.4 Real-time Many-core Architectures

As described in Section 2.3.3, modern many-core systems mainly focus on functionalities. Therefore, these systems cannot be directly applied in a real-time system. This section examines research which uses many-core architecture in a real-time system, e.g. XMOS [85] and PicoChip [52].

XMOS is a real-time many-core architecture proposed at the University of Bristol [85], which can be scaled from a single-core system to a thousands-core system. In particular, the architecture proposes real-time communication channels, a real-time scheduler and predictable instruction sets. The experimental results show that the system supports predictable inter-core communication and instructions handling. On the I/O side, the processors are able to read the I/O pins using a special interface – I/O reading becomes quick and efficient.However, the architecture suffers from the following issues: 1. The system requires a particular instruction set, which has to be a general propose. 2. Even though the system is able to provide more predictable I/O

access, it cannot eliminate the transmission latency, so is not timing-accurate.

Similar to XMOS, the PicoChip designed by picoChip Designs Ltd [52] proposes a many-core architecture for real-time systems. However, the system mainly focuses on the architecture designs, e.g. switches and communication channel. The system cannot provide an efficient solution for issues on the I/O side.

In addition, neither project supports the virtualization technology, and therefore do not have an efficient method to support the protection features.

2.4 Virtualization Technology

Virtualization technology is a general term for the abstraction of computing resources from their physical implementation. In computer and embedded architectures, the abstracted resources are frequently the processors, memory and I/O devices (e.g. Ethernet, keyboards, etc.). However, the term can be used much more generally and can relate to almost any part of a system.

Virtualization introduces a layer in the abstraction hierarchy of a system which exposes a set of virtual resources on top of which items at higher abstraction levels (e.g. OSs and applications) can be implemented, whilst the mapping of virtual resources to physical resources is hidden.

This section has two main parts. Sections 2.4.1 to 2.4.3 consider virtualization technologies, including classification, analysis of conflicts between virtualization technology and expected features and research on real-time virtualization technologies. In the second part (Sections 2.4.5 and 2.4.6), I/O virtualization is the main topic, containing descriptions and reviews of I/O virtualization technologies, as well as related hardware assists.

2.4.1 Notions of Virtualization

In virtualization systems, the abstraction layer is called the virtual machine monitor (***VMM***) or hypervisor. It hides the physical resources (e.g. processors, memory, I/O devices, etc.) from the upper layers (e.g. OSs, user applications). Because physical resources are directly controlled by the VMM rather than the OSs, it becomes possible to execute different OSs in parallel on the same hardware. As a result, the hardware resources are partitioned into one or more logical units, termed virtual machines (***VM***).

2.4.1.1 Virtual Machine (VM)

Virtual Machines (***VMs***) are the most well-known application of virtualization and are commonly adopted in all areas of computing. VMs became popular in the early 1970s as an alternative to system simulation [65], including the run-time interpretation of the entire ISA of simulated processors and simulation of the memory and system buses. The simulated machine becomes a VM.

In [98], a VM is defined as:

> An efficient, isolated duplicate of a real machine.

The software running on each VM is guest software, which is defined as [68]:

> Each virtual machine is a guest software environment that supports a stack consisting of an operating system (OS) and application software. Each operates independently of other VMs and uses the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform.

2.4.1.2 Virtual Machine Monitor (VMM)

The paper also presents the idea of a virtual machine monitor (***VMM***):

> A piece of software running on the host machine that enables this virtualization to take place.

The main responsibility of a VMM is abstracting the single physical resource to multiple virtual resources for Guest VMs and hiding the physical resources and other VMs. In a virtualized system, multiple guest OSs are able to execute on their own virtual resources in parallel without knowledge of the existence of other VMs, because all shared resources are controlled by the VMM.

The definition of a VMM has the following features:

- **Equivalence**: Software operations executed by the VM have to be the same as if it were run on a native machine, even if the VMM hosts multiple VMs.
- **Resource control**: The VMM has complete control over the virtual resources.

2.4.1.3 Protection Features

Deployment of virtualization brings superior benefits for the whole system, e.g. reduced volume and cost of hardware, load balance, etc. In this research, we mainly focus on the protection features of isolation and parallel accesses.

- **Isolation**: In a virtualization system, VMs are logically isolated, which means the applications executed in one guest VM can never affect other VMs, even if it breaks. Moreover, isolation can be divided into spatial and temporal isolation. Specifically, with spatial isolation, a partition (i.e. VM) is completely allocated to a unique address space (e.g. code, virtual I/O resources, etc.). This address space is not accessible by other partitions (i.e. VMs). With temporal isolation, a partition (i.e. VM) is executed under a cyclic policy. The execution of a partition is not impacted by others [109]. Note that this research mainly focuses on temporal isolation (see Chapter 7).

- **Parallel accesses**: Because all VMs are logically isolated, applications on different VMs are allowed to directly access their own resource (virtualized) in parallel without recognising the existence of other VMs.

Currently, there are a large number of modern VMMs available, which are also more frequently known by the modern term "hypervisor". We review some of the classic VMMs in Section 2.4.4, while reviewing the classifications of virtualization technology.

2.4.2 Classification of Virtualization

Currently, virtualization technologies are normally classified as:

- **Bare-metal or hosted virtualization [65]**: According to whether a VMM is run either on the hardware directly or run on top of a host OS, virtualization technologies can be classified as bare-metal virtualization (Type-1) or hosted virtualization (or Type-2).

- **Full and para-virtualization [65]**: Depending on whether the guest OS is required to be modified by adding hyper-calls into the VMMs, virtualization technologies can be classified into full-virtualization or para-virtualization.

2.4.2.1 Bare-metal and Hosted Virtualization

Hosted virtualization (Type-2) is widely used in the desktop market, as its compatibility means it can be easily ported to different platforms [65]. In a hosted virtualization system, the VMM has to be executed above the OS as a program, see Figure 2.10.

Hence, extra software overhead has been introduced. Moreover, the efficiency and performance of the guest OS has also been reduced significantly, compared to the original system.

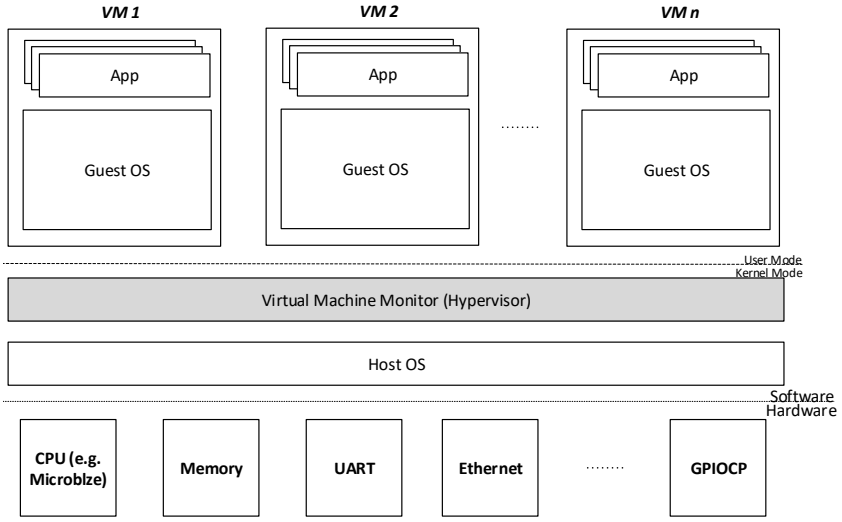


Figure 2.10: Hosted-Virtualization

In order to alleviate the introduced software overhead, bare-metal virtualization (Type-1) was proposed. In practice, bare-metal virtualization is widely used in the server market because of its lighter software overhead [65, 100]. With bare-metal virtualization, the VMM executes directly on hardware, controlling and synchronising the access of guest OSs to the physical resources. Figure 2.11 demonstrates the architecture.

Compared to hosted virtualization, bare-metal virtualization consumes less software overhead and gives better system performance, since no hosted OS is required. For example, Xen [25] is a typical example of bare-metal virtualization. As evaluated in [64], the system performance of the Guest OS in a Xen-based system is able to achieve 60% of a native OS.

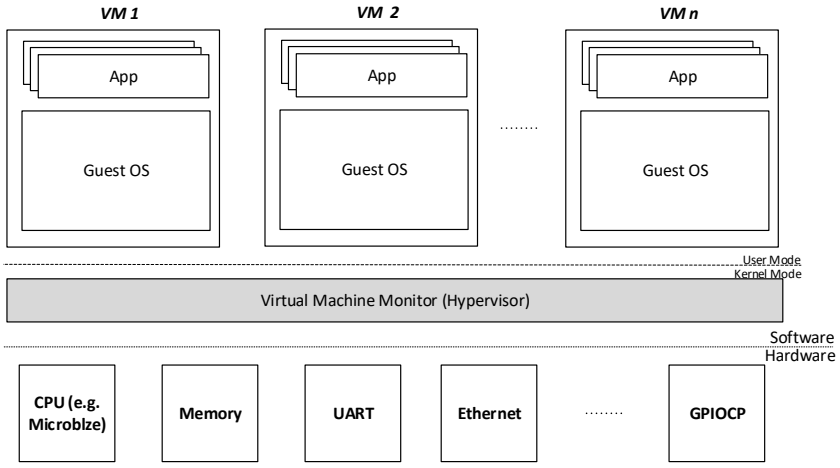


Figure 2.11: Bare-metal virtualization

2.4.2.2 Full and Para-Virtualization

Another way to classify virtualization technologies is full virtualization vs. para-virtualization. Full virtualization allows the guest OS to be executed on the VMM without any modification, while para-virtualization requires the guest OS to be modified by adding hyper-calls into the VMM. Representative hosted (Type-2) full virtualization solutions include KVM, VirtualBox, Microsoft Virtual PC, VMWare Workstation. Representative bara-metal (Type-1) para-virtualization solutions include Xen, L4, VMWare ESX. There are some research attempts at constructing bare-metal (Type-1), full virtualization solutions, such as Kinebuchi *et al.* [75] which implements such a solution porting the QEMU machine emulator to run as an application on L4Ka::Pistachio microkernel; in turn, an unmodified guest OS can run on top of QEMU; Schild *et al.* [103] has successfully executed an unmodified guest OS on L4. There are also hosted (Type-2) para-virtualization solutions, e.g. VMWare MVP (Mobile virtualization Platform) [38].

2.4.3 Conflict in Performance and Real-time Features

The most significant challenges in virtualization technologies are the loss of performance features and real-time features, which mainly result from indirection and interposition of privileged instructions, as well as complicated shared resource management (i.e. scheduling and prioritisation) [59] [100].

2.4.3.1 Indirection and Interposition of Privileged Instructions

When an application running within a VM issues a privileged instruction, (e.g. a system call and an I/O request), the processors traps into the VMM – emulating the privileged operations on the VM state that the VMM manages.

Figure 1.1 depicts the flow of a privileged instruction (i.e. an I/O request) in a conventional hosted and full virtualization system. Specifically, when an application running within a VM issues an I/O request, it is initially processed by the I/O drivers of the guest OS kernel (running within the VM). The device driver in the guest VM issues the request to a virtual I/O device, and the VMM then intercepts.

This indirection and interposition of privileged instructions poses difficulties for the virtualization system [113]:

- *Significant software overhead* [102] – Most of these operations are processed in the software, which causes significant processor overhead.
- *Larger response time of the privileged instructions* [102] [113] – Compared with an original system, virtualization technology requires more time to handle the same instruction from a guest OS, also causing a decline in system performance. This issue significantly conflicts with the performance features.
- *Decreased predictability* [120] – Longer access paths resulting from the dissimilar access paths of different requests (from user applications to hardware devices) increase the uncertainty of access times, detracting from predictability. This issue significantly conflicts with the real-time features.

2.4.3.2 Complicated Shared Resource Management

Managing and scheduling shared resources is another overhead of virtualization technology. A VMM should be responsible for all shared resource management introduced in Section 2.3.3. Generally, complicated shared resource management results in the following drawbacks for the whole system:

- *Significant system overhead* - CPU scheduling is mostly implemented at the software level, and shared resource contention is mostly handled at the hardware level, both consuming significant system overhead.

- *Decreased predictability* - The complexity of shared resource management makes the system difficult to predict. This issue conflicts with real-time requirements.
- *Bad scalability* - With the number of cores and CPUs increasing in a system, the workload of resource management will also increase which causes a more serious performance reduction of the whole system. This issue significantly conflicts with the performance features.

In order to eliminate the issues described above, a number of real-time virtualizations have been proposed. Some related research is reviewed in the following section.

2.4.4 Real-time Virtualization

In this section, we review some real-time virtualizations, which mainly include Xen-based (Type-1) solutions and KVM-based (Type-2) solutions. Currently, Xen and KVM are the most popular VMMs of the virtualization technologies.

2.4.4.1 Xen-based Solutions (Type-1 Virtualization)

Cherkasova *et al.* [46] review and evaluate three CPU schedulers in Xen, Borrowed Virtual Time (***BVT***), Simple Earliest Deadline First (***SEDF***) and Credit. Note that, due to the deprecation of BVT, we only discuss SEDF and Credit in the thesis.

In Xen, the default configured scheduling algorithm is Credit Scheduler. This implements a proportional-share scheduling strategy where a user is able to adjust the CPU share for each VM. Moreover, the VMM also features automatic workload balancing of virtual CPUs (***vCPUs***) across physical cores (***pCPUs***) on a multi-core processor. This algorithm ensures that no pCPU will be idle when there is a runnable vCPU in the system. Each VM is associated with a *weight* and a *cap*. Once the cap equals 0, the VM will receive extra processor time unused by other VMs in Work-Conserving (***WC***) mode. Conversely, if the cap is larger than 0, it limits the amount of processor time given to a VM to not exceed the cap in Non-Work-Conserving (***NWC***) mode. By default, the credits of all runnable VMs are recalculated in intervals of 30ms in proportion to each VM's weight parameter and the scheduling time slice is 10ms.

When it comes to the SEDF scheduler, each VM is able to specify a lower bound on the CPU reservations that it requests by specifying a tuple of slice and period. Therefore the VM can receive at least slice time units during each period time units. Different from the Credit scheduler, SEDF is a partitioned scheduling algorithm that does not allow VM migrations between different cores, which no global workload balancing is required.

Masrur *et al.* [83] presented improvements to Xen's SEDF scheduler. As presented in the paper, a VM is able to use its whole budget (slice) within its period even if it blocks for I/O before using up its whole slice (in the original SEDF scheduler, the unused budget is lost once a task blocks). Moreover, a certain critical VM is able to be designated as a real-time domain and given higher priority than the other domains scheduled with SEDF. The limitation of this work is that each real-time domain is constrained to contain a single real-time task. In their evolved work [84], this limitation has been removed by introducing a hierarchical scheduling architecture — both hypervisor and guest VMs deploy deadline-monotonic fixed-priority scheduling. Additionally, they have also proposed a method for selecting optimum time slices and periods for each VM in the system to achieve schedulability while minimizing the lengths of time slices.

RT-Xen [116] is the first real-time hypervisor scheduling framework for Xen [25]. It integrates compositional and hierarchical scheduling architecture within Xen and instantiates a suite of fixed-priority servers (e.g. Deferrable Server). Empirical evaluation shows that RT-Xen can provide effective real-time scheduling to a guest Linux OS within a 1ms quantum, which is excellent predictability. Currently, RT-Xen has not provided real-time features for I/O requests. In addition, the software implementation of RT-Xen implies an introduced software overhead and reduction of system performance. Similar work to RT-Xen can be seen in [117] [71].

Yoo *et al.* [117] proposed the Compositional Scheduling Framework [117] in Xen-ARM. Jeong *et al.* [71] developed PARFAIT on Xen-ARM, a hierarchical scheduling framework. Moreover, Lee *et al.* [78] improved the soft real-time performance of the Credit Scheduler in Xen. Yu *et al.* [118] proposed enhanced real-time improvements to the Xen Credit Scheduler, so that real-time vCPUs are always given priority over non real-time vCPUs and can preempt any non real-time vCPUs that may be running.

2.4.4.2 KVM-based Solutions (Type-2 Virtualization)

KVM is a representative VMM in Type-2 virtualization (hosted virtualization), using Linux as the host OS. Therefore, any improvements to the Linux kernel are directly inherited in improvements to the VMM in KVM, e.g. real-time patches on Linux.

Kiszka [76] proposed improvements to real-time features regarding the KVM. Firstly, real-time VMs are assigned with real-time priorities. Moreover, para-virtualized scheduling interfaces were also integrated, which allows task-grain scheduling by introducing two hyper-calls for the VMs: 1). informing the VMM regarding the current priority of the running task; 2). informing the VMM when an interrupt handling is completed.

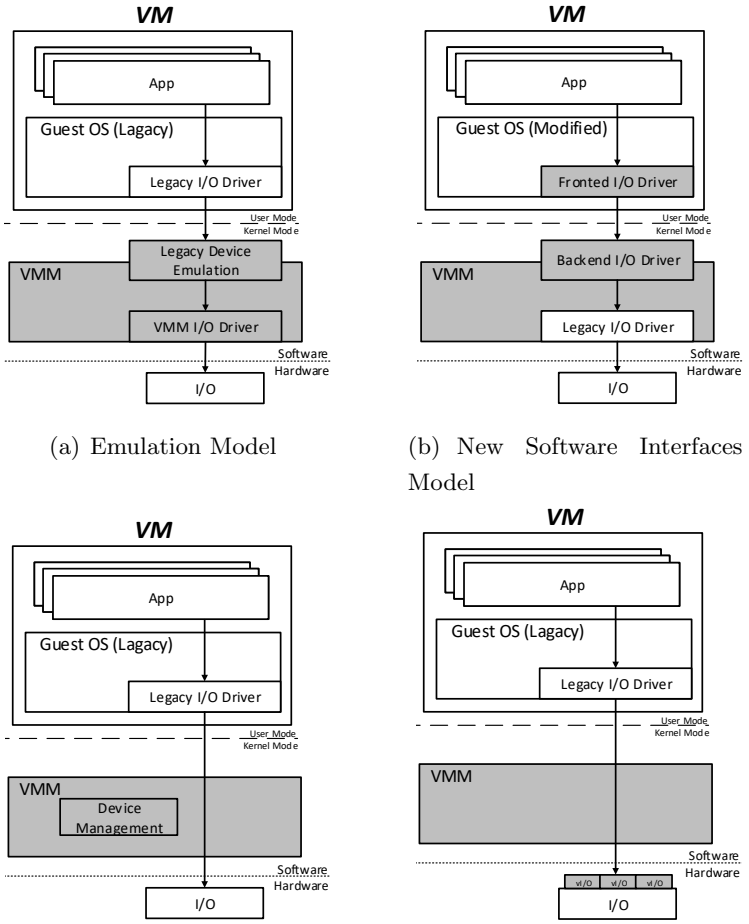
The work has efficiently improved the performance features of KVM, however it is no longer a strict full virtualization system like the traditional KVM. Zhang *et al.* [119] introduced two enhancements in real-time features to KVM via coexisting RTOS and GPOS VMs – giving the guest RTOS vCPUs higher priority than GPOS vCPUs and using processor shielding to dedicate which core to the RTOS guest and shield it from GPOS interrupts. Experimental results indicate that the RTOS interrupt response latencies are reduced. Evolved from [119], Zuo *et al.* [123] introduced additional improvements by adding two hyper-calls, enabling a guest OS to boost priority of its vCPU when a high-priority task is started.

Currently, some research has been carried out on real-time virtualization, although little of it includes I/O (most focus on processor virtualization). However, even though some work has proposed solutions to I/O virtualization (e.g. Quest-V [80] and [69]), it cannot satisfy the requirements of performance features and real-time features simultaneously. In the following sections, I/O virtualization and related improvements are reviewed.

2.4.5 I/O Virtualization

A VMM has to support virtualization of I/O requests from VMs. The I/O virtualization may be supported by a VMM through any of the following models:

- ***Emulation model*** (Figure 2.12(a)): A VMM might expose a virtual I/O device to VMs by emulating an existing (legacy) I/O device. A VMM emulates the functionalities of I/O devices in the software layer,



(a) Emulation Model

(b) New Software Interfaces Model

(c) Assignment Model

(d) I/O Device Sharing Model

Figure 2.12: Models to Achieve I/O Virtualization
(The grey parts are involved in Virtualization implementation)

no matter what physical I/O devices are available in the hardware layer. Emulation models provide good compatibility (existing I/O drivers can be used directly), but suffer from the limitations of performance features, such as lower I/O throughput and longer I/O response time.

- ***New software interfaces model*** (Figure 2.12(b)): The model is similar to the emulation model, but instead of emulating a legacy I/O device, the VMM exposes synthetic device interfaces to VMs. The synthetic device interfaces are defined to enable efficient I/O virtualization, with enhanced performance. However, this model suffers from reduced compatibility, resulting from the requirements of a modified guest OS or

drivers.

- ***Assignment model*** (Figure 2.12(c)): A VMM may assign the physical I/O devices to VMs directly. The I/O drivers executed in each VM allow the guest OS to interact with the physical I/O devices directly with minimal involvement, or even no involvement of the VMM. However, in order to achieve a robust I/O assignment, additional hardware assistance is required, which guarantees the accesses of the assigned device are isolated (see Section 2.4.4.1).

- ***I/O device sharing model*** **(extended from assignment model)** (Figure 2.12(d)): In this model, an I/O device may support multiple functional interfaces. Moreover, each interface may be independently assigned to an independent VM. The I/O device is capable of accepting multiple I/O requests through any of these functional interfaces and processing them using the device's hardware resources.

Depending on specific requirements, a VMM may support any of the above models for I/O virtualization. For example, the I/O emulation model is suited for full-virtualization, meanwhile I/O assignment model achieves the best performance while hosting I/O-intensive workloads within VMs. In addition, the new software interfaces model gains a trade-off between compatibility and performance and the I/O device sharing model efficiently reduces the software overhead.

Generally speaking, the emulation model (Figure 2.12(a)) is associated with full virtualization. Specifically, emulation is implemented within a standalone VMM, which multiplexes a physical I/O device to multiple virtual I/O devices, meanwhile abstracting access interfaces to the legacy drivers in each VM, e.g. VMWare Workstation [38]. With VMWare Workstation, an I/O request (privileged instruction) sent from a guest OS always traps into VMM. Afterwards, the VMM decodes/translates the trapped request and maps it to the corresponding physical I/O device. This process is transparent but not efficient [64], since the trap handling suffers from significant overhead and unpredictable processing time. Therefore, the emulation model does not meet the requirements of performance features and real-time features for a real-time I/O system. Moreover, with the emulation model, the VMM has to fully control the physical I/O devices, which is complicated to implement and if a

physical I/O device is changed or upgraded, the VMM has to be modified as well.

In the new software interfaces models (Figure 2.12(b)), split drivers are required for the back-end and front-end. Specifically, back-end drivers are executed in the VMM and provide special interfaces to the front-end drivers in guest OSs. An I/O request sent to a front-end driver is always transmitted to a back-end driver. It will then be interpreted and mapped to a physical I/O device via the legacy I/O driver. The most well-known VMM using the new software interfaces model is Xen [25]. The software interfaces model does not require direct control of I/O devices by the VMM, therefore its implementation is simpler. Research focused on the software interfaces models has successfully achieved better performance features and real-time features by modifying the kernels of guest OSs and simplifying the VMM, e.g. RT-Xen [116] (which will be specifically introduced in Section 2.4.4.1). The most difficult challenge in the software interfaces models is the requirement of modification of a guest OS, which means legacy OSs cannot be executed directly.

Different from the emulation model and the new software interfaces model, the assignment model (Figure 2.12(c)) proposes an I/O virtualization model with minimal or even no involvement of the VMM. This model eliminates the extra overhead generated by the VMM and improves I/O performance significantly [113]. However, additional hardware assistance is required to achieve robust and isolated I/O assignment. Examples of hardware assistance models are Intel's VT-d [68] and AMD's IOMMU [39]. In an assignment model, the hardware assistance (i.e. VT-d and IOMMU) ensures the isolation of I/O address space between different VMs. More details are described in Section 2.4.6.

Extended from the assignment model, an I/O device sharing model (Figure 2.12(d)) does not require the involvement of hardware assistance — I/O virtualization is achieved by I/O devices. Specifically, in an assignment model, I/O virtualization is associated with I/O devices — an I/O device virtualized to multiple virtual I/O devices and allocated to different VMs. Single Root I/O virtualization (***SR-IOV***) [90] and Multi Root I/O virtualization (***MR-IOV***) [90] are the representatives of the assignment model. Note that, more details will be described in Section 2.4.6.

As described above, conventional I/O virtualization models (i.e. emulation model and new software interfaces model) significantly conflict with per-

formance features and real-time features. In order to eliminate these issues, the trend of I/O virtualization is developed towards hardware-assistance (i.e. assignment model and I/O device sharing model).

2.4.6 Hardware-assisted I/O virtualization

Hardware-assisted virtualization over a multi-core or many-core architecture is a steadily growing field that is gaining momentum. In this section, we review three typical related approaches: Intel's VT-d [68] (assignment model), SR-IOV [16] (I/O device sharing model) and a real-time I/O subsystem for commercial-off-the-shelf-based (***COTS-based***) embedded systems. Note that, since the technologies adopted by AMD's IOMMU [39] are extremely similar to Intel's VT-d, we only review and discuss VT-d in this thesis.

VT-d is hardware support for isolating and restricting device access to the owner of the partition managing the device, developed by Intel [68]. VT-d includes three key capabilities: 1). allowing an administrator to assign I/O devices to guest VMs in any desired configuration; 2). supporting address translations for device DMA data transfers; and 3). providing VM routing and isolation of device interrupts. Generally speaking, VT-d uses an emulation model to provide a hardware VMM that allows user applications running in the guest VMs to access and operate the I/O devices directly. Compared with conventional software virtualization, VT-d offloads most of the overhead of virtualization to the hardware level. In a system with VT-d, in addition to I/O drivers, extra drivers for VT-d are also required in the software layer. In [90], a hardware-based I/O virtualization approach using memory-mapped I/O, MMU, and IOMMU is proposed, which achieves maximally only 73.10% of the normal DMA write data rate. In addition, real-time properties (predictability and timing-accuracy) of the system cannot be guaranteed.

Single Root I/O virtualization (SR-IOV) is a specification, which proposes a set of hardware enhancements for the PCIe device. SR-IOV aims to remove major VMM intervention for performance data movement to I/O devices, such as the packet classification and address translation. A SR-IOV-based device is able to create multiple "light-weight" instances of PCI function entities (also known as VFs). Each VF can be assigned to a guest for direct access, but still shares major device resources, achieving both resource sharing and high performance. Currently, many I/O devices already support the SR-IOV specification, such as [50] and [51]. Similar to the Intel VT-d, to support a

SR-IOV-based I/O more drivers are needed in the software, which reduces the performance of the I/O. Additionally, real-time features cannot be guaranteed.

In [40], a real-time I/O management system is proposed, which comprises 1) real-time bridges with I/O virtualization capabilities, and 2) a peripheral scheduler. This proposed framework is used to transparently put the I/O subsystem of a commercial-off-the-shelf-based **(COTS-based)** embedded system using real-time scheduling, minimizing the timing unpredictability due to the peripherals sharing the bus. As described in the experiments, the proposed real-time I/O management system efficiently improves the uncertainty of I/O requests, which achieves predictability, but without timing-accuracy. Additionally, the system performance cannot be improved.

Generally, current popular hardware-assisted I/O virtualization technologies have efficiently eliminated performance reduction compared to a system with traditional I/O virtualization. However, they are not able to support enhanced I/O performance. Moreover, the requirements of real-time features cannot be guaranteed.

2.5 Programmable Timely I/O Controllers

As described in Sections 2.2 and 2.3, one of the main issues in real-time I/O operations in multi-core and many-core systems is the transmission latency from an application to the I/O devices resulting from application scheduling, processor scheduling and I/O contention.

In order to reduce this latency, hardware assistants are commonly used. For example, Solarflare NIC ASIC [15] is a network adaptor enabling user applications access to hardware directly from the software layer, bypassing the OS kernels, as shown in Figure 2.13.

As demonstrated in [15], the lowest I/O response time can be reduced to less than 500 nanoseconds in a system with Solarflare NIC ASIC, resulting from the significantly reduced transmission latency, as the OS kernel is not required. However, hardware assistants can only reduce the transmission latency, rather than eliminate it.

In order to eliminate the latency, using a programmable timely I/O controller has become a popular trend. With a programmable timely I/O controller, a user application is able to pre-program the controller to operate an I/O device, rather than requesting the I/O device when required. The deploy-

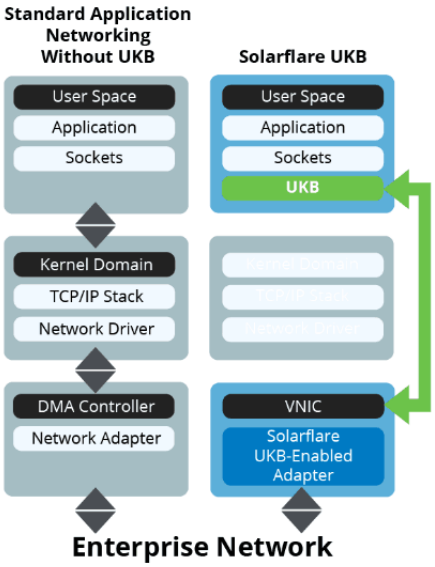


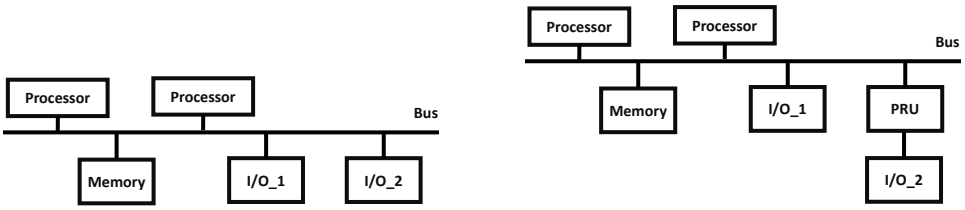
Figure 2.13: Traditional System and a System with Solarflare NIC ASIC [15]

ment of programmable timely I/O controllers has removed the transmission latency generated by application scheduling and processor scheduling. In this section, we review two classical programmable timely I/O controllers, which could be connected to a NoC mesh for GPIO control – TI's Programmable Real-time Unit (***PRU***) [42] (see Section 2.5.3) and Freescale's Time Processor Unit (***TUP***) [18] (see Section 2.5.2). Note that, both I/O controllers are designed to output constant data. If the output data is non-constant, there will still be jitter when the data is sent.

2.5.1 Programmable Real-time Unit (PRU)

Programmable Real-time Unit (***PRU***) is a low-latency, deterministic real-time I/O subsystem designed by TI [42], which is deployed along with ARM cores in the Sitara AM335x, AM437X, AM5x processors and AMIC10 SoCs. In a PRU-based system, PRU is physically connected between the system bus and I/O devices, which enables user applications to gain low-latency I/O controls via pre-programming, see Figure 2.14.

Inside each PRU subsystem, two 200-MHz real-time RISC cores are contained. The execution time of each instruction executed on each core is fixed at one cycle – 5 ns. Since real-time cores are not equipped with an instruction pipeline, single-cycle instruction execution is ensured. The PRU's small,



(a) An Example of a Conventional Embedded System

(b) An Example of a PRU-based Embedded System

Figure 2.14: Comparison of Conventional and PRU-based Embedded Systems

deterministic instruction set with multiple bit-manipulation instructions can always be executed within a predictable timing variance, which decreases the timing uncertainty of the sub-system. Moreover, the 5 ns cycle time of instruction execution time, as well as low-latency data transfers and high-speed I/O accesses assure that I/O operations can be performed in a predictable period of time [42].

Even though PRU provides increased predictability of I/O operations, it suffers from the following two drawbacks: 1) cannot guarantee that an I/O operation occurs at a specific time in the future – i.e. not timing-accurate; 2) the requirement of two 32-bit RISC cores generates additional hardware overhead – i.e. not resource efficient.

2.5.2 Time Processor Unit (TPU)

Another widely used pre-programmable real-time I/O controller is the Time Processor Unit(***TPU***) developed by FreeScale [18]. Similar to PRU, TPU is a co-processor independent of the main processor which is responsible for predictable I/O operations. Therefore, the architecture of a TPU-based system is very similar to the PRU-based system, see Figure 2.14(b).

2.5.3 Programmable Real-time Unit (PRU)

A Programmable Real-time Unit (***PRU***) is a low-latency, deterministic real-time I/O subsystem designed by TI [42], which is deployed along with ARM cores in the Sitara AM335x, AM437X and AM5x processors and AMIC10 SoCs. In a PRU-based system, the PRU is physically connected between the system bus and I/O devices, which enables user applications to gain low-

latency I/O controls via pre-programming, see Figure 2.14.

Inside a TPU, a set of pre-programmed functions have been integrated into the ROMs, with different ROMs for different I/O controls, e.g. SPI, I^2C, etc. Moreover, a TPU also provides interfaces for designers to create customised I/O control functions and save them in the RAMs. These ready-built I/O control functions can be executed on the two RISC cores inside the TPU, which are physically connected to the I/O devices. Predictable timing control of each I/O operation can be guaranteed by a 16-bit time base connected to each RISC core.

Different from the PRU, the customised interfaces enable developers to simplify complicated I/O controls and create brand new I/O controls according to specific requirements. However, the TPU suffers from the same drawbacks as the PRU, non-timing-accurate I/O controls and significant hardware overhead.

Moreover, no matter whether PRU or TPU is used, I/O controls are associated with the RISC cores. Therefore, the number of cores in each programmable timely I/O controller indicates the number of I/O operations that can be handled in parallel. This limitation significantly conflicts with the requirements of scalability (performance feature) and parallel access (protection feature) in a real-time I/O system.

2.6 Implementations Fabrics for Embedded Systems

In this section, we present an overview of the implementation fabrics that are widely used in embedded systems. Determining an appropriate fabric is vital in the system design, since it may result in a drastic effect on the efficiency of the system.

This section contains four subsections. Section 2.6.1 introduces the history of Application-Specific Integrated Circuits (***ASIC***s), followed by an analysis of the advantages and disadvantages. Section 2.6.2 illustrates the concepts of Field-Programmable Gate Arrays (***FPGA***s), including the history, architectures, and analysis of the benefits and drawbacks. In Section 2.6.3, comparisons between ASICs and FPGAs are presented. The design flows of ASICs and FPGAs are introduced in Section 2.6.3, which aims to demonstrate how to make hardware designs generic to both ASICs and FPGAs.

2.6.1 Application-Specific Integrated Circuits (ASICs)

As described in [97], Application-Specific Integrated Circuits (***ASIC***s) are integrated circuits designed to satisfy a particular purpose. In order to be application-specific, an ASIC is custom-built at dedicated silicon fabrication plants, which results in an extremely high cost to set up the plant — generating a *photo-mask*. A photo-mask is normally used to lay out the various layers of silicon to compose the final design in an ASIC fabrication process.

The main advantages brought by ASICs is the extremely high transistor density. In an ASIC design, over 100 million transistors can be contained in a tiny area. Compared to a similar system built using stock parts, the requirements of power in an ASIC system is significantly lower, as well as the maximum clock frequency being higher. As described in [105], the same design implemented on an FPGA is around three times slower than that design implemented on an ASIC. This disparity may be even larger in certain designs [105].

There are two main drawbacks to ASICs: 1) The cost of photo-mask (required by ASIC) is incredibly high, e.g. the price of a 45nm photo-mask can reach up to $0.75m. This drawback determines that an ASIC is only cost-effective when a large mount is required. 2) The design of an ASIC has to be completely fixed during fabrication time. This disadvantage determines that the design of the ASIC has to be perfect before being manufactured as it cannot be corrected even if errors are later found. Moreover, these two drawbacks also result in high costs for simulation and verification. This lack of flexibility results in a trend towards using Programmable Logic Devices (***PLD***s).

PLDs try to maintain the speed and integration levels of ASICs, as well as achieving the same amount of flexibility, which aims to eliminate the disadvantages of ASICs. In the early stages, PLDs were very simple, which only allowed the synthesis of single combinatorial logic functions. Later, as integration increased, the effective logic density of these devices also increased leading to the development of FPGAs, which are truly 'reprogrammable ASICs'.

2.6.2 Field-Programmable Gate Arrays (FPGAs)

FPGAs belongs to the *gate arrays* class in PLDs. In gate array architectures, resources (i.e. transistors, logic gates and other active devices) and interconnecting wires have been placed in a lattice pattern. The interconnecting wires

are configurable and can be arranged to connect the resources via a *routing* process.

FPGAs were firstly proposed in the 1980s [41] and labelled as an alternative method of ASIC verification. Before FPGAs were proposed, the verification of an ASIC required either the design to be manually built by connecting discrete components or fabricated as a custom-built ASIC, with both costing significant time and money. The feature of re-programmability provided by FPGAs has eliminated the issues. The prototyping method of FPGA not only increased the efficiency of ASIC design, but also opened up a new implementation method. Because of the increment in size and speed, as well as the decrement to the cost, FPGAs have been widely used in embedded systems instead of ASICs [41]. With FPGA, the significant set-up costs resulting from ASICs have been avoided.

2.6.2.1 FPGAs Architectures

On a primary FPGA, the basic resources are Configurable Logic Blocks (***CLB***s), interconnects and input/output blocks (***IOB***s), which are shown in Figure 2.15.

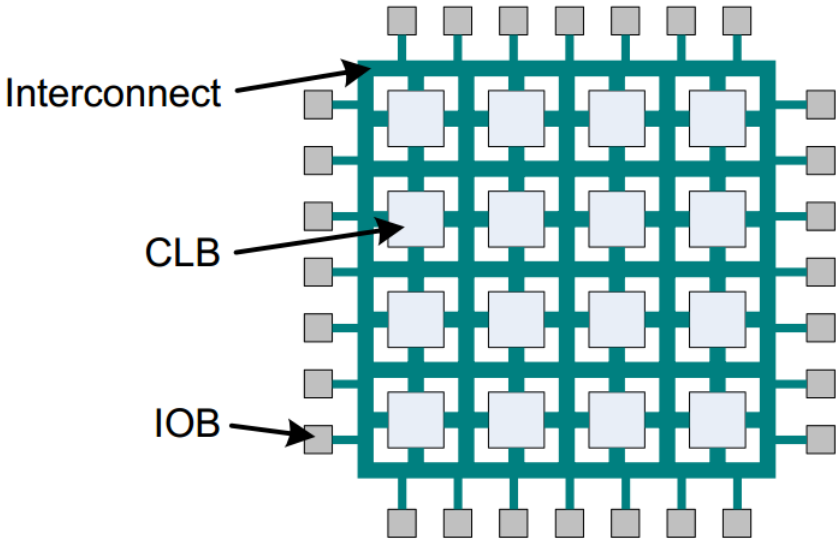


Figure 2.15: Early FPGA Architecture [29]

In an FPGA architecture, CLBs are the majority of the components used to create the sections of logic that implement the primary functionality of the device. They are built on Look-up Tables (***LUT***s) and flip-flops. CLBs are

connected to each other by interconnects which can be configured to route signals across the FPGAs.

When it comes to modern FPGA architectures, interconnects follow a hierarchical model. The majority of interconnects are constructed from short wires (termed 'local interconnects'), which can be used to span a small number of CLBs. Local interconnects are normally used to connect the IOBs of adjacent CLBs to form a single large logic function, e.g. a shift register. As well as local interconnects, 'global interconnects' in the FPGA are longer wires which can be used to span the entire components. Compared to the number of local interconnects, the number of global interconnects is much less because of the high cost. Among global interconnects, 'global clock nets' take charge of propagating clock signals throughout the FPGA.

Whilst FPGAs are composed of LUTs and interconnects, some additional modules are also included in modern FPGA architectures, in order to satisfy specialised requirements. The introduction of additional modules results in better design flexibility compared to a conventional FPGA. In order to perform complicated control operations, some ready-built processors are also integrated into high-end FPGAS, e.g. the ARM cores are contained in Xilinx ZC706 [31]. Because the implementation of the ARM processors is ASICs, their clock frequencies are significantly higher than the soft processors synthesised from FPGA (e.g. Microblaze [10]).

Moreover, in order to store data or programs, storage units are also required by FPGAs. Due to the inefficiency of the RAM synthesised by LUTs, ready-built RAM blocks are also commonly integrated in model FPGA architectures, with Block RAMs in Xilinx and Altera FPGAs(***BRAM***s) [29]. Currently, the size of BRAMs owned by the largest Xilinx FPGA (i.e. VC709) is nearly 54 Megabytes, which can be highly configurable with different widths, depths and number of access ports [29].

Currently, many other ready-built elements have been integrated into FPGA fabrics. For example, dedicated multiplier units, clock management circuits etc. These ready-built elements always have higher efficiency than the elements synthesised by FPGAs.

2.6.3 ASICs vs FPGAs

This section compares the features of ASICs and FPGAs, and summarises their advantages and disadvantages [97] [41]. One of the fabric designs is

chosen according to the specific requirements.

- ***Performance(ASIC)***: an ASIC is around three times faster than a FPGA while achieving the same design. In certain designs, this disparity may be even larger [41].
- ***Cost(ASIC)***: once the photo-mask is established, the cost of producing in an ASIC is much lower than an FPGA [41].
- ***Power(ASIC)***: compared to the same system built on an FPGA, the requirement of power in an ASIC system is significantly lower [41].
- ***Analog Circuit(ASIC)***: an ASIC can support analog circuits, and mix signal designs, which are generally not possible in an FPGA [97].
- ***Time-to-market(FPGA)***: due to simpler manufacturing steps and no requirement of a photo-mask, the FPGA always needs less time from design to product as achieving the same design in an ASIC [41].
- ***Reprogramability(FPGA)***: A FPGA can be reprogrammed in a few minutes, while an ASIC may take more than 4-6 weeks to make the same changes [41]. Note that, an ASIC design can be modified, but requires more complicated procedures [41].

2.6.4 Design Flows

This section firstly introduces the design flows in ASICs and FPGAs. It then describes the similarities and dissimilarities between the two processes. The main aim of this section is to show how a hardware design can be generic to both ASICs and FPGAs.

2.6.4.1 Design Flow of ASICs

As introduced in [79], the design flow of ASICs is divided into front-end designs and back-end designs, see Figure 2.16. Specifically, the front-end designs include specification, RTL coding, simulation, synthesis and pre-layout timing analysis, meanwhile the back-end design includes auto-place-rout (***APR***), back annotation, post-layout timing analysis and logic verification.

- ***Specification***: defines the features and functionalities of an ASIC chip. Moreover, Chip planning is also performed.

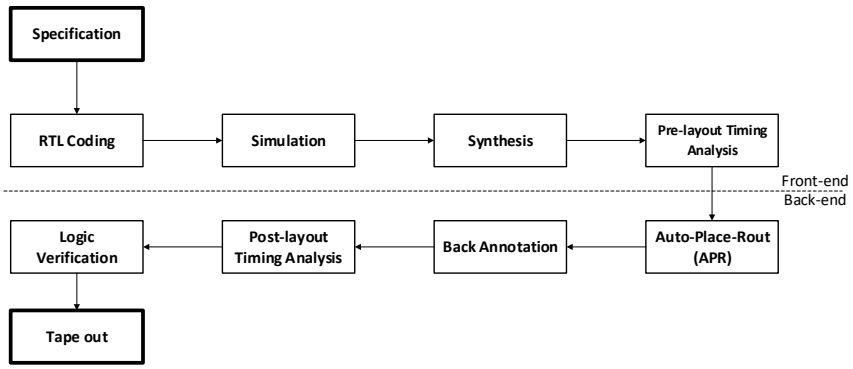


Figure 2.16: Simple Design Flow of ASICs

- ***RTL Coding***: implements the architectures and micro-architectures derived from the specification. The implementation is coded in synthesisable RTL, e.g. VHDL, Verilog etc.
- ***Simulation***: generates the test benches to simulate RTL code. A test bench is basically a wraparound environment surrounding a design. It injects a specified set of stimuli into the inputs of the design in order to check if the outputs of the design match designer expectations.
- ***Synthesis***: converts the RTL code into logic gates which have the same logic functionalities as described in the RTL code.
- ***Pre-layout Timing Analysis***: builds the timing model for the synthesised RTL code, which performs the timing analysis of the design. This process catches any possible timing violations in the design when used across specified temperature and voltage range.
- ***Auto-Place-Rout (APR)***: places and routes the synthesised logic gates. This process owns some degree of flexibility, therefore, the designers can place the logic gates of each module according to a pre-defined floor plan.
- ***Back Annotation***: extracts the RC parasitics in the layout.
- ***Post-layout Timing Analysis***: catches real timing violations, e.g. hold and setup. This step is similar to pre-layout timing analysis, but focuses on physical layout information.
- ***Logic Verification***: acts as a final sanity check to ensure the design has the correct functionalities. After this, the ASICs can be taped out.

2.6.4.2 Design Flow of FPGAs

Similar to the design flow for ASICs, the design flow for FPGAs is also divided into front-end design and back-end design, see Figure 2.17.

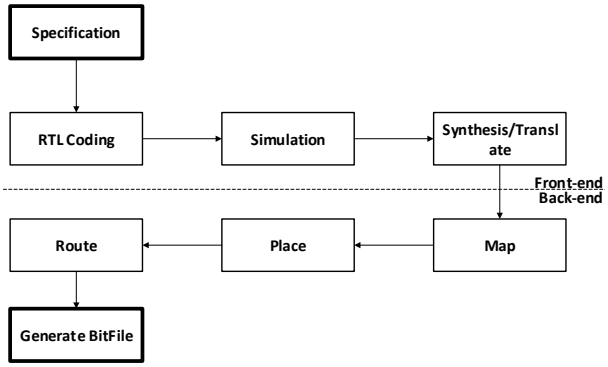


Figure 2.17: Simple Design Flow of FPGAs

Compared with the ASICs design flow (see Figure 2.16), the front-end designs in the FPGAs design flow are nearly the same, except that pre-layout timing analysis is not required. However, the back-end designs in the FPGAs are totally different from the back-end designs in the ASICs design:

- ***Map***: allocates translated/synthesised logic gates into CLBs and other atomic elements of the target FPGA fabric.
- ***Place***: configures mapped CLBs onto the device. Because the placement algorithm is a version of the bin packing problem (NP-complete), this step always takes a long time.
- ***Route***: finalise the interconnects between the placed CLBs. This stage tries to use the shortest interconnection lines to reduce the propagation delay and power consumption.
- ***Bitfile Generation***: converts the placed and routed design to a bitfile, which can be used to configure a target FPGA and can be applied to an exact FPGA.

2.6.5 Generic Fabric Designs

As introduced in Sections 2.6.4.1 and 2.6.4.2, the design flows of ASICs and FPGAs are divided into back-end designs and front-end designs. Furthermore,

the front-end designs in both fabric designs are extremely similar, apart from the synthesis stage. Therefore, a front-end design can be easily ported from an ASIC to a FPGA and vice versa.

Normally, the front-end designs are implemented in Hardware Description Language (***HDL***), e.g. VHDL, Verilog, etc. In the synthesis stage of both fabric designs, HDL-built designs are synthesised/translated into logic gates, and can then be used in the back-end designs. Therefore, the gate level design is generic, which can be implemented in the back-end designs on an FPGA or an ASIC.

The real-time I/O system proposed in this thesis is mainly designed and implemented in front-end hardware design. Therefore, the prototype system can be easily ported between FPGAs and ASICs.

2.7 Summary and Problem Statements

As introduced in Chapter 1, real-time I/O systems simultaneously require performance, real-time guarantees and protection.

In this chapter (Chapter 2), we firstly reviewed the background of the thesis, i.e. real-time systems, I/O systems and system architectures. We then reviewed work in the area of performance, real-time and protection features, i.e. virtualization technologies and programmable timely I/O controllers. Finally, we introduced the two types of implementation fabric for embedded systems, ASICs and FPGAs. This chapter is used in the thesis when exploring the hypothesis presented in Section 1.5. In this section, we summarise the major research described in this chapter and detail the existing research problems.

Section 2.1 introduced the concept of real-time systems and what they mean for *predictability*, for example, deriving the WCET. Two typical methods used to estimate the WCET were described, static and measurement-based analysis.

Section 2.2 introduced the idea of the three main components of an I/O systems, I/O devices, I/O controllers and I/O drivers. Performance features and real-time features of I/O systems were discussed and analysed, which leads to the following research issues:

In I/O systems, the variety of I/O devices and I/O drivers

results in complexity in deriving the WCETs of I/O operations, which makes guaranteeing predictability problematic. At the same time, the transmission latencies caused by device drivers and application process scheduling make timing-accurate I/O control problematic. Moreover, the amount of I/O drivers integrated into the systems lead to a significant software overhead and longer I/O response time, which conflicts with performance features.

Therefore, the I/O systems significantly conflicts with real-time features and performance features.

Section 2.3 introduced two typical classes of multi-core and many-core architectures, i.e. bus-based multi-core architectures and NoC-based many-core architectures, followed by review and analysis of the I/O systems in these architectures. This section leads to the following research problems:

With the number of cores increasing, systems move from single-core architectures to multi-core and many-core architectures. The introduction of resource management (processor scheduling and I/O contention) in multi-core and many-core architectures magnifies the reduction in performance and real-time features caused by conventional I/O systems.

Therefore, the performance features and real-time features of I/O systems in multi-core and many-core architectures can be worse than the I/O systems in single-core architectures.

Section 2.4 discussed virtualization technologies, which can be divided into two parts. The first part of this section introduced the basic ideas and classifications of virtualization technology, in order to clarify that they bring superior protection features. The analysis of the popular virtualization technologies (state-of-art) implies the following research difficulties:

Virtualization technology brings protection features, via the indirection and interposition of privileged instructions, as well as complicated shared resource management. These two

main issues lead to extra system overhead (in both hardware and software), longer response time of the privileged instructions, reduced predictability and scalability.

Therefore, with virtualization technologies, the performance features and real-time features of I/O systems are reduced even further compared to multi-core and many-core architectures. This is evidenced by the evaluation results in Chapter 3.

In order to inherit the excellent protection features from virtualization technologies, whilst alleviating the side effects of performance and real-time features, related research on real-time virtualization and hardware-assisted virtualization is reviewed in the second part of Section 2.4. Some approaches provide excellent predictability, e.g. RT-Xen; some approaches can also alleviate reduction in performance features. However,

Currently existing approaches do not provide timing-accuracy, enhanced I/O performance and scalability (they only alleviate the reduction in performance features).

In order to achieve better timing-accuracy and predictability on I/O operations, Section 2.5 reviews two programmable timely I/O controllers, PRU and TPU. Because both PRU and TPU enable pre-programming, the transmission latencies between user programs and I/O devices can be removed. Nonetheless,

These I/O controllers cannot guarantee that an I/O operation occurs at a specific time in the future, i.e. they are not absolutely timing-accurate. Moreover, both I/O controllers require two 32-bit RISC cores, which generate additional hardware overhead, i.e. they are not resource efficient. Also, the number of RISC cores determines the number of I/O devices which can be operated in parallel, i.e. there is poor scalability and parallel access

Section 2.6 can be divided into two parts. Specifically, in the first part, the histories of ASICs and FPGAs were introduced, followed by the analysis

of their advantages and disadvantages. In the second part, a comparison between ASICs and FPGAs was presented, including their design flows, which demonstrated how to make hardware designs generic between ASICs and FPGAs. The real-time I/O system proposed in this thesis is mainly designed and implemented in front-end hardware design. Therefore, the prototype system can be easily ported between FPGAs and ASICs.

Chapter 3

Real-time I/O System

In the thesis, we assume that performance features, real-time features and protection features are simultaneously required by user applications. Although our designed real-time I/O system is architecture-agnostic, we assume that applications are implemented on a multi-core system, specifically an embedded Network-on-Chip (NoC) to enable the timing accuracy of multiple I/O devices in parallel.

In standard computer and embedded architectures, an I/O system can be evaluated using multiple metrics, e.g. memory footprint and I/O throughput etc [72, 90]. This chapter presents the basic idea of the expected features in real-time I/O systems, as well as the corresponding evaluation metrics of performance features, real-time features and protection features. Moreover, the evaluated results, with regard to the expected features of the baseline systems, are also demonstrated.

For the purposes of the discussion in this thesis, we define the following terms in this way:

- *I/O request* — Sent directly from a user application. It could be a high-level abstracted command, which cannot be used directly on an I/O controller.
- *I/O instructions* — Can be used to control an I/O device controller directly.

3.1 Baseline Systems

In this thesis, all designs and evaluation experiments are implemented on Xilinx ZC706 and VC709 development boards [20]. The many-core system adopted in the thesis is a 2D mesh type open source NoC called BlueShell [95], implemented using Bluespec System Verilog (***BSV***) [4]. Further implementation is detailed in Appendix B. The number of processors (i.e. MicroBlaze [11]) in BlueShell can be scaled from 1 to 2, 4, 9, 16.... The RTOS running on each processor is FreeRTOS (kernel version, FreeRTOS v9.0.0). An example of a 4x5 size NoC with 16 processors is shown in Figure 3.1.

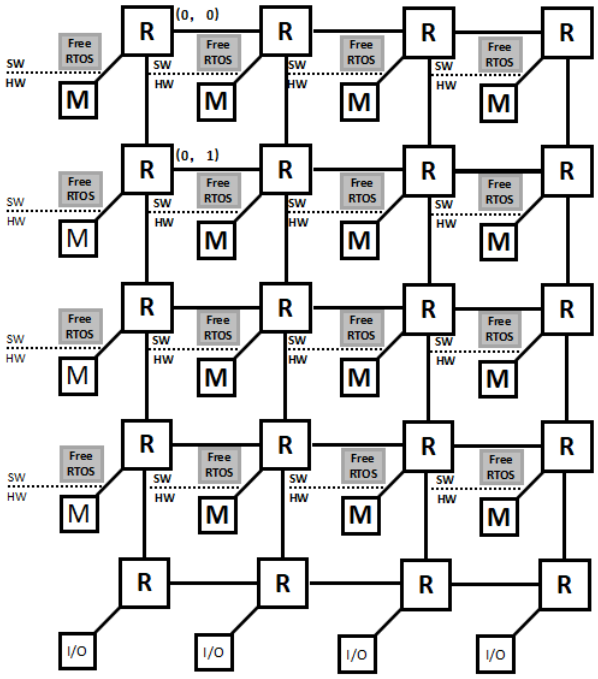


Figure 3.1: Example of a Baseline System
M - Microblaze R - Router

Note that, in the different evaluation experiments, the size of the NoC and the number of processors may vary. The evaluated I/O devices can be GPIO, SPI NOR-Flash, VGA or Ethernet, etc.

In the following sections, the basic concept of the six expected features in a real-time I/O system and the corresponding evaluation metrics are illustrated. The evaluated results of a corresponding baseline system are given and detailed information on the baseline system will also be shown in tabular form (see

Table 3.1: Baseline System Information

Evaluated Feature	Evaluated I/O Device	Size of NoC	Number of Processors	Display Method

Table 3.1).

3.2 Performance Features

In this thesis, the performance features of an I/O system include enhanced I/O performance and scalability.

3.2.1 I/O Performance

I/O performance is normally evaluated using I/O response time and I/O throughput. Specifically, if an I/O operation is requested by a user application at time T_r and completed by the I/O device at time T_c, the I/O response time R can be calculated as:

$$R = T_r - T_c \tag{3.1}$$

A smaller R implies a shorter I/O operation execution time, and therefore better I/O performance. The units of I/O response time (R) are normally seconds ($\boldsymbol{s}$), milliseconds ($\boldsymbol{ms}$), microseconds ($\boldsymbol{us}$) and nanoseconds ($\boldsymbol{ns}$).

The quantity of data (Q) handled by an I/O device in a specific time duration (τ) is measured by I/O throughput (C).

$$C = \frac{Q}{\tau} \tag{3.2}$$

A higher C means more data can be handled in a fixed time duration giving better I/O performance. The unit of I/O throughput is normally megabytes per second ($\boldsymbol{MB/s}$), kilobytes per millisecond ($\boldsymbol{KB/ms}$) and bytes per microsecond ($\boldsymbol{B/us}$), etc.

Therefore, a smaller I/O response time (R) and a larger I/O throughput (C) are required by a real-time I/O system.

Information on the baseline system used to evaluate the I/O performance is detailed in Table 3.2. The I/O response time and throughput in the proposed

real-time I/O system are compared with the baseline systems via tables and bar charts. The required enhancement to I/O performance is demonstrated by comparing the I/O response time and I/O throughput between baseline systems and proposed systems. Therefore, the I/O response time and I/O throughput in a baseline system are not shown in this chapter, rather, the comparison results are shown in Chapter 4, 6 and 7.

Table 3.2: Baseline System — I/O performance

Evaluated Feature	Evaluated I/O Device	Size of NoC	Number of Processors	Display Method
I/O Performance	SPI Flash	3x3	4	Bar Chart

3.2.2 Timing Scalability Model

The scalability of an I/O system in terms of timing can be considered by evaluating the average response time of an I/O device ($\overline{R}$) in a many-core system with a different number of processors. In the experimental systems, only one application is set to run on each processor, which continues to send I/O requests. The experiment settings aim to achieve the same context in different systems in different experiments, while the I/O achieves maximum throughput.

In theory, the optimal average I/O response time ($\overline{R_N}$) in a n-core system should be n times the average I/O response time, as in a single-core system ($\overline{R_1}$). Such a system would be timing scalable. In practice, the difference between the actual and optimal average I/O response times in an n-core system is regarded as the performance loss by the I/O system, termed ΔR:

$$\Delta R = \overline{R_N} - n * \overline{R_1} \tag{3.3}$$

The average I/O performance loss for each processor is calculated as Δr:

$$\Delta r = \frac{\overline{R_N} - n * \overline{R_1}}{n} \tag{3.4}$$

In a many-core system, if $\Delta r = 0$, it means no loss of I/O performance occurred, compared to a single-core system. Conversely, a larger Δr implies a reduction of I/O performance and reduced timing scalability of the evaluated I/O system.

Table 3.3: Baseline System — Timing Scalability

Evaluated Feature	Evaluated I/O Device	Size of NoC	Number of Processors	Display Method
Timing Scalability	SPI Flash	4x4	1, 4, 9	Table

The timing scalability of an I/O system can be evaluated in existing single-core and many-core (NoC-based) architectures. (For baseline systems, see Table 3.3).

The average I/O response time of reading one byte of data from an SPI NOR-flash and the corresponding Δr in different architectures with different scheduling policies are shown in Table 3.4. (Further experiment design is described in the Section 6.4). It is clear that in traditional many-core systems (baseline systems), with the number of processors increased, Δr also increases drastically, which implies a significant reduction in I/O performance and poor scalability of the I/O system.

Therefore, the proposed I/O system is expected to have a smaller increment

Table 3.4: Timing Scalability Model in Single-core, 4-core and 9-core Baseline Systems (unit: clock cycle)

	Scheduling Policy: RR (Global)		Scheduling Policy: FIFO (Local)	
Processor Index	$\overline{R}$	Δr	$\overline{R}$	Δr
Single-core Baseline System				
(0,0)	513	0	408	0
4-core Baseline System				
(0,0)	9015	1750	2916	284
(0,1)	8995		2875	
(1,0)	9213		2638	
(1,1)	8985		2645	
9-core Baseline System				
(0,0)	36060	3535.8	9357	496.5
(0,1)	35860		8915	
(0,2)	36049		8415	
(1,0)	36237		8203	
(1,1)	36410		9748	
(1,2)	36576		7476	
(2,0)	36741		7467	
(2,1)	36930		7576	
(2,2)	37102		6121	

in Δr as the number of processors increases, compared to baseline systems.

3.3 Real-time Features

As described in Section 1.3, the real-time features of an I/O system contain predictability and timing-accuracy.

3.3.1 Predictability

In real-time systems, the predictability of a task requires the range of its execution time to be determined (in particular, the Worst-Case Execution Time (***WCET***)), which aims to show that the task is able to meet any timing constraints. In both academia and industry, the WCET is normally achieved in two ways, static analysis and measurement-based analysis. These two methods are introduced in Sections 2.1.2.1 and 2.1.2.2, respectively.

In this research, the measurement-based analysis is adopted to evaluate the predictability of an I/O system. In the analysis, an experiment measuring the I/O response time when reading an SPI NOR-flash was executed 1,000 times and the corresponding I/O response time (R) was recorded. The variance of the I/O response time across 1,000 executions of the experiment indicated the predictability of the I/O system, with a larger variance indicating worse predictability.

The predictability of an I/O system can be evaluated in existing single-core and many-core NoC-based architectures. (For baseline systems, see Table 3.5). In the evaluations, the I/O requests of the SPI NOR-flash read 1, 4, 8, 16 bytes of data. (Further experiment design is described in Section 5.4 and Section 4.3)

Table 3.5: Baseline System — Predictability

Evaluated Feature	Evaluated I/O Device	Size of NoC	Number of Processors	Display Method
Predictability	SPI Flash	4x4	9	Table

As shown in Table 3.6, the variation in I/O response time across 1,000 iterations of the experiment in a 9-core system can be increased to 284,142 clock cycles while requesting to read 16 bytes of data.

Therefore, a smaller variance is required by a real-time I/O system, giving better predictability compared to a baseline system.

Table 3.6: I/O Response Time in Baseline Systems (unit: Clock Cycles)

Written Bytes	Scheduling Policy: Local FIFO		Scheduling Policy: Global Round-Robin	
	Worst Case	Variation	Worst Case	Variation
1	9,357	1,541	65,885	59,736
4	58,844	7,061	327,813	286,733
8	936,166	98,026	4,555,159	3,823,104
16	3,702,565	284,142	17,345,151	15,475,355

3.3.2 Timing-accuracy Model

The error in the timing-accuracy of I/O operations is defined as the absolute time difference between the time at which an I/O operation is required (T_r) and the actual time that the I/O operation (e.g. read) occurs (T_a):

$$E = T_r - T_a \tag{3.5}$$

Thus a smaller E implies a higher timing-accuracy of the I/O operation. If E equals 0, this I/O operation occurs at the expected time - i.e. totally timing-accurate. In practice, if E is less than one cycle period, then the I/O operation occurred at the required clock cycle.

The timing-accuracy that can be achieved in existing single-core and many-core architectures (for baseline systems, see Table 3.7) can be assessed by constructing a system on FPGA and measuring the effect of the latencies between the application and I/O device on the timing-accuracy of the I/O.

Table 3.7: Baseline System — Timing-accuracy

Evaluated Feature	Evaluated I/O Device	Size of NoC	Number of Processors	Display Method
Timing-accuracy	GPIO	4x4	9	Table

Errors found in 1,000 test runs for four systems are given in Table 3.8. (Further experiment design is described in Section 5.4.1). It is clear that, even in a single-core system, E is not close to a single cycle, with the timing error in multi-core and many-core systems considerably worse due to communication bottlenecks and contention of the system. With a VMM added, this issue is magnified further. Note that the experiment only measures hard-

ware latencies (across buses/NoC meshes) of I/O instructions issued by the application CPU. Therefore, clearly software effects (control/data flow within code), scheduling (between competing software tasks), real-time OS system calls and the implementation of I/O virtualization would add considerably to the overall latencies in Table 3.8.

Therefore, a certain WCET and none error in timing-accuracy ($E = 0$) are required by the I/O system — real-time features.

Table 3.8: The Errors in Timing-accuracy of I/O Operations in Baseline Systems (unit: ns)

CPU Index	E			
	Minimum	Median	Mean	Maximum
Single-Core Architecture				
	2090.0	2090.0	2012.5	2100.00
Dual-core Architecture				
Core 0	2440.0	2480.0	2477.2	2500.0
Core 1	2446.0	2450.0	2470.0	2490.0
NoC-based Many-core Architecture (Scheduling Policy: Local FIFO)				
(0,0)	3140.0	3140.0	3145.8	3160.0
(0,1)	3000.0	3000.0	3005.8	3020.0
(0,2)	2790.0	2790.0	2795.8	2810.0
(1,0)	2720.0	2720.0	2725.8	2740.0
(1,1)	3070.0	3070.0	3075.8	3090.0
(1,2)	2860.0	2880.0	2899.4	2940.0
(2,0)	2580.0	2580.0	2585.8	2600.0
(2,1)	2650.0	2650.0	2655.8	2670.0
(2,2)	2860.0	2930.0	2902.2	2950.0
NoC-based Many-core Architecture (Scheduling Policy: Global RR)				
(0,0)	4220.0	4220.0	4045.6	4260.0
(0,1)	4000.0	4000.0	4010.2	4080.0
(0,2)	3800.0	3800.0	3890.8	3920.0
(1,0)	3780.0	3780.0	3802.2	3840.0
(1,1)	4070.0	4070.0	4078.8	4100.0
(1,2)	3860.0	3880.0	3920.0	4000.0
(2,0)	3620.0	3620.0	3670.8	3760.0
(2,1)	3710.0	3710.0	3715.2	3770.0
(2,2)	3860.0	3930.0	3940.2	3980.0

3.4 Protection Features

As described in Section 1.4, the protection features of an I/O system include parallel accesses and isolation.

3.4.1 Parallel Access

The feature of parallel access means different user applications request (an) I/O device(s) at the same time.

Parallel access of an I/O system is normally evaluated through the error of timing-accuracy of different I/O operations (E). For example, two user applications are both required to access I/O devices at time T_0. If the E of the I/O operations requested from the two user applications are both equal to 0, the I/O system enables the feature of parallel access.

3.4.2 Isolation

Isolation requires the independence of I/O operations, which means the I/O operations requested from a user application should never be affected or attacked by a side channel. Because isolation is hard to verify using experiments, a discussion around supporting mechanisms is given in Section 2.4.

Currently, the most widely adopted methodology for achieving both parallel access and isolation is virtualization technology. As demonstrated in [93], [90], [109] and [103], the features of parallel access and isolation are already well supported by virtualization technologies. Some related work is reviewed in Section 2.4.

3.5 Summary

In this chapter, we described the basic concepts of the features required by real-time I/O systems, I/O performance, scalability, predictability, timing-accuracy, parallel access and isolation, followed by the description of each feature. We also proposed evaluation metrics corresponding to the features and some measured results in traditional architectures – baseline systems.

From the evaluation results, we noticed that protection features (i.e. parallel access and isolation) are already fully supported by virtualization technology. However, performance features (i.e. enhanced I/O performance and scalability) and real-time features (i.e. predictability and timing-accuracy) of

I/O systems in traditional architectures suffer from significant effects when the number of processors/cores is increased. With virtualization technologies deployed, the reductions are magnified even further. This implies that supporting good performance features and real-time features for I/O systems in multi-core and many-core systems is difficult. Furthermore, the difficulty will be magnified even further, if virtualization technologies are employed (for performance and real-time features).

Chapter 4

VCDC: The Virtualized Complicated Device Controller

As described in Chapter 1, Section 1.4 and 1.2, research question 3 is related to the research question 1. Specifically, research question 1 asks "*How can I/O performance in real-time systems be enhanced by an increased number of cores?*"; and part of the research question 3 is "*How can performance features and real-time features for I/O systems be achieved while I/O virtualization is deployed (to achieve protection features)?*" (Note that, the real-time features are also required by research question 3). The aims of this chapter are to examine research question 1, and the performance requirements of research question 3.

Specifically, I/O virtualization enables time and space multiplexing of I/O devices, by mapping multiple logical I/O devices upon a smaller number of physical devices. However, due to the existence of additional virtualization layers (i.e. VMM), requesting an I/O device from a guest virtual machine requires a complex sequence of operations. This leads to I/O performance loss, and makes precise and predictable timing of I/O operations problematic (the details are specifically introduced in Section 2.4.3).

This chapter proposes a hardware I/O virtualization system, termed the Virtualized Complicated Device Controller (***VCDC***). This I/O system allows user applications to access and operate I/O devices directly from guest VMs, and bypasses the guest OS, the Virtual Machine Monitor (VMM) and low layer

I/O drivers. We show that the VCDC efficiently reduces the software overhead and enhances the I/O performance (performance feature), predictability and timing-accuracy (real-time features). Furthermore, VCDC also exhibits good scalability that can handle I/O requests from a variable number of processors in a system.

This chapter has five sections. Specifically, Section 4.1 proposes an overview of VCDC, including some brief background, context, and high level design ideas. Section 4.2 introduces the specific design and implementations of VCDC. Afterwards, the evaluations on VCDC is demonstrated in Section 4.3. At last, the summary of this chapter is given in Section 4.4.

4.1 Overview

4.1.1 Background

In the last decade, virtualization technology has been widely used not only in server and desktop platforms, but also in embedded systems [102]. Using virtualization brings superior benefits for the whole system, including increased resource utilization, reduced volume and cost of hardware, and a better load balance in cores [102] [33] [111].

In real-time systems, the primary benefits offered by virtualization are parallel access and isolation (protection features). Specifically, guest virtual machines (VMs) are logically isolated, which means the applications executed in one guest VM can never affect the other virtual machines, even if it breaks down. The feature of isolation also brings significant support for the timing analysis of the tasks in a virtual machine [59]. Note that the isolation in virtualization technology can be split into temporal and spatial isolation. As mentioned in Section 2.4.1.2, the thesis mainly focuses on temporal isolation.

In real-time systems, the I/O performance is often a bottleneck of an I/O-bound system [44], which mainly results from the very slow processing speed of normal I/O devices compared to CPUs. This results in a performance reduction for the whole system. When it comes to multi-core and many-core systems, these issues are magnified, because of CPU scheduling and contention over I/O resources. For example, in a traditional bus-based multi-CPU system, if an I/O operation is requested by a user application, the system should deal with the scheduling of cores inside one CPU as well as the I/O resource scheduling among all the CPUs. These issues are magnified with virtualization

technology. When an application invokes an I/O request from a guest *Virtual Machine (VM)*, this I/O request will be transmitted via low layer drivers to the guest OS, *Virtual Machine Monitor (VMM)* and Host OS, which results in a serious loss of the system and I/O performance, see Figure 1.1.

Furthermore, virtualization technology can also impact the real-time features of an I/O system (specifically introduced in Section 2.4.3). Briefly, in a single-core system, latencies caused by device drivers and application process scheduling make predictable and timing-accurate I/O control problematic. In many-core systems, these issues are magnified: the transmission latencies from a processor to an I/O controller can be substantial and variable due to the communication bottlenecks and contention. These issues are magnified even further with virtualization technology. Virtualizing one physical I/O to multiple virtual I/Os, complex I/O resource management (e.g. scheduling and prioritization) and the complicated path of an I/O request worsen the transmission latencies from a processor to an I/O controller. Hence, it is difficult for an application from a guest VM to issue an I/O operation that will result in a timing-accurate device level I/O operation.

Virtualization relies on hardware support, therefore today's chip manufacturers have promoted different technologies for I/O virtualization in order to mitigate these issues. Intel's Virtualization Technology for Directed I/O (*VT-D*) [68], which can provide a direct I/O access from guest VMs, is one example. The IOMMU [39] is applied to commercial PC-based systems to offload memory protection and address translation, in order to provide a fast I/O access from guest VMs. However, even with hardware assistance, the I/O performance from the guest VMs cannot reach the original I/O performance in a system without virtualization, let alone improve on it. Achieving timing accuracy of I/O operations in a virtualized system, even with hardware support is difficult [122].

4.1.2 Design Idea

To overcome these issues, a hardware I/O system for multi-core and many-core systems was designed. The contribution of this chapter is the Virtualized Complicated Device Controller *VCDC*, which integrates the VMM and I/O drivers into the hardware layer, thus achieving significant improvements of I/O performance in guest VMs. The VMM in VCDC virtualizes a physical I/O device to multiple virtual I/O devices for guest VMs. For example, in

a 16-core system, the VMM can separate a single monitor into 16 individual partitions and provide access interfaces for each guest VM. In addition, the I/O drivers in VCDC provide high layer control interfaces for the guest VMs. With VCDC, the user applications in a guest VM are able to operate an I/O via very simple requests. Furthermore, if a user application is going to request the VGA controller to display a character from a guest VM, such as *'A'*, at coordinate *(2, 1)*, the user application is only required to transfer the ASCII of the character followed by its coordinates to the VGA part inside VCDC, that is *'0x41' ,'0x02', '0x01'*.

The VCDC utilises a timing-accurate I/O controller [120] to provide clock cycle level accurate I/O operations (more details, please see Chapter 6).

4.2 Virtualized Complicated Device Controller (VCDC)

Having presented the I/O problems suffered by virtualization technology in many-core and multi-core real-time systems, in this section we proceed by introducing our proposed Virtualized Complicated Device Controller (VCDC), which enables:

- *Better I/O performance* (performance feature) — Includes the lower response time of I/O operations and higher I/O throughput.
- *Scalability* (performance feature) — We propose a distributed implementation. When the VCDC is employed, to add one more CPU into a system, the user applications are only required to add one group of dedicated CPU FIFO, which aims to provide an interface between the added CPU and the VCDC.
- *Predictability* (real-time feature) — I/O operations requested from a guest OS are more predictable than under conventional virtualization.
- *Cycle level timing-accuracy* (real-time feature) — All I/O operations over the GPIO pins can be issued with an accuracy of a single cycle via being integrated with our clock cycle level timing-accurate I/O controller [120].
- *Lower software overhead* — Moves the VMM and low level I/O drivers from kernel mode (at the software level) to the VCDC.

- *Abstracted high layer access* — The user application in a guest virtual machine is able to request and operate an I/O device via invoking simple high layer drivers. For example, a user application can request to read a series of data from an SPI-Flash by sending a request with parameters to the VCDC: "*Read SPI-Flash* (instruction), from the *start address* to the *end address* (parameters)".

- *Global arbitration* — We propose a modularized implementation, whereby the scheduling policy of the arbiter can be switched easily between round robin, fixed priority and customized scheduling policies [21].

4.2.1 Virtualization in the VCDC Systems

VCDC provides I/O virtualization for guest VMs, such that a physical I/O device can be virtualized to multiple virtual I/Os for each virtual machine. In a system with VCDC, the I/O virtualization has the following features:

- *Bare-metal virtualization [102]* - Host OS is not required. A guest OS can be executed on a processor, directly.

- *Para-virtualization [77]* - The I/O management module of a guest OS should be replaced by our high layer I/O drivers, which can significantly reduce the software overhead.

The VCDC transforms each high level I/O request to single or multiple I/O instruction(s), that can be used on the physical I/O directly. For example, in our prototype implementation, a physical monitor (VGA controller) is virtualized into four sections. The screen of the monitor is separated into four sections by VCDC, which is used to display the content sent from each guest VM. In each VM, the initial coordinate of the (virtual) screen is (0, 0), which is respectively mapped to the following physical coordinates of the screen: (0, 0), (0, 100), (0, 200) and (0, 300). When a user application in the guest VM #3 sends an I/O request "*Display 'Hello World' at coordinate (0, 0)*", the VCDC will transform this request to "*Display 'Hello World' at coordinate (0, 300)*" and send corresponding instructions to the VGA controller.

4.2.2 Guest Virtual Machine and Guest OS

In our approach, each processor has an individual guest VM. As bare-metal virtualization is deployed (no host OS required), in each guest VM, a guest OS

is able to execute in kernel mode to achieve full functionality. Given that the VCDC provides part of the device driver, we also employ para-virtualization (modified OS kernel) to reduce software size, which we build using some high layer I/O drivers to replace the original I/O manager. Currently, we have provided three modified OS to support the I/O virtualization [21], which are FreeRTOS [7], ucosII [19] and Xilkernel [29]. In Figure 6.3, we use FreeRTOS as an example to illustrate the modification of a guest OS kernel in the VCDC systems.

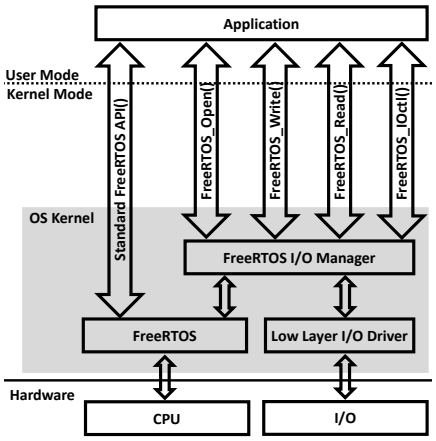


Figure 4.1: FreeRTOS Kernel in a non-VCDC systems

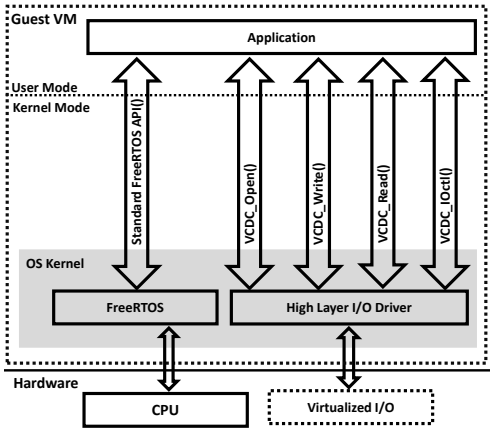


Figure 4.2: FreeRTOS Kernel in a VCDC system

Compared with the original FreeROTS kernel (Figure 4.1), the user application in a guest VM in VCDC system (Figure 4.2) is able to access and

operate I/O via the high layer I/O drivers, which are independent of the core module of the FreeRTOS.

Additionally, user applications running on the original FreeRTOS kernel can be ported to the modified kernel directly in a VCDC system (without any modification), since we have not modified the OS interfaces.

4.2.3 Overall Architecture

A typical use of the VCDC within a NoC architecture is shown in Figure 4.3 – all the I/O functions are performed by the VCDC rather than remotely by software.

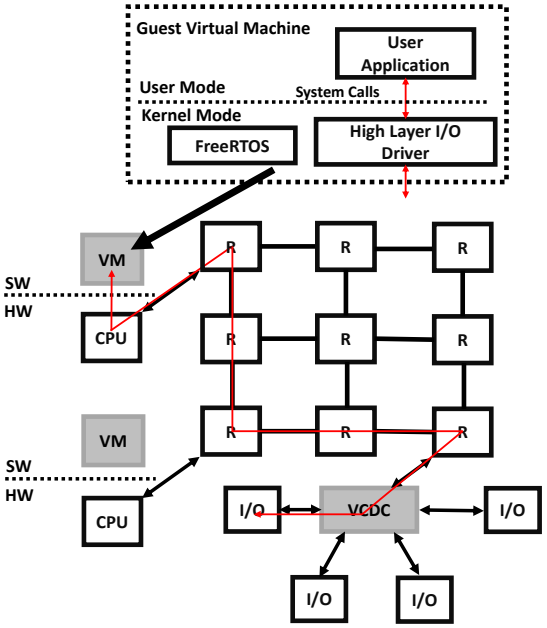


Figure 4.3: Overall architecture of a NoC with VCDC
VM - Virtual Machine; R - Router / Arbiter

At run-time, an application in a guest VM can invoke a high layer I/O driver on the VCDC to achieve the required I/O. The communications packets are transferred between the CPU and the VCDC via routers in the NoC. As an example, the path of such an I/O request message is shown in Figure 4.3 as a red line.

Note that use of a NoC is not required by VCDC — alternatively, a shared bus could be used. However, in the experiments presented in this thesis use a

NoC architecture.

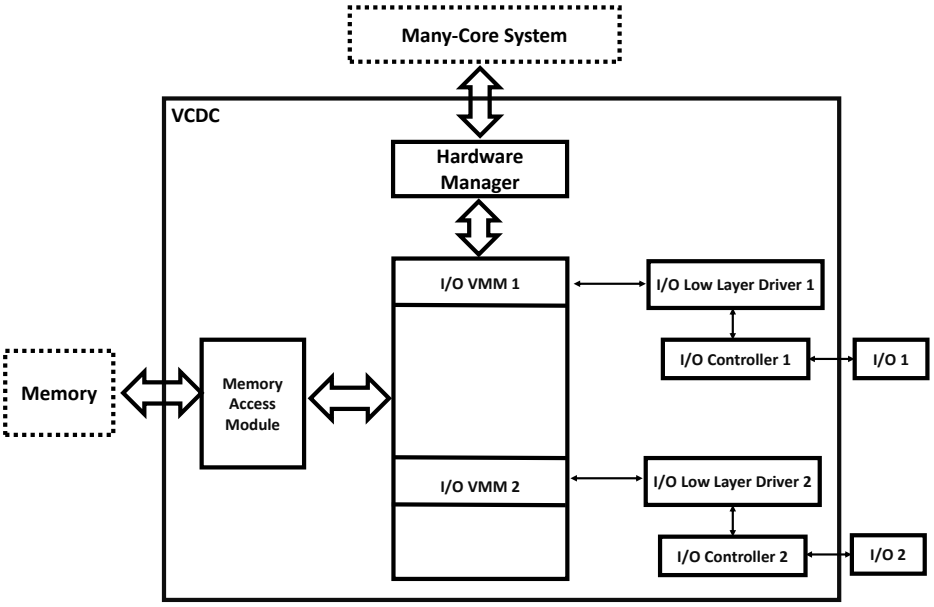


Figure 4.4: Architecture of VCDC

- *Hardware Manager* - Provides the interface to/from application CPUs via the NoC mesh.
- *I/O Virtual Machine Monitor (I/O VMM)* - Provides the functionality of virtualization for I/O devices.
- *I/O Low Layer Drivers* - Encapsulates the corresponding drivers of the specific I/O controllers (via I/O instructions).
- *I/O Controllers* - Controls the I/O devices, and can be driven by the low layer drivers directly.
- *Memory Access Module* - Provides the memory access interfaces for I/Os.

4.2.4 Detailed Architecture

These architectural elements are detailed in the following subsections.

4.2.4.1 Hardware Manager

The hardware manager is responsible for communicating with application CPUs, allocating incoming messages (I/O requests) from different CPUs to

corresponding I/O VMMs, as well as allocating response messages (I/O responses) from I/O VMMs back to CPUs. The architecture of the hardware is shown in Figure 4.5, with the right hand part allocating incoming requests from the NoC; and the left hand part taking ending data back to CPUs from VCDC.

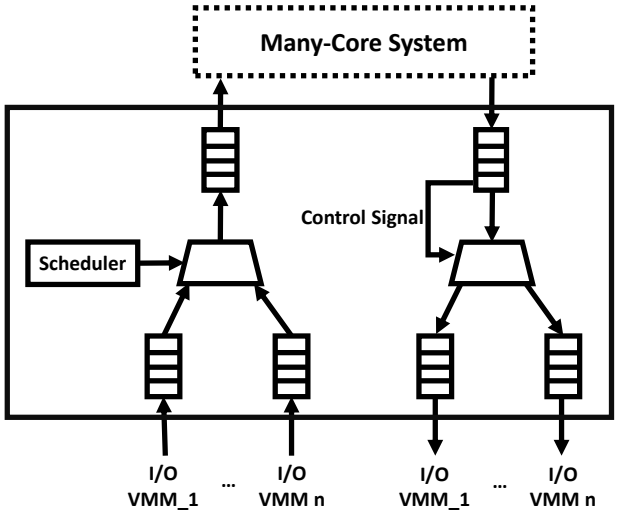


Figure 4.5: Architecture of Hardware Manager

The hardware manager is responsible for communicating with application CPUs, allocating incoming messages (I/O requests) from different CPUs to corresponding I/O VMMs, as well as allocating response messages (I/O responses) from I/O VMMs back to CPUs. The architecture of the hardware is shown in Figure 4.5, with the right hand part allocating incoming requests from the NoC; and the left hand part taking ending data back to CPUs from VCDC.

The right hand part of the hardware manager is mainly comprised of one input FIFO, a multiplexer and multiple output FIFOs (dependent on the number of I/O VMMs). The output FIFOs are connected to the different I/O VMMs. Similarly, the left hand part of the hardware manager is mainly comprised of multiple input FIFOs (dependent on the number of I/O VMMs), a multiplexer, an output FIFO and a scheduler. The input FIFOs are connected to the I/O VMMs, in order to receive the data to be sent back to the CPUs. The scheduler controls the multiplexer to choose which input FIFO can transmit data into the output FIFO (if neither input FIFO is empty the FIFOs are

chosen in a round-robin manner).

Additionally, the FIFOs used to connect with I/O VMMs can be connected to I/O controllers directly, which assists in supporting different I/O devices.

4.2.4.2 I/O VMM

I/O VMM maintains the virtualization of I/O devices. Considering that the functionalities and features of I/O devices are different, it is very difficult to build a general-purpose module to achieve virtualization for all kinds of I/O devices. Therefore, this thesis concentrates upon specific-purpose I/O VMM for commonly used I/O devices – eg. UART, VGA, DMA, Ethernet, etc. Users can also easily add their customized I/O VMM into VCDC via provided interfaces [21]. All of these I/O VMMs have a general architecture, see Figure 4.6.

The general architectures of the I/O VMMs are the same, except for the virtualization module. The I/O VMM is comprised of two groups of communication FIFOs, four multiplexers, two schedulers, groups of dedicated CPU

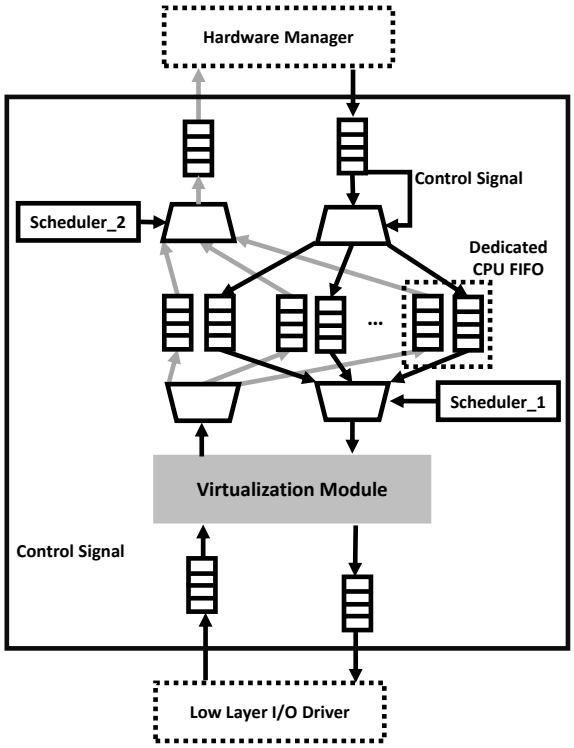


Figure 4.6: Architecture of Hardware Manager

FIFOs and a virtualization module.

The two groups of communication FIFOs are connected with the hardware manager and a low layer I/O driver respectively, providing the communication interfaces between the hardware manager and the low layer I/O drivers. The dedicated CPU FIFOs are built to store the I/O requests sent from different CPUs and I/O response messages sent back from the I/O (as buffers); one CPU owns an individual group of dedicated CPU FIFOs. The number of groups of dedicated FIFOs is generic, so that users can add any number (maximum to 64) of dedicated CPU FIFOs into the VCDC [21], which provides scalability. The two schedulers take charge of the scheduling of I/O requests and I/O response. Specifically, *Scheduler_1* determines which I/O request can be served by the virtualization module first, and *Scheduler_2* determines which I/O response can be sent back to the hardware manager first.

The virtualization module transforms I/O requests (sent to a virtual I/O) to I/O instructions (can be used to control a physical I/O). The implementation of this virtualization module depends on the specific I/O devices to be controlled. Currently, we have provided the virtualization modules for many commonly used I/O devices, including UART, VGA, DMA, Ethernet and an SPI NOR-flash. This thesis (including Section 4.3.3) focusses upon the virtualization module for Ethernet as an example I/O virtualization module.

4.2.4.3 Low Layer I/O Driver

Low layer I/O drivers take charge of encapsulating the specific I/O drivers for a specific I/O controller (shown in Figure 4.7). Users can also easily add their customized low layer I/O drivers via our provided interfaces [21]. We encapsulate the functions of I/O drivers into separate hardware modules, e.g. read the data from a specific address of the SPI NOR-flash.

As shown, a low layer I/O driver is comprised of two FIFOs (one input and one output), two multiplexers, one mutex and multiple functions of I/O drivers. Specifically, the input FIFO is responsible for receiving I/O instructions from I/O VMM, and the output FIFO takes charge of receiving I/O responses from the I/O controller. In order to guarantee that the low layer I/O driver is able to execute the I/O instructions in the same sequence as they are sent by the I/O VMM, a mutex is added. While instructions are being carried out by one of the hardware functions, other I/O instructions must be blocked to wait to access the I/O controller.

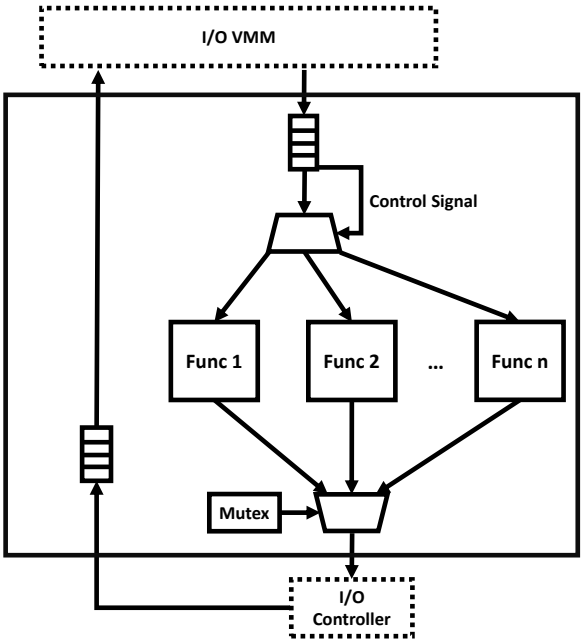


Figure 4.7: Architecture of I/O Low Layer Driver

4.2.4.4 Memory Access Module

VCDC also provides an interface to access the external memory (DDR), which is named BlueTree [62]. I/O devices are able to use this interface to read and write the external memory, such as the DMA. We will not introduce the implementation of the memory access module in this chapter; for more details please see [62], [60] and [61].

4.2.4.5 Timing-accurate Real-time I/O Controller

Clock cycle level timing-accurate I/O operations can be achieved by connecting the GPIO Command Processor (GPIOCP) [120] — see Chapter 5; and the integration of VCDC and GPIOCP is in Chapter 6.

4.3 Evaluation

The VCDC was implemented using Bluespec [4] and synthesised for the Xilinx VC709 development board [20] (further implementation details are given in Appendix A and B). The VCDC is connected to a 4 x 5 size 2D mesh type open source NoC [95] containing 16 Microblaze processors [11] running the

modified guest OS FreeRTOS (v9.0.0) in the guest VM. The modification of the FreeRTOS is described in Section 4.2.2. The architecture is shown in Figure 4.8.

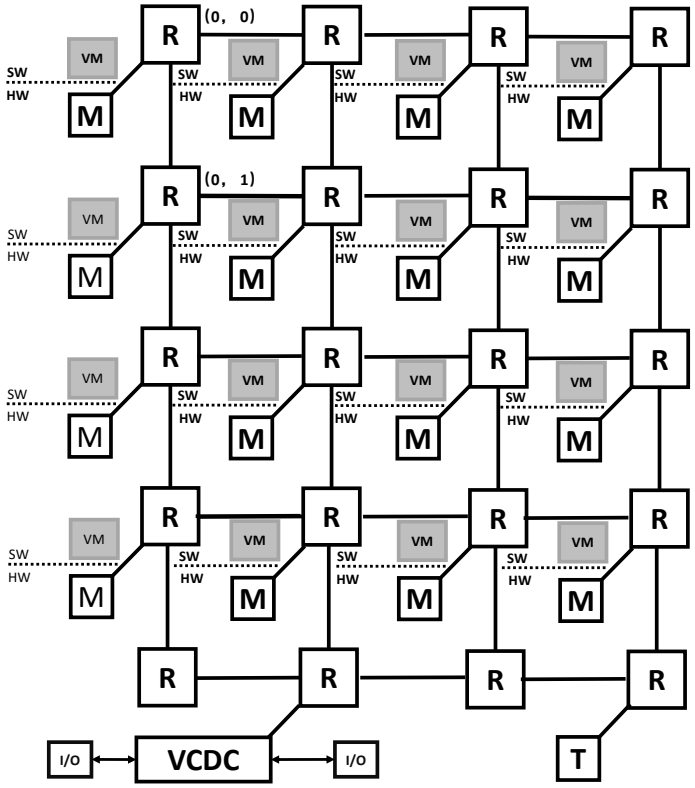


Figure 4.8: Experimental Platform
R - Router / Arbiter; M - Microblaze;
VM - Guest Virtual Machine; T - Timer

To enable comparison, a similar hardware architecture was built, but without the VCDC and I/O virtualization – note that this architecture requires I/O operations requested by Microblaze to pass through the mesh to the I/O rather than being controlled by a VCDC. The OS running on each Microblaze is FreeRTOS (v9.0.0) with its official I/O management module [6]. Note that, the non-VCDC system does not support any virtualization, instead, the I/O scheduling is achieved via the I/O manager in FreeRTOS. Both architectures run at 100 MHz.

4.3.1 Performance features: Response Time of I/O Operations

This experiment evaluates the performance of the I/O system whilst CPU and I/O are fully loaded in a VCDC and non-VCDC system. In both architectures, 9 CPUs are active, whose coordinates are from (0, 0) to (0, 2), (1, 0) to (1, 2) and (2, 0) to (2, 2). In both architectures, all the active CPUs have an independent application that is set to be running, which continuously reads data from an SPI NOR-flash (model: S25FL128S). Specifically, the experiments are divided into four groups, depending on the read bytes in each I/O request: 1, 4, 64 and 256. All the experiments are implemented 1000 times and recorded in tables. A lower I/O response time indicates a higher performance of the corresponding I/O system. We name the experiments according to the global scheduling policy and bytes of read data in one I/O request. For example, *non-VCDC-RR-4B* stands for a non-VCDC system with round-robin global scheduling policy; and 4 bytes of data read from the NOR-flash in one I/O request.

In the non-VCDC architecture, we modify the I/O management of FreeRTOS to be suitable for many-core systems[1]. While the user applications on different CPUs are requesting the I/O at the same time point, the scheduling policy can be set as FIFO (non-VCDC-FF) and Round-Robin (non-VCDC-RR) respectively.

Results of 1000 experiments are given in Table 4.1, showing that the response time of I/O requests in the non-VCDC architecture is significantly higher for the reading of 1 byte, 4 bytes, 64 bytes or 256 bytes from the NOR-flash, especially while Round-Robin scheduling policy being employed. For example, the average response time of non-VDCD-RR-1B is higher than $360,000$ ns ($36,000$ clock cycles). In contrast, in VDCD-1B, the worst I/O response time is lower than $4,000$ ns (400 clock cycles). The high I/O response time in non-VCDC-RR is mainly caused by the software implementation of round-robin I/O scheduling policy (complicated on-chip communication is required). In experiments with more bytes being read, the VCDC system maintains its superior performance. For example, in VCDC-256B, the I/O response time is lower than $900,000$ ns ($90,000$ clock cycles), which is similar to the worst case of the I/O response time in non-VCDC-RR-1B - $658,850$ ns ($65,885$ clock cycles).

[1]The I/O management in FreeRTOS is designed for a single-core system; in our experiments, we modify it to be suitable for many-core systems.

Table 4.1: I/O response time in VCDC and non-VCDC systems (unit: clock cycle)

	Non-VCDC System Scheduling Policy: FIFO			Non-VCDC System Scheduling Policy: RoundRobin			VCDC System		
CPU Index	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
Read 1 Byte									
(0, 0)	9357	9357	9357	6149	65885	36060	285	285	285
(0, 1)	7425	8989	8915	7073	65849	35860	380	403	396
(0, 2)	7057	8598	8415	7096	65849	36049	380	403	395
(1, 0)	7057	8207	8203	7096	65826	36237	357	403	391
(1, 1)	9748	9748	9748	7073	65826	36410	403	403	403
(1, 2)	7425	8966	7476	7073	65826	36576	334	334	334
(2, 0)	7034	8598	7467	7073	65826	36741	357	403	366
(2, 1)	7057	8207	7576	7096	65826	36930	357	403	377
(2, 2)	6121	6121	6121	7073	65803	37102	334	334	334
Read 4 Bytes									
(0, 0)	58002	58477	58021	29515	316248	173091	1066	1123	1093
(0, 1)	29611	36281	34908	33243	309490	168542	1247	1408	1356
(0, 2)	29657	37017	36191	34770	322660	176642	1293	1569	1398
(1, 0)	28875	36258	35264	34770	322547	177561	1362	1569	1412
(1, 1)	58361	58844	58381	33243	309382	171130	1316	1385	1325
(1, 2)	29588	35499	30208	34657	322547	179222	1247	1408	1270
(2, 0)	29979	37040	31290	35223	327813	182972	1247	1569	1322
(2, 1)	28139	36235	34785	32641	302799	169881	1293	1431	1369
(2, 2)	57579	58062	57599	32535	302693	170670	1247	1270	1249
Read 64 Bytes									
(0, 0)	907744	929955	918905	408908	4381352	2398035	18770	19245	18935
(0, 1)	450935	478696	460279	393536	4216640	2307883	19007	20272	19521
(0, 2)	479501	579758	538170	476993	4426369	2423243	19053	22549	20808
(1, 0)	473268	571294	520525	476993	4424823	2435851	19145	23032	21203
(1, 1)	909739	936166	921822	488037	4541994	2512343	19076	19398	19188
(1, 2)	449348	473636	456782	475305	4423507	2446804	19007	20157	19418
(2, 0)	474027	579068	535487	475305	4423507	2469029	19007	22043	20535
(2, 1)	472095	565429	518137	489451	4555159	2542512	19007	22549	20895
(2, 2)	900332	920618	907492	468232	4356158	2456170	19007	19237	19073
Read 256 Bytes									
(0 ,0)	3628902	3702565	3674076	1586442	16998330	9303655	75609	78231	76046
(0, 1)	1810819	1897023	1826232	1848174	17206343	9370227	75839	79841	77648
(0, 2)	1897828	2181970	2119170	1830492	17041721	9280577	75885	88305	83101
(1, 0)	1890399	2132060	2046512	1862700	17279325	9512215	75997	89708	84212
(1, 1)	3631085	3708365	3679649	1848508	17147673	9620444	75908	78484	76336
(1, 2)	1808220	1897000	1823103	1842516	17147673	9528055	75839	79542	77494
(2, 0)	1897391	2180659	2116159	1828370	17016021	9497681	75839	87040	82616
(2, 1)	1890422	2131301	2044241	1869796	17345151	9731236	75839	89202	83631
(2, 2)	3616296	3682191	3641053	1826248	16990334	9579806	75839	78346	76212

Additionally, the variance of I/O response time across 1000 experiments shows that VCDC systems have a better performance than the non-VCDC systems. For example, in the non-VCDC-FF-1B, the highest variance of I/O response time is greater than 15,000 ns (1,500 clock cycles). When it comes to the non-VCDC-RR-1B, the situation becomes worse: the highest variance of I/O response time reaches 600,000 ns (60,000 clock cycles). Conversely, in the VCDC-1B, the highest variance of I/O response time is less than 500 ns (50 clock cycles). For experiments with more bytes being read, VCDC systems still have a better performance. For example, in non-VCDC-RR-256B, the maximum variance of the I/O response time reaches 154,118,880 ns (15,411,888 clock cycles). Conversely, in VCDC-256B, the maximum variance of the I/O response time is only 137,310 ns (13,731 clock cycles), which is 1/1000 of the variance in the non-VCDC-RR-256B.

Therefore, the evaluation results show that a system with VCDC can provide more predictable I/O operations with lower response time.

4.3.2 Performance features: I/O Throughput

We evaluated the I/O throughput in two architectures (with VCDC and without VCDC). In the experiment, we use the same NOR-flash illustrated in Section 4.3.1 be connected to the VCDC as our evaluation object.

In both architectures, one independent application is set to be running on each of four Microblaze CPUs (coordinates are from (0,0) to (0,3)) and continuously writing to the NOR-flash - one byte can be written during one I/O request. We record the written bytes from each CPU within 1 second as the I/O throughput. The result of higher I/O throughput implies a better performance of the I/O system. All the evaluations are implemented 1000 times. The evaluation results are shown in Figure 4.9.

In the figure, four groups of bar charts present the average I/O throughput in the VCDC system and the non-VCDC system; and the error bar on each bar chart presents the variance of the I/O throughput in these 1000 experiments. As shown, on all CPUs considered, the VCDC system always provides a better performance on I/O throughput. Specifically, the I/O throughput from any of the CPUs in the VCDC system is nearly 7 times higher than the non-VCDC system with FIFO scheduling policy, and 20 times higher than the non-VCDC system with round-robin scheduling policy. Additionally, when it comes to the variance of I/O throughput, the VCDC system has a better

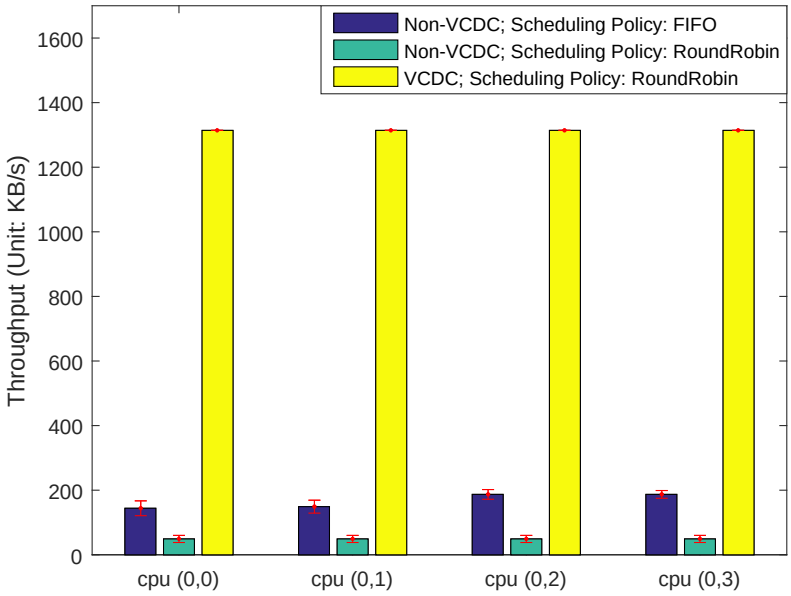


Figure 4.9: Performance feature: I/O Throughput
FIFO — Local FIFO; RoundRobin — Global RoundRobin

performance than the non-VCDC systems. Note that, the VCDC system with local FIFO scheduling has better I/O performance than the VCDC system with global round robin scheduling. This because the local FIFO scheduling can be executed on each CPU in parallel, and the global round robin can be only calculated via a processor independently.

In general, the evaluation results in this section show that a system with VCDC can provide higher I/O throughput with smaller variance than a non-VCDC system.

4.3.3 Performance Feature: Scalability

In this section, we evaluate the scalability of the VCDC by measuring the I/O response time of Ethernet packets sent from different CPUs in single-core, 4-core, 8-core and 16-core systems, respectively.

4.3.3.1 Ethernet Virtualization

A full Ethernet packet comprises an Ethernet header, an IP header, a TCP header and the payload [96]. The virtualization of Ethernet is implemented by virtualizing the IP address of Ethernet packets sent from each processor.

In a many-core or multi-core system, all the Ethernet packets sent from different CPUs should have the same IP address. In a system with VCDC, the virtualization module sets the last 8 bits of the source IP address as the CPU ID, so that the Ethernet packets sent from each CPU can have a unique source IP address. With VCDC employed, one CPU is able to communicate with a dedicated destination without interference from other CPUs.

In our approach, the VCDC connects with the Xilinx 1G/2.5G Ethernet subsystem [26], which comprises three IP cores: a Tri-mode Ethernet MAC (*TEMAC*) [110], a Gigabit MII (*GMII*) [110] and an AXI Ethernet buffer [27], see Figure 4.10.

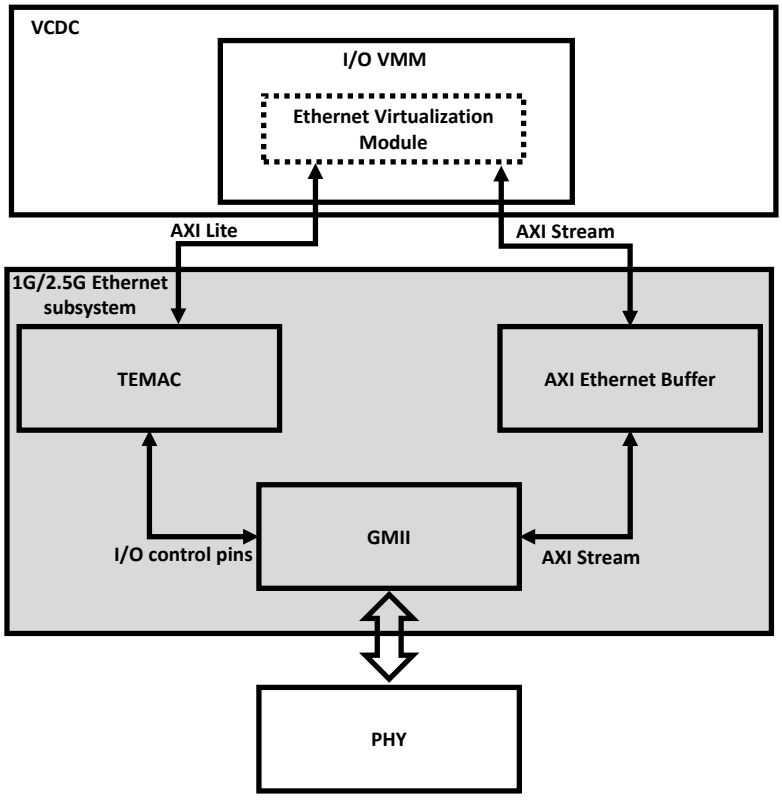


Figure 4.10: Connection between VCDC and Ethernet System

In Xilinx 1G/2.5G Ethernet subsystem, the GMII provides an interface between MAC and PHY, which is controlled by the TEMAC and the AXI Ethernet buffer. Specifically, the TEMAC takes charge of the control parts of the GMII, such as initialization and settings of communication speed. The AXI Ethernet buffer takes charge of transmission of Ethernet packets. When

an Ethernet packet is received by the AXI Ethernet buffer, the packet will be sent to the GMII directly via an AXI stream interface, then sent to the physical layer.

As described in Section 4.2.4.2, inside I/O VMM, the virtualization module is responsible for the virtualization of a specific I/O. Figure 4.11 describes the inner architecture of the virtualization module inside the I/O VMM for Ethernet.

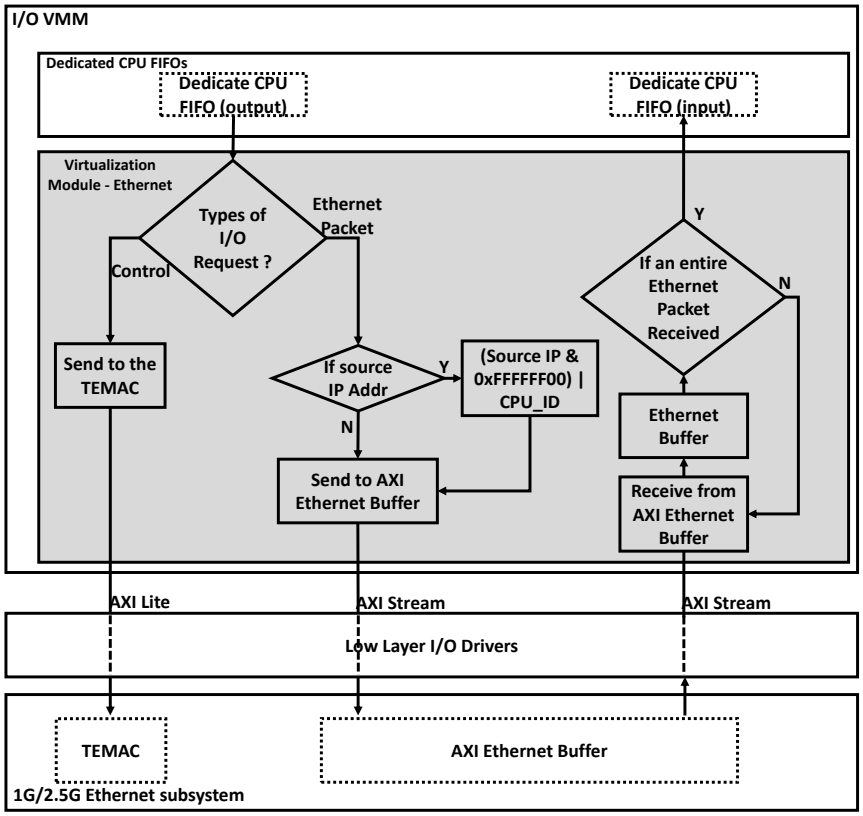


Figure 4.11: Virtualization Module of Ethernet I/O VMM

The virtualization module inside the Ethernet I/O VMM has two parts: down and up. The down part takes charge of the analysis and allocation of incoming I/O requests from the dedicated CPU FIFOs. Specifically, the I/O requests received by the virtualization module are divided into the control operations and the Ethernet packets. If the incoming I/O request is the control operation, the virtualization module will allocate it to the TEMAC inside the Ethernet subsystem via the low layer I/O drivers (AXI lite interface). If the incoming I/O request is an Ethernet packet, the virtualization module will

virtualize its IP address according to its corresponding CPU IP; and send it to the AXI Ethernet buffer via the low layer I/O drivers (AXI Stream interface). Additionally, the up part takes charge of receiving Ethernet packets from the physical layer (PHY). It buffers and sends an entire Ethernet packet back to the corresponding dedicated CPU FIFO according to the destination IP address of this Ethernet packet.

4.3.3.2 Experiment

The experiment is divided into two groups, dependent on the global scheduling policy of the VCDC: round-robin (named VCDC-RR) and fixed priority (named VCDC-FP). In VCDC-RR and VCDC-FP, the experiments can be further divided into four parts, according to the number of active CPUs. In these four parts of the experiments, we activate 1, 4, 8 and 16 Microblazes respectively. We name these experiment parts according to the label of the experiment plus the number of active CPUs. For example, in a 4-core VCDC system with round-robin global scheduling policy, the experiment is labelled VCDC-RR-4.

The software application running on each active CPU is the same, and is designed to continuously send 1 KB Ethernet packets via VCDC to a dedicated component. The 1 KB Ethernet packets sent from different CPUs are exactly the same, including the MAC header, the IP header, and the payload. However, the VCDC will virtualize the source IP address of each Ethernet packet based on the rules in Section 4.3.3.1. Additionally, the dedicated component is designed to monitor the response time of these Ethernet packets by recording the response time and analysing the virtual source IP address of the packets. All the experiments were implemented 1000 times; and the experiment results are depicted in tables.

In VCDC-FP, CPU (0, 0) is always set as the highest priority, followed by CPU (1, 0), (2, 0) , (3,0) and (1, 0) etc. The experiment results are shown in Table 6.8. As shown, for all multi-core systems, the I/O response time from the CPU with the highest priority is always fixed around 12 us; and the I/O requests from the CPUs with the lower priorities are always blocked by the the I/O requests with higher priorities, which guarantees the execution of the I/O requests with higher priorities. For example, in VCDC-FP-8, the average response time of the I/O requests from CPU (0,0) (the highest priority) is kept to 12 us, which means it can never be blocked by others. When it

Table 4.2: Average Response Time of Loop Back 1KB Ethernet Packets in VCDC System (Global Scheduling Policy: Fixed Priority; Unit: us)

	Number of CPUs			
CPU Index	1	4	8	16
(0, 0)	12.09	12.07	12.09	12.08
(1, 0)	-	25.50	25.51	25.50
(2, 0)	-	36.92	36.94	36.93
(3, 0)	-	48.35	48.36	48.35
(0, 1)	-	-	59.78	59.78
(1, 1)	-	-	71.21	71.19
(2, 1)	-	-	82.62	82.62
(3, 1)	-	-	94.06	95.06
(0, 2)	-	-	-	105.46
(1, 2)	-	-	-	116.90
(2, 2)	-	-	-	128.31
(3, 2)	-	-	-	139.74
(0, 3)	-	-	-	151.17
(1, 3)	-	-	-	162.58
(2, 3)	-	-	-	174.02
(3, 3)	-	-	-	185.44

comes to the I/O requests from CPU (3, 1) (the lowest priority), the I/O response time is always around 96 us, which is 8 times the highest priority I/O requests. The I/O response time of the lowest priority I/O request is extended due to blocking from other CPUs, which means that the VCDC system does not introduce an extra delay for the lowest priority I/O request. In an 8-core system, the theoretical optimal response time of the lowest priority I/O request should be 8 times the highest priority I/O request, and our experiment results obtain this. Similarly, in VCDC-FP-16, the average response time of the I/O request from CPU (3,3) (the lowest priority) is around 190 us, which is 16 times the response time of the highest priority I/O requests. The results still meet the theoretical optimal value. These experiments indicate a good scalability of the VCDC.

For VCDC-RR, the experimental results are shown in Table 6.9. As shown, with an increase in the number of CPUs, the I/O response time of each CPU

Table 4.3: Average Response Time of Loop Back 1KB Ethernet Packets in VCDC System (Global Scheduling Policy: Round Robin; Unit: us)

	Number of CPUs			
CPU Index	1	4	8	16
(0, 0)	12.32	46.71	90.58	180.15
(1, 0)	-	47.20	90.88	180.71
(2, 0)	-	47.68	91.22	179.99
(3, 0)	-	48.19	91.58	180.66
(0, 1)	-	-	91.93	180.04
(1, 1)	-	-	92.27	180.71
(2, 1)	-	-	92.63	180.09
(3, 1)	-	-	92.98	180.77
(0, 2)	-	-	-	180.04
(1, 2)	-	-	-	180.71
(2, 2)	-	-	-	180.09
(3, 2)	-	-	-	180.77
(0, 3)	-	-	-	180.04
(1, 3)	-	-	-	180.71
(2, 3)	-	-	-	180.09
(3, 3)	-	-	-	180.77

is proportional to the number of CPUs. Specifically, compared to the response time of an I/O request in VCDC-RR-1, the average I/O response time of an I/O request in VCDC-RR-4, VCDC-RR-4 and VCDC-RR-16 is respectively around 4, 8 and 16 times the average I/O response time in a single-core system. These results are close to the theoretical optimal values, which shows a good scalability of the VCDC.

4.3.4 Hardware and Software Overhead

This section can be mainly divided into two parts. In the first part, we compare the software overhead of a VCDC system and non-VCDC system with a software implementation of I/O management (i.e. I/O manager in FreeRTOS), see Table 4.4. In the second part, we compare the hardware overhead of a VCDC and a Microblaze CPU (running as a VMM), see Table 4.5.

4.3.4.1 Software Overhead

As shown in Table 4.4, the VCDC system significantly reduces the software overhead. Specifically, the software I/O manager is not required and the size of I/O drivers is smaller in the VCDC system.

Table 4.4: Software Usage(object code)

Software Module	VCDC	Non-VCDC (FIFO)	Non-VCDC (Round-Robin)
I/O Manager (KB)	0	139.2	148.5
UART Driver (KB)	60.5	122.4	122.4
VGA Driver (KB)	70.2	105.2	105.2
Non-Flash Driver (KB)	90.2	135.8	145.6
Ethernet Driver (KB)	88.7	210.2	230.2

4.3.4.2 Hardware Overhead

This section evaluates the hardware consumption of VCDC, which can be divided into two parts: 1) the comparison between VCDC and a commonly used SPI controller; 2) the comparison between VCDC and a full-featured Microblaze with same I/O functionalities (I/O drivers) installed.

Table 4.5: Hardware Usage (Without GPIOCP)

Hardware Consumption	VCDC	SPI Controller	Microblaze	
			FIFO	RR
Look Up Tables	4812	886	1860	1860
Registers	1413	615	2133	2133
Block RAMs (KB)	0	0	8	8
DDR3	309.8 KB	0	712.8 KB	751.9 KB

As shown in Table 4.5, compared with a dedicated I/O controller (SPI controller), VCDC consumes more FPGA hardware resources, including look

up tables and registers.

When compared with a full-featured Microblaze, the VCDC consumes extra look up tables but fewer registers and BRAMs. Moreover, compared to the memory sizes required by I/O drivers stored in DDR (detailed in Table 4.4), VCDC only consumes half DDR resources.

It is a trade-off between software overhead and hardware overhead. However, the VCDC system brings significant improvements of the I/O performance, including I/O throughput, response time, variance and scalability.

4.3.5 On-chip Communication Overhead

In NoC-based many-core systems, all the I/O requests are transmitted as on-chip packets. A larger requirement for on-chip packets means a higher on-chip communication overhead. In this section, we compare the on-chip communication overhead while invoking commonly used I/O requests in a VCDC and non-VCDC system by recording the number of packets on the NoC. In the NoC [95], the width of all the on-chip packets is 32 bits. The evaluation results are demonstrated in Table 6.10. The table shows that whilst the invoked I/O request is simple (e.g. displaying one pixel via the VGA in a single-core system), the on-chip communication overhead is similar in all systems. When the I/O operations become complex or the number of CPUs is increased, the on-chip communication overhead in non-VCDC architecture is significant; in contrast, the VCDC architecture has a lower on-chip communication overhead, for example, reading 10 bytes data from the SPI flash in 10-core systems.

4.3.5.1 Bottleneck of On-chip Communication

In the VCDC a single channel interface is used for transmitting VCDC requests. It connects the many-core system and the VCDC (see Section 4.2.4.1). Frequently invoked VCDC requests might cause traffic congestion at the interface of the VCDC, which decreases the predictability of I/O operations. This traffic congestion can further affect communication issues at the system level.

4.3.5.2 Discussion

The current implementation assumes that the number of communication channels in the interface between the many-core system and VCDC can be increased (relatively easy on an FPGA implementation). Then, multiple communication

Table 4.6: On-chip Communication Overhead

I/O Device	I/O Operation		Number of on-chip Packets (Each Packet: 32-bit)		
			Non-VCDC FIFO	Non-VCDC Round-Robin	VCDC
VGA	Display 1 Pixel	1 CPU	6	6	3
		4 CPUs	24	33	12
		10 CPUs	60	87	30
	Display 10 Pixels	1 CPU	60	60	30
		4 CPUs	240	357	120
		10 CPUs	600	897	300
SPI Flash	Read 1 Byte	1 CPU	12	12	4
		4 CPUs	48	57	16
		10 CPUs	120	237	40
	Read 10 Bytes	1 CPU	120	120	40
		4 CPUs	480	597	160
		10 CPUs	1200	1497	400

channels can alleviate communication traffic significantly. However, changing the number of communication channels requires a rebuild of the whole hardware, which is not suitable for a ready-built IC.

4.4 Summary

In this chapter, we have presented the concept of predictable hardware I/O virtualization for multi-core and many-core systems — the Virtualized Complicated Device Controller (VCDC). It enables applications to access and operate I/O devices directly from guest VMs, bypassing the guest OS, the VMM and low layer I/O drivers in software layer.

Evaluation reveals that VCDC can virtualize a physical I/O to multiple virtual I/Os with significant performance improvements, including faster I/O response time, greater I/O throughput, less on-chip communication overhead and good scalability. When it comes to the system overhead, the VCDC represents a trade-off between software and hardware, decreasing the software usage but requiring a greater consumption of hardware.

The contributions of the chapter are as follows. Firstly, Section 4.1 gives the overview of VCDC, including the backgroun, and design. Specifically, Section 4.1.1 introduces the benefits brought by I/O virtualization (i.e. protection features) and then presents the conflicts between I/O virtualization and performance features. Afterwards, Section 4.1.2 introduces the main design idea

of the VCDC — the integration of the VMM and I/O drivers into the hardware layer, enabling applications to access and operate I/O devices directly from guest VMs, bypassing the guest OS, the VMM and low layer I/O drivers in software layer, thus achieving significant improvements of I/O performance in guest VMs.

Secondly, Section 4.2 presents the specific design and implementation details of VCDC, including the high level designs (see Figure 4.3 and 4.4) as well as the detailed designs of internal components — hardware manager, I/O VMM, low layer I/O drivers, memory access module and timing-accurate real-time I/O controller (see Section 4.2.4.1 to 4.2.4.5).

Finally, Section 4.3 evaluates the performance features (both I/O performance and scalability), hardware overhead and on-chip communication overhead of VCDC. Specifically, Section 4.3.1 and 4.3.2 evaluate the I/O performance via I/O throughput and I/O response time (see Chapter 3, Section 3.2). The evaluation results reveal that VCDC significantly enhances the I/O performance compared to a non-VCDC architecture — increased I/O throughput and reduced I/O response time. Section 4.3.3 evaluates the scalability via measuring the I/O response time of a VCDC architecture with a different number of processors. As shown in the evaluation results, the VCDC system achieves better scalability compared to a non-VCDC architecture. Moreover, Section 4.3.4 evaluates the overhead related to VCDC — significantly reduced software overhead, but extra hardware overhead. Finally, Section 4.3.5 demonstrates the significantly reduced on-chip communication overhead of VCDC, compared to a non-VCDC system.

Chapter 5

GPIOCP: Timing-Accurate Real-time I/O Controller

The main aim of this chapter is to solve the second research problem: " *Apart from performance features, how can the predictability and timing-accuracy of I/O operations in multi-core and many-core real-time systems be guaranteed?*" (see Chapter 1, Section 1.3 and Chapter 2, Section 2.7).

Specifically, modern SoC/NoC chips often provide General-Purpose I/O (***GPIO***) pins for connecting devices that are not directly integrated within the chip. Predictable and timing-accurate control of devices connected to GPIO is often required within embedded real-time systems — I/O operations should occur at exact times, with minimal error, neither being significantly early or late. This is difficult to achieve due to the latencies and contentions present in architecture, between processor instigating the I/O operation, and the device connected to the GPIO — software drivers, OS, buses and bus contentions all introduce significant variable latencies before the command reaches the device. This is compounded in NoC devices utilising a mesh interconnect between processors and I/O devices.

The contribution of this chapter is a resource efficient programmable I/O controller, termed the GPIO Command Processor (***GPIOCP***), that permits applications to instigate complex sequences of I/O operations at an exact time, so achieving timing-accuracy at a single clock cycle level — predictable and timing-accurate. Also, I/O operations can be programmed to occur at some point in the future, periodically, or reactively. The GPIOCP is a parallel I/O controller, supporting cycle level timing accuracy across several devices con-

nected to GPIO simultaneously. Moreover, the GPIOCP exploits the tradeoff between using a full sequential CPU to control each GPIO connected device, which achieves some timing accuracy at high resource cost; and poor timing-accuracy achieved where the application CPU controls the device remotely. The GPIOCP has efficient hardware cost compared to CPU approaches, with the additional benefits of total timing accuracy (CPU solutions do not provide this in general) and parallel control of many I/O devices.

This chapter has five sections. Specifically, Section 5.1 proposes an overview of GPIOCP, including some brief background, context, and high level design ideas. Section 5.2 introduces the specific design and implementations of GPIOCP, followed by its control commands in Section 5.3. Afterwards, the evaluations on GPIOCP are presented in Section 5.4. At last, the summary of this chapter is given in Section 5.5.

5.1 Overview

As introduced in Chapter 1, Section 1.3, in real-time systems, I/O operations often need to be both predictable and timing-accurate [104, 120], in order to assure a timely reaction when critical situations occur (e.g. the braking operation of a car always has to be handled within a hard deadline [44]), or when an accurate control over I/O devices is required (e.g. an automotive engine requires I/O timing accuracy to inject fuel at the optimal time [89]).

5.1.1 Context

The specific architectural context of this chapter is predictable and timing-accurate control of I/O devices that are off-chip, accessed via GPIO pins, potentially using some bus protocol over those pins. This is in contrast to devices that are integrated within a chip (ie. a SoC or NoC chip) — such integrated devices have their latencies and timing-accuracy largely fixed by the existing architecture.

5.1.2 Approach

GPIOCP is a resource efficient programmable I/O controller that permits applications to instigate complex sequences of I/O operations at an exact time, so achieving timing-accuracy of a single clock cycle and predictability. This is achieved by loading application specific programs into the GPIOCP,

which can be interpreted at run-time to generate the specified sequence of control signals over a set of General Purpose I/O (GPIO) pins, eg. for read / write. Applications then invoke a specific program at run-time by sending the GPIOCP commands such as *run command X at time Y* (where Y is some future time). This achieves predictability and cycle level timing-accuracy as the latencies of the bus or NoC are removed.

For example, a periodic read of a sensor value by an application can be achieved by loading the GPIOCP with an appropriate program, then at run-time the application issues a command such as *run command X at time Y and repeat with period Z* —- the values are read at exact times, with the latency of moving the data back to the application considered within that application's execution time.

The GPIOCP is a parallel multi-functional controller, supporting different I/O devices in parallel —- so can provide timing-accurate I/O for applications simultaneously — i.e. parallel accesses. The GPIOCP can also be reprogrammed at run-time to control an I/O device in a different way, or potentially allowing hot-swap of I/O devices (noting that the program needs to be moved to the controller, requiring that traffic to be included in any system timing analysis).

5.2 GPIO Command Processor (GPIOCP)

The GPIO Command Processor (***GPIOCP***) proposed within this chapter enables:

- *Predictability and timing-accuracy*: All I/O operations over the GPIO pins can be predictably issued with an accuracy of a single cycle. Note that, the output values among GPIO pins can be constant or dynamic.

- *Programmability*: The GPIOCP holds small programs designed to control connected devices. They are loaded into GPIOCP memory by the application during system initialisation (so that loading does not interfere with normal execution and timeliness of the system). Importantly, commands within the program can be executed at exact times (cf. conventional CPU instructions).

- *Control of multiple connected devices in parallel*: Multiple I/O devices connected to the GPIO pins can be controlled in parallel, whilst main-

taining predictability and timing-accuracy of a single cycle.

A typical use of the GPIOCP within a NoC architecture is shown in Figure 5.1 — low level driving of the I/O device is performed by the GPIOCP rather than remotely by the application. At run-time an application can invoke a command program on the GPIOCP to achieve required I/O. This can execute immediately or at some time in the future; can be periodic; and can return data to the application CPU.

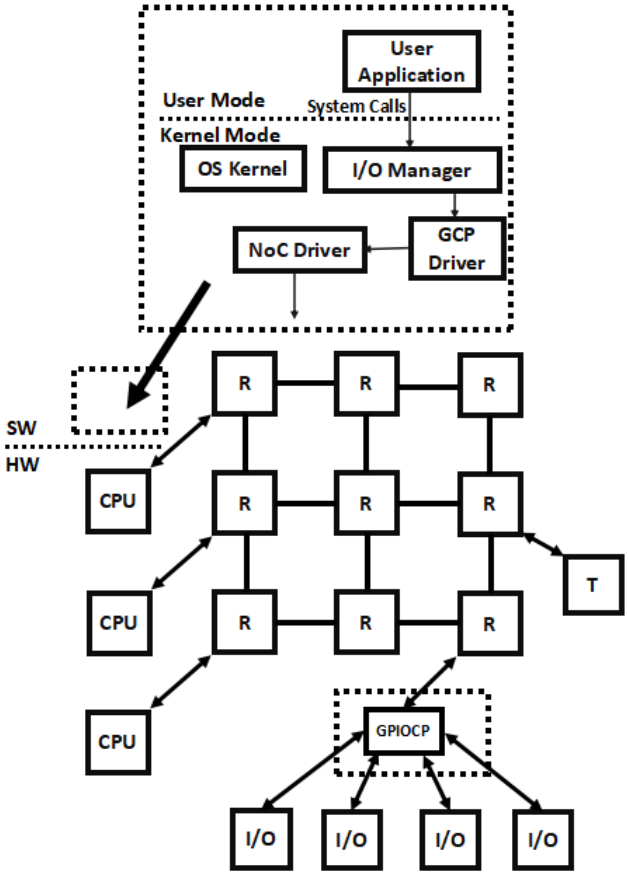


Figure 5.1: GPIOCP Connected to a NoC
(R - Router / Arbiter; T - Global Timer)

The architecture of the GPIOCP consists of the following main parts (see Figure 5.2):

- *Hardware manager*: Provides the interface to/from application CPUs via the NoC mesh.

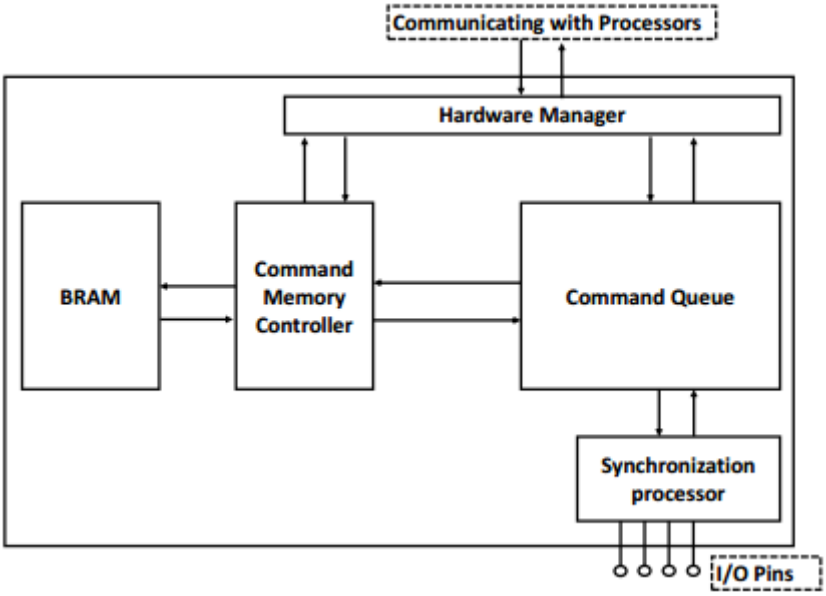


Figure 5.2: Architecture of GPIOCP

- *Command memory controller*: Manages internal GPIOCP memory to store/retrieve commands and data.
- *Command queue*: Manages GPIO CPUs which execute commands.
- *Synchronisation processor*: Provides synchronisation between the GPIO CPUs and external GPIO pins.

These architectural elements are detailed in the following subsections.

5.2.1 Hardware Manager

The hardware manager is responsible for communicating with application CPUs, allocating incoming messages to either the command memory controller (to store new commands) or the command queue (to initiate an existing command). The architecture of the hardware manager is shown in Figure 5.3, with the left part allocating incoming requests; the right part allows data to be sent from GPIOCP to CPUs.

The GPIOCP receives two forms of request:

- *Type 1*: Creating a new GPIO command — allocated to the output FIFO which is connected to the command memory controller.

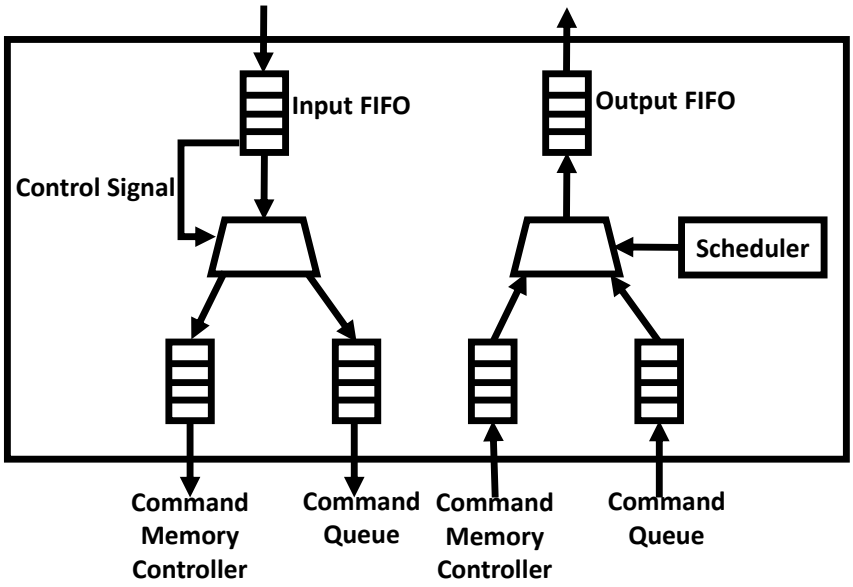


Figure 5.3: Architecture of Hardware Manager

- *Type 2*: Invoking a ready-built GPIO command Type 1 requests — allocated to the output FIFO connected to the command queue.

Similarly, the right-hand part of hardware manager (in Figure 5.3) is mainly comprised by two input FIFOs, a multiplexer, an output FIFO and a scheduler. The two input FIFOs are respectively connected to the command memory controller and the command queue, in order to receive the data to be sent back to the CPUs. The scheduler controls the multiplexer to choose which input FIFO can transmit data into the output FIFO (if both input FIFOs are not empty the FIFOs are chosen in a round-robin manner).

5.2.2 Command Memory Controller

The Command Memory Controller stores new GPIO command into the FPGA Block RAM (***BRAM***s); and accesses existing GPIO commands for execution by a GPIO CPU (within the command queue). The architecture of command memory controller is shown in Figure 5.4. Memory is divided into pages, with one GPIOCP command per page; each page containing command identifier (integer 4 bytes), command length (4 bytes) and the commands themselves (GPIO commands discussed in section 5.3), Eg. a 32KB BRAM can be split into 128 pages, each able to store identifier, length and up to 62 commands

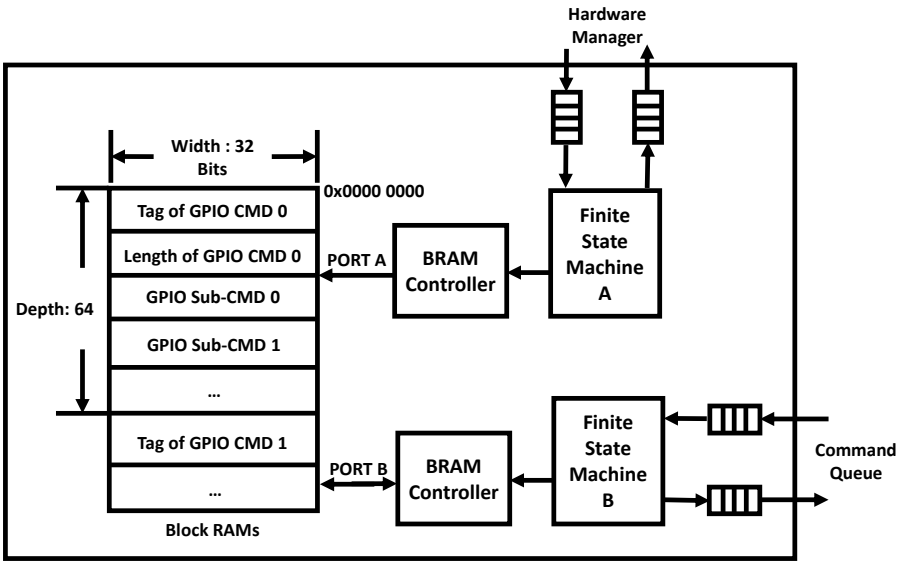


Figure 5.4: Architecture of Command Memory Controller

(each of 32 bits). Finally, the dual-ported nature of BRAM is exploited to provide separate interfaces to the command memory controller and command queue to improve performance.

5.2.3 Command Queue

The main functionality of command queue is allocating GPIOCP commands to GPIO CPUs for execution (architecture is shown in Figure 5.5). The command translation module requests commands from the internal memory via the command memory controller, sending the commands to a GPIO CPU for execution.

Each GPIO CPU is a simple finite state machine, with guaranteed execution time so achieving timing-accuracy. Each GPIO CPU has a dedicated I/O status cache (4 bytes), which only stores the status of I/O pins belonged to this GPIO CPU. This dedicated cache synchronises its I/O status with a globally shared register at a fixed frequency. The status of all I/O pins are stored in this shared register. Meanwhile, a register bank is also built in each GPIO CPU, which is used to achieve the 8th and 10th GPIO subcommand — delay for a specific time or set the group of I/O pins, equalling to the value stored in a register.

Moreover, a global timer is connected to all GPIO CPUs so providing time

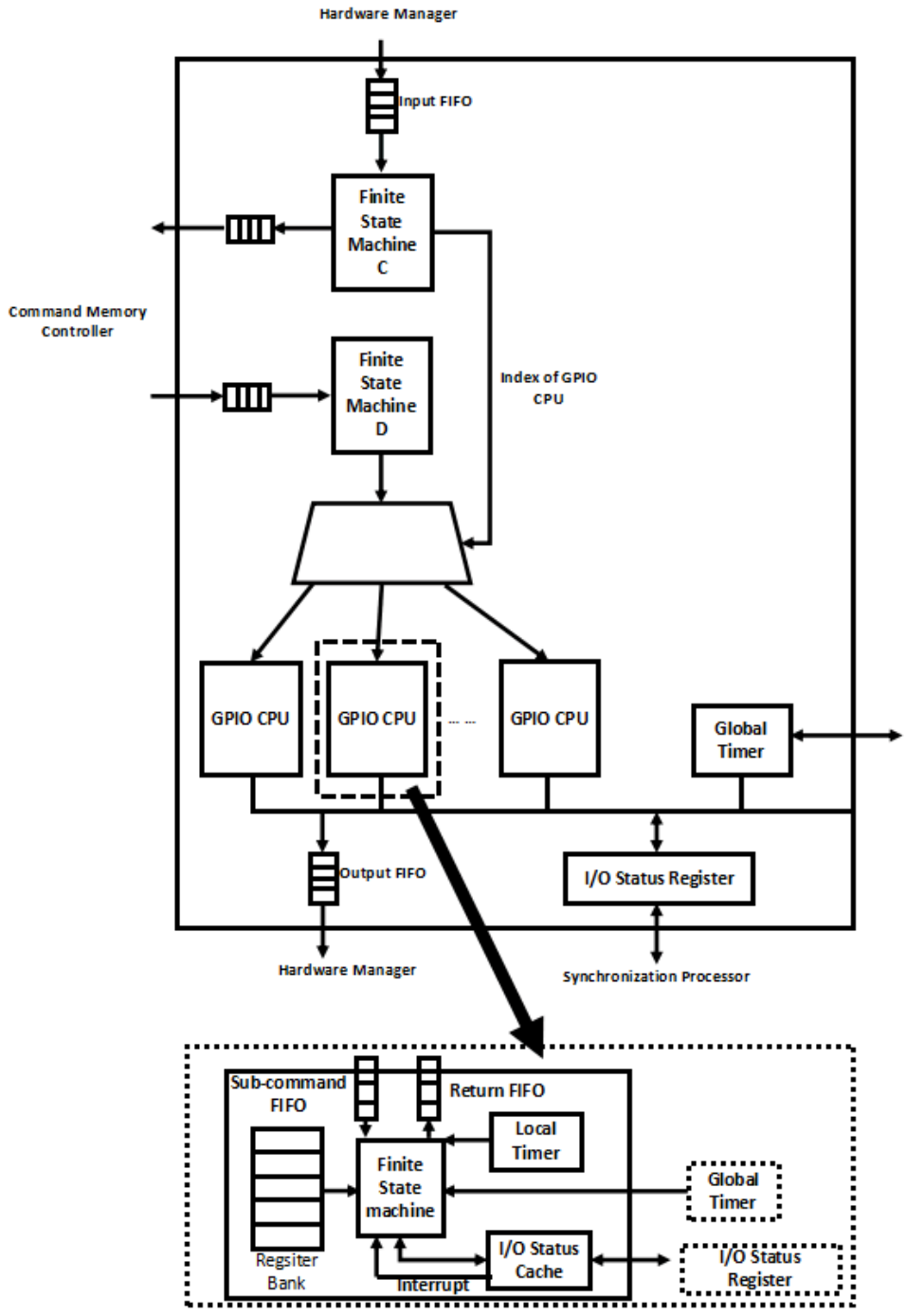


Figure 5.5: Architecture of the GPIO Command Queue

synchronisation between different processors and system clock — e.g. if several GPIO CPUs all need to execute a command at time t, the global timer enables

this. However, the conversion between absolute time (I/O devices) and relative time (user application) requires additional drivers – normally built into the software layer.

A supervisor/arbiter is also built inside the command queue, whose responsibility is handling the conflicts between different I/O requests, e.g. timing and bus conflicts, etc.. Because, the supervisor/arbiter is not related to the functionality of GPIOCP, it is not contained in the architecture diagram.

5.2.4 Synchronisation Processor

The synchronisation processor takes charge of synchronising the values of I/O pins, which may be written by different GPIO CPUs and I/O devices. The architecture of the synchronisation processor is shown in Figure 5.6.

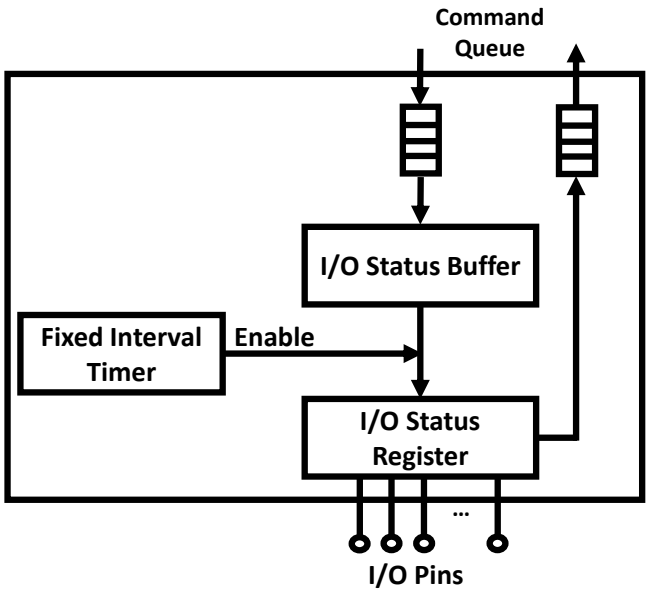


Figure 5.6: Architecture of Synchronization Processor

The modified values of I/O pins received from the input FIFO are stored in the I/O status buffer, rather than being updated immediately. A fixed interval timer enables the synchronisation between the I/O status buffer and the I/O status register every 5 clock cycles. Once the value of I/O status register changed, the changed value will be sent back to the command queue via the output FIFO. The timer owns the same frequency as the system clock.

5.3 GPIOCP Commands

A set of composable I/O control instructions, termed sub-commands are provided, consisting of I/O control subcommands, timing control sub-commands and a loop control sub-command. Application specific programs can thus be built from the sub-commands and stored in a page in GPIOCP internal memory (see Section 5.2.2)

GPIO write sub-commands supported are:

1) Execute the next write sub-command at a specific time.

2) Set a specific I/O pin to high/low.

3) Set a group of I/O pins to specific values.

4) Delay for a specified time (in clock cycles[1]).

GPIO read sub-commands supported are:

5) Execute the next reading sub-command at a specific time;

6) Read the value(s) of an specified I/O pin(s).

7) Read the value(s) of an specified I/O pin(s) while a predefined I/O pin triggered high/low.

The GPIO control sub-command supported is:

8) Delay for a specific time, stored in the register (in clock cycles).

9) Go to a specified GPIO sub-command.

The special GPIO write sub-commands supported is:

10) Set a group I/O pins to specific values, stored in the register.

The timing control sub-commands (1, 4, 5 and 8 above) provide timing constraints for I/O control sub-commands (2, 3, 6 and 7 above) which guarantee that an I/O device can be operated accurately at a given clock cycle — predictability and timing-accuracy. Running a subcommand 4 or sub-command 8 may take more than one clock cycle, but can be bounded by users. Any other seven sub-commands always use exactly 1 clock cycle. Therefore, the running time of GPIO commands is predictable, as they are comprised by GPIO sub-commands.

[1]In this thesis, all the I/O devices share a single synchronization clock source with the whole system. For example, if the frequency of the system clock is 100 MHz, the granularity of a clock cycle is 10 ns.

We define a GPIO sub-command as a 32 bit instruction, defined as follows (see Figure 5.7):

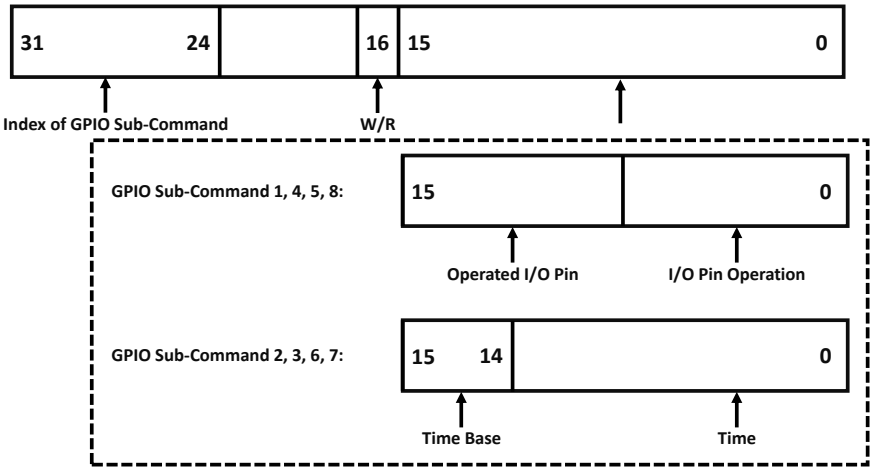


Figure 5.7: Format of GPIO Subcommand

- Bit 0 - Bit 15: Operation parameters of the GPIO subcommand, which allows for different additional information for different sub-commands. Specifically, in sub-commands 1, 4, 5 and 8, additional information regards timing; in subcommands 2, 3, 6 and 7, additional information regards the specific I/O pins; in sub-command 9, additional information includes the index of the sub-command to go to.
- Bit 16: The function type of GPIO sub-command read/write; '1' stands for writing function and '0' represents reading function.
- Bit 24 - Bit 31: The index of GPIO sub-command.

5.3.1 Example

To achieve a PWM signal on I/O pin #6 with 50% duty cycle the following sub-commands can be used:

1) Execute the next sub-command at time 200ns (i.e. delay until start of PWM signal): *0x010100C8*.

2) Pull I/O pin #6 high: *0x02010601*.

3) Wait for 20ms: *0x04018014*.

4) Pull I/O pin #6 low: *0x02010601*.

5) Wait for 20ms: *0x04018014*.

6) Go back to 2 and infinite loop: *0x090102FF*.

Note that, the output values and delay times both can be dynamic — set by the values in the registers. In order to make the example simple, constant values are adopted.

5.3.2 Invoking a GPIOCP Command

Requesting the GPIOCP to execute a command stored in a GPIOCP internal memory page requires the unique index of that command to be sent from the user application CPU to the GPIOCP. The format of the request is given in Figure 5.8:

- Bit 8 - Bit 15: The index of GPIO CPU which will execute this GPIO command.
- Bit 16: Read (0) / write (1).
- Bit 24 - Bit 31: The index of GPIO command.

For example, to execute command with index #2 on GPIOCP #3 would be: *0x02010300*.

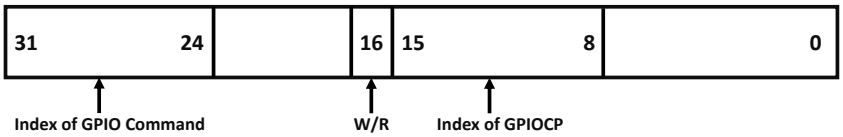


Figure 5.8: Format of GPIO Command

5.4 Evaluation

The GPIOCP is implemented using Bluespec System Verilog (***BSV***) [4], and synthesised for the Xilinx VC709 development board [20] (further implementation detailed in Appendix A and B). The GPIOCP is connected to a 4*3 size 2D mesh type open source NoC (BlueShell [95]) containing 9 Microblaze CPUs [10] running the uCosII RTOS (v1.41) [19].The architecture is shown in Figure 5.9.

To enable comparison, a similar hardware architecture was built, without the GPIOCP – note that this architecture requires I/O operations requested by CPUs to pass through the mesh to the GPIO rather than being controlled by a GPIOCP. Both architectures run at 100 MHz.

5.4.1 Real-time Performance

In this section, we evaluate the real-time performance (i.e. predictability and timing-accuracy) of GPIOCP, via the errors in timing-accuracy(E) (see Equation 3.1, in Section 3.2.1) and corresponding variances.

In the evaluation, when processors are required to access and read the GPIO at a specific time, then for a non-GPIOCP architecture the processors have to instigate the I/O operation, for the GPIOCP architecture, this can be delegated to the GPIOCP to achieve timing accuracy. This was measured by connecting a timer to the GPIO (updating its value every cycle), with every

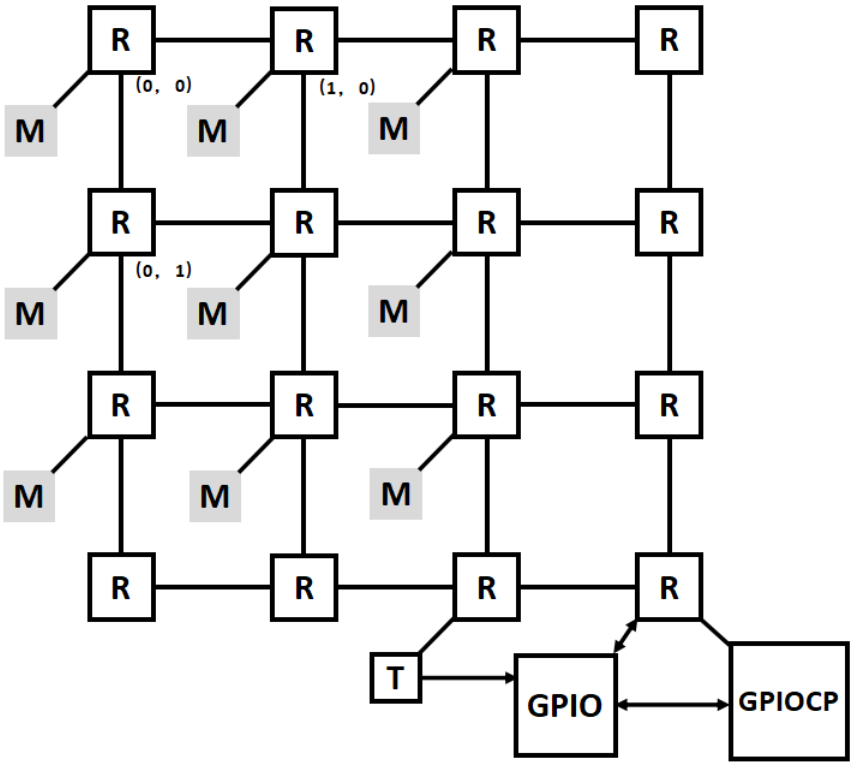


Figure 5.9: Experiment Platform
R - Router/Arbiter M - Microblaze T - Global Timer

CPU needing to read the value simultaneously. Results of 1000 experiments are given in Table 3.4 and 5.1, showing that the latencies and variance for the non-GPIOCP architecture (baseline system) are significant (errors calculated according to Equation 3.5); in contrast, the GPIO architecture is accurate at the cycle level.

Table 5.1: Errors in Timing-accuracy (E) in GPIOCP architecture

CPU Index	E (unit: ns)				E (unit: clock cycle)			
	Min	Med	Mean	Max	Min	Med	Mean	Max
(0, 0)	0.0	0.0	0.0	0.0	0	0	0	0
(0, 1)	0.0	0.0	0.0	0.0	0	0	0	0
(0, 2)	0.0	0.0	0.0	0.0	0	0	0	0
(1, 0)	0.0	0.0	0.0	0.0	0	0	0	0
(1, 1)	0.0	0.0	0.0	0.0	0	0	0	0
(1, 2)	0.0	0.0	0.0	0.0	0	0	0	0
(2, 0)	0.0	0.0	0.0	0.0	0	0	0	0
(2, 1)	0.0	0.0	0.0	0.0	0	0	0	0
(2, 2)	0.0	0.0	0.0	0.0	0	0	0	0

5.4.2 Hardware Overhead

The resource efficiency of the GPIOCP when implemented on the Xilinx VC709 FPGA development board is shown in Table 5.2. The GPIOCP is compared with FPGA (i.e. softcore) implementations of a processor and an SPI controller. The former enables comparison against approaches that use a dedicated processor as an I/O controllers, and a dedicated core for the specific I/O device. The GPIOCP utilises significantly less hardware than the processor, but more than the dedicated controller. Thus a parallel reprogrammable I/O controller can be achieved in less resource than a processor, and offers true timing accuracy across multiple GPIO connected external devices; but more resources than a dedicated controller that is useful for only one I/O device. Note that in this comparison, the GPIOCP is configured with 2 GPIO CPUs and 8 KB storage units.

However, even when GPIOCP works with a normal I/O controller (timing-accuracy is not required), the extra overhead can not be eliminated.

Table 5.2: FPGA Hardware Usage
L - Lookup Table, R - Register, B - BRAM

Microblaze Softcore Processro			SPI Softcore			GPIOCP 2 GPIO CPUs		
L	R	B	L	R	B	L	R	B
1170	1568	8	326	501	0	886	15	4

5.4.3 Case Study

The effectiveness of the GPIOCP approach is illustrated by considering the control of a 3D printer [13] which requires X and Y co-ordinates (via multiple motors) updating at a 5Mhz frequency. The printer is required to print out the following patterns:

(Note: This is in excess of usual 3D printer motor control frequencies but illustrates the effectiveness of the GPIOCP approach in that higher control frequencies are possible — offering potentially more accurate printing.)

- Pattern 1: $f_1(t) = 8$
- Pattern 2: $f_2(t) = t$
- Pattern 3: $f_3(t) = 128/t + sin(t) - t * cos(t)$
- Pattern 4: $f_4(t) = 80 * (sin(t))^5$
- Pattern 5: $f_5(t) = 128/t$
- Pattern 6: $f_6(t) = \sqrt{((14^2 - (t-14)^2) + 18)}$

Table 5.3: Deadline Miss Rate in Two Architectures

Pattern Index	Non-GPIOCP	GPIOCP
1	81.67%	0.00%
2	83.33%	0.00%
3	91.67%	0.00%
4	88.83%	0.00%
5	86.67%	0.00%
6	85.00%	0.00%

Table 5.4: Variances in Two Architectures

Pattern Index	Non-GPIOCP	GPIOCP
1	0.0000	0
2	2.8735	0
3	18.6886	0
4	25.7971	0
5	18.8541	0
6	3.4698	0

Control of the motors to draw the above patterns is required at a frequency of 5 MHz. No time was measured for the calculation of values by CPU, these were pre-calculated.

This was implemented with and without GPIOCP support. The non-GPIOCP implementation used a single processor accessing the GPIO directly; the GPIOCP implementation placed the entire control for generating the pattern in the GPIOCP as a single program.

To evaluate the patterns generated by the two implementations, the GPIO pins were monitored (by as separate core) that compared the generated values against expected values (stored as a pre-calculated table within the monitor). Table 5.3 and 5.4 respectively shows the miss-rate (values that were not written at the correct frequency) and variance (RMS error of output value compared with expected value at that time) – both are expressions of timing accuracy defined by Equation 3.1, in Section 3.2.1.

The non-GPIOCP implementation has a high miss rate, even for simple patterns (even the constant, pattern A), showing the latency of controlling GPIO from a processor. Where the pattern is simple, variance is low for the non-GPIOCP showing that if the pattern value does not change quickly, then outputting the wrong value (for the time) has less effect. However where the pattern varies more over time, variance increases. The GPIOCP implementation has zero miss rate and variances for all patterns – hence is timing-accurate and predictable.

5.5 Summary

In this chapter, the concept of a programmable I/O controller (GPIOCP) with a clock cycle level granularity and real-time features (i.e. predictability and timing-accuracy) has been presented. This enables application specific I/O control protocols, as well as operating multiple I/O devices in parallel with clock cycle level accuracy, all with timing accuracy appropriate to demanding real-time systems.

Evaluation reveals that GPIOCP can handle multiple I/O operations with clock cycle accuracy, in many cases totally timing accurate. However, the hardware overhead was 50% less compared to a testbed with the same functionality build using a minimalistic version of the soft core microprocessor; ie. using a Microblaze CPU instead of the GPIOCP. Therefore, compared to the timely I/O controllers reviewed in Chapter 2.5.3 (e.g. RPU & TPU), GPIOCP enables timing-accurate and predictable I/O operations with more flexibility. Meantime, GPIOCP consumes less hardware consumption (no CPU cores needed).

Summarising, section 5.1 discusses the GPIOCP, highlighting the real-time capability for I/O operations (i.e. predictability and timing-accuracy), followed by the introduction of the difficulties on achieving these features — transmission latencies and I/O contention. Section 5.1.1 introduced the system context of the GPIOCP, ie. even though the design of GPIOCP is architecture agnostic, it was connected to an embedded NoC. Section 5.1.2 introduced the main design of the GPIOCP: enabling programmability and permitting user applications to instigate complex sequences of I/O operations at an exact time, so achieving timing-accuracy of a single clock cycle and predictability. Then, Section 5.2 presented the specific design and implementation details of GPIOCP, including the high level designs (see Figure 5.1 and 5.2) as well as the detailed designs of internal components — hardware manager, command memory controller, command queue and synchronisation processor (see Section 5.2.1 to 5.2.4). Finally, Section 5.3 described the steps of programming GPIOCP and using GPIOCP to control I/O devices with real-time features (i.e. predictability and timing-accuracy) via invoking GPIOCP commands and GPIO sub-commands. Finally, section 5.4 evaluated the real-time features, and hardware overhead of GPIOCP. Specifically, Section 5.4.1 measured the error in timing-accuracy (E) of two architectures and corresponding variances (with and without GPIOCP). The evaluation results reveal that a GPIOCP-

based system can handle multiple I/O operations with clock cycle accuracy and predictability. Section 5.4.2 evaluated the resource efficiency of GPIOCP when implemented on the Xilinx VC709 FPGA development board. The evaluation results imply the hardware overhead of GPIOCP is 50% less compared to a tested minimalistic version of the soft core microprocessor (with the same functionalities).

Chapter 6

BlueIO: The Scalable Real-Time Hardware I/O Virtualization System

As described in Chapter 4, VCDC has provided significant improvements on performance features of I/O operations (i.e. I/O performance and scalability). Meanwhile, as introduced in Chapter 5, GPIOCP has solved the first research question, which enables real-time features on I/O operations (i.e. predictability and timing-accuracy). The main aim of this chapter is to solve the third research problem: "*How can performance features and real-time features for I/O systems be achieved when I/O virtualization is deployed (to achieve protection features)?*", see Chapter 1, Section 1.4.

In this chapter, we propose the design and implementation of ***BlueIO*** — which provides support for real-time I/O virtualization. We demonstrate how a BlueIO-based I/O virtualization system can be exploited to meet real-time requirements with significant improvements in I/O performance and a low running cost on different OSs. We also present a hardware consumption analysis of BlueIO, in order to show that it linearly scales with the number of CPUs and I/O devices, evidenced by our implementation which targets both FPGA and VLSI.

This chapter has five sections. Specifically, Section 6.1 proposes an overview of BlueIO, including general architecture, context and the I/O virtualization in the BlueIO-based system. Section 6.2 introduces the design and implementation of BlueIO. Afterwards, the hardware consumption analysis is demon-

strated in Section 6.3, followed by evaluations in Section 6.4. Finally, the summary of this chapter is given in Section 6.5.

6.1 Overview

The main design ethos of BlueIO is the integration of key functionalities of I/O virtualization, low layer I/O drivers (***VCDC***, see Chapter 4) and clock cycle level timing-accurate I/O control (***GPIOCP***, see Chapter 5) — all within the hardware layer, meanwhile providing abstracted high-layer access to software layers (Guest VMs). I/O virtualization provides isolation and parallel access to I/O operations. The hardware implementation of I/O virtualization offloads most or the overhead of virtualization into hardware, and enables the guest OSs to execute on ring 0 with full privilege. The hardware implemented low layer I/O drivers and the abstracted high layer access interfaces (VCDC) provides better I/O performance and scalability compared to baseline systems. The deployment of the GPIOCP guarantees the I/O operations will occur at a specific clock cycle (i.e. be timing-accurate and predictable).

6.1.1 General Architecture

Figure 6.1 depicts our proposed general embedded virtualization architecture. It can be seen, the RTOS kernel in each VM can be executed in kernel mode

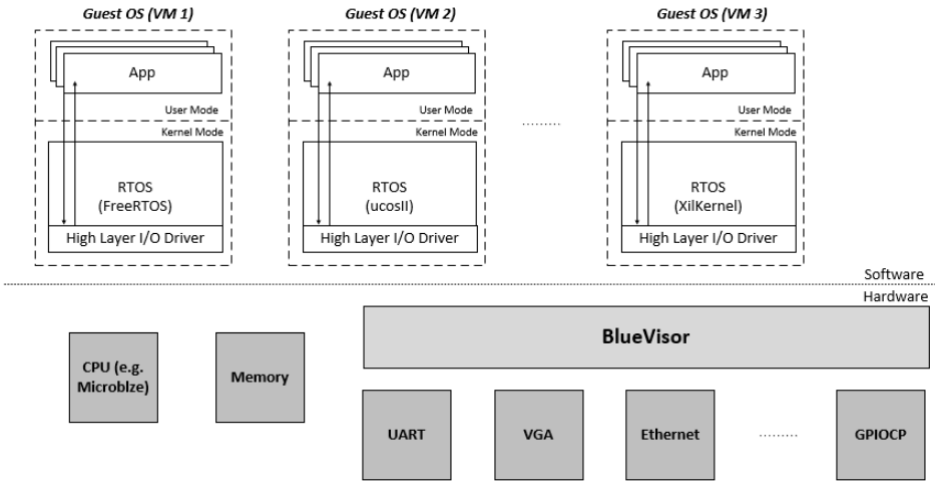


Figure 6.1: Embedded Virtualization Architecture

(ring 0) to achieve full functionality. Meanwhile, it can provide a real-time environment essential for the development of applications which need to guarantee specific deadlines. Finally, the I/O system, running in hardware, is responsible for I/O virtualization, physical isolation between VMs, and providing high layer access interfaces for user applications (in Guest VMs).

6.1.2 Context

In order to enhance the predictability of I/O requests, the BlueIO system is mounted to a 2D mesh type open source NoC, termed BlueTiles [95]. Use of a NoC is not required by BlueIO, because it is a general-purpose I/O system, which is agnostic to the type of bus and the software running on CPUs. To support a complete BlueIO system, the platform requires:

- Communication channels between BlueIO and CPUs;
- A global synchronization timer;
- A memory access interface (in the proposed design, BlueTree [62] is adopted as the memory access interface (see Section 6.2.4).)

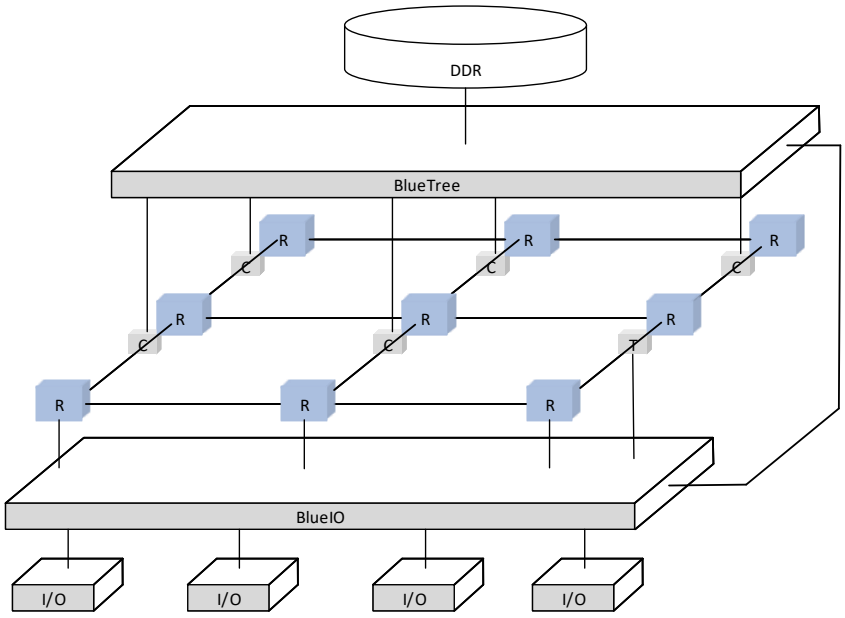


Figure 6.2: Platform Overview
C - Core; R - Router / Arbiter; T - Global Timer

The use of BlueIO within BlueTiles is shown in Figure 6.2. BlueIO is physically connected to the home port (via the physical link) of a router, the global timer T, and the memory access interface — BlueTree.

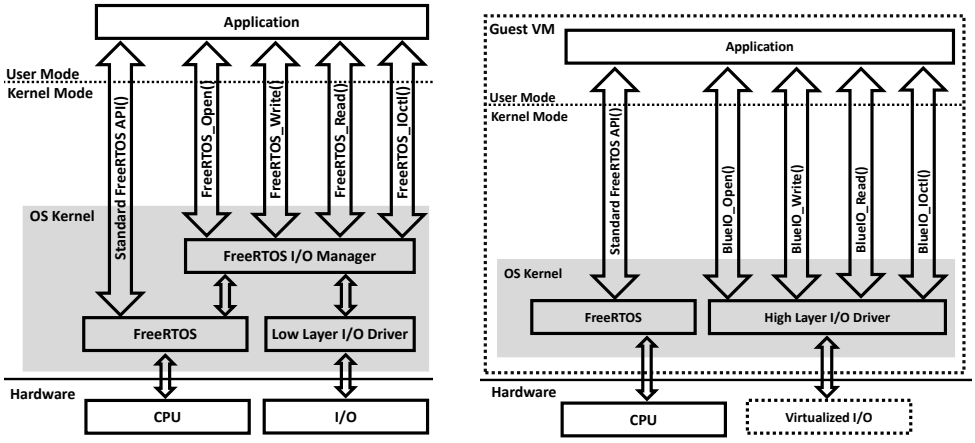
6.1.3 Virtual Machine (VM) and Guest OS

In our proposed approach, each CPU has an individual guest VM. Virtualization support in the system has following features:

- *Bare-metal virtualization* [102] - A guest OS can be executed on a CPU directly, without host OS. Therefore, a guest OS is able to execute in kernel mode to achieve full functionality.
- *Para-virtualization* [77] - I/O management module in each guest OS has to be replaced by high level I/O drivers, which enables smaller OS software and simplified I/O access paths.

Currently, in the proposed design, three OS kernels have been modified to support the I/O virtualization [21], i.e. FreeRTOS [7], uCosII [19] and Xilkernel [29]. In Figure 6.3, we use FreeRTOS as an example to demonstrate this modification.

Compared with the original FreeROTS kernel (Figure 6.3(a)), the user application in a modified kernel (Figure 6.3(b)) is able to access and operate I/Os via the high layer I/O drivers, which are independent from the kernel of



(a) Traditional FreeRTOS Kernel (b) Modified FreeRTOS Kernel

Figure 6.3: Traditional and Modified FreeRTOS Kernels

FreeRTOS. Additionally, user applications running on the original OS kernel can be ported to the modified kernel directly (without any modification), since we have not modified the OS interfaces.

The architecture builds upon three existing technologies, Virtualized Complicated Device Controller (**VCDC**) [72], GPIOCP [120] and BlueTree [60–62]. The full implementation of the BlueIO architecture is described in section 6.2.

6.2 BlueIO

The proposed BlueIO system contains four main modules (see Figure 6.4):(Note that: the I/O devices supported in the system can be both virtualized and non-virtualized; specifically, I/O virtualization is achieved via connecting I/O controllers and I/O devices to the VCDC).

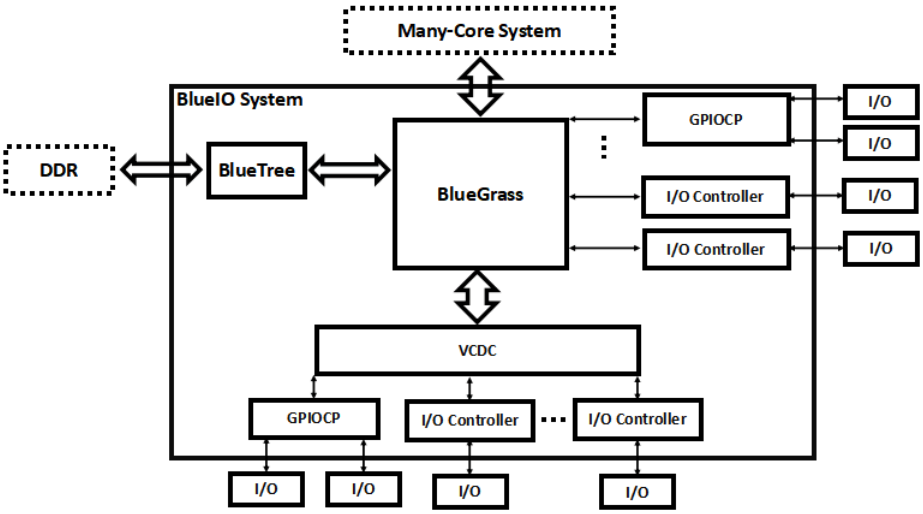


Figure 6.4: The Structure of the BlueIO

- BlueGrass — Is a communication interface between application CPUs, VCDC [120] (see Chapter 4), I/O controllers and external memories (DDR).
- Virtualized Complicated Device Controller (VCDC) [120] — Integrates functionalities of I/O virtualization and low layer I/O drivers.
- GPIO Command Processor (GPIOCP) [120] — Is a programmable real-time I/O controller, that permits applications to instigate complex se-

quences of I/O operations at an exact single clock cycle.

- BlueTree [60–62] — Provides an interface to access the external memory for I/O devices (e.g. DMA).

Note that GPIOCP and VCDC have been described extensively in Chapter 5 and 4, respectively. This section describes how they are utilized within BlueIO, and their interconnectivity.

6.2.1 BlueGrass

BlueGrass is the communication interface between application CPUs and BlueIO, including four communication interfaces:

- Interface from/to application CPUs;
- Interface from/to I/O controllers;
- Interface from/to VCDC;
- Interface from/to the external memory.

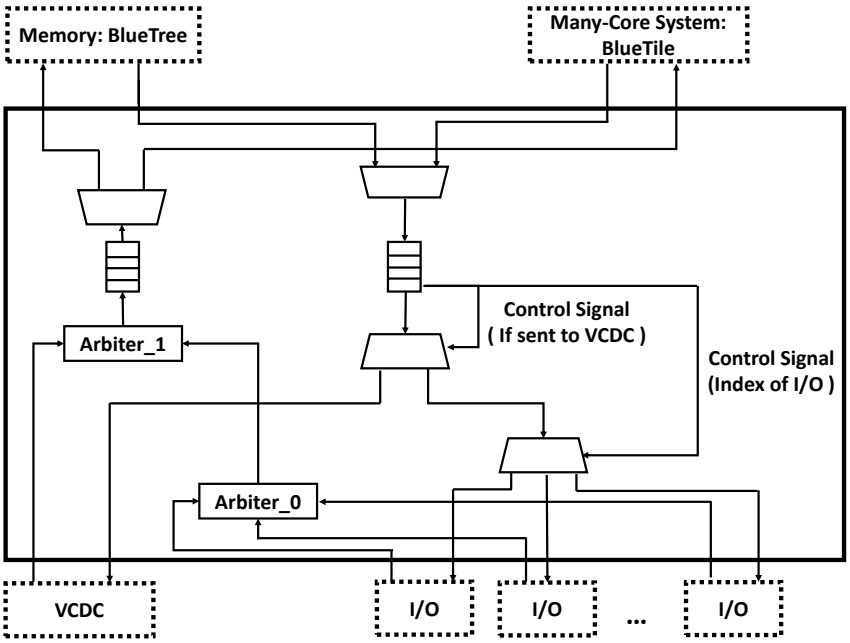


Figure 6.5: The Structure of the BlueGrass

In the proposed design, BlueGrass is physically connected to the NoC mesh (BlueTiles) and the memory access interface (BlueTree). Additionally, the I/O controllers can be directly connected to the Bluegrass to maintain the original functionalities, or indirectly connected to the VCDC to acquire I/O virtualization.

The structure of BlueGrass (see Figure 6.5) contains two parts: downward path and upward path. The downward path is responsible for allocating either I/O requests or memory fetched data to I/O devices. In addition, the upward path is responsible for sending I/O response back to application CPUs, as well as memory requests to the external memories.

Specifically, the downward path consists of three half-duplex multiplexers and a FIFO. The 2-into-1 multiplexer connected to BlueTile [95] and BlueTree [60] is designed to receive, and then queue the I/O requests and memory fetched data to the downward FIFO. The downward FIFO allocates these queued I/O requests and memory fetched data to a specified I/O according to the format of packets. The upward path consists of two arbiters, one half-duplex multiplexer and one FIFO. The arbiters determine the served sequence of I/O response and memory requests sent from each I/O. In order to prevent one single I/O dominating the upward path, and to be able to satisfy the requirement that the I/O system can be time-predictable, we have provided multiple real-time scheduling policies to both arbiters, including the Round-Robin, fixed priority and FIFO. In addition, users are also allowed to add a customized scheduling policy to the arbiters via our provided interface. The upward FIFO and connected 1-into-2 multiplexer are responsible for sending I/O responses and memory requests out of the BlueIO system.

6.2.2 Virtualized Complicated Device Controller (VCDC) [72]

As described in Chapter 4 [72], the Virtualized Complicated Device Controller (***VCDC***) was proposed to implement I/O virtualization and I/O drivers in hardware.

The VCDC can be physically connected to a many-core system, which is composed of two main parts (see Figure 6.6):

- I/O VMM - Maintains the virtualization of I/O devices.
- Low Layer I/O Drivers - Encapsulates the specific I/O drivers for a specific I/O controller (e.g. read the data from a specific address of the

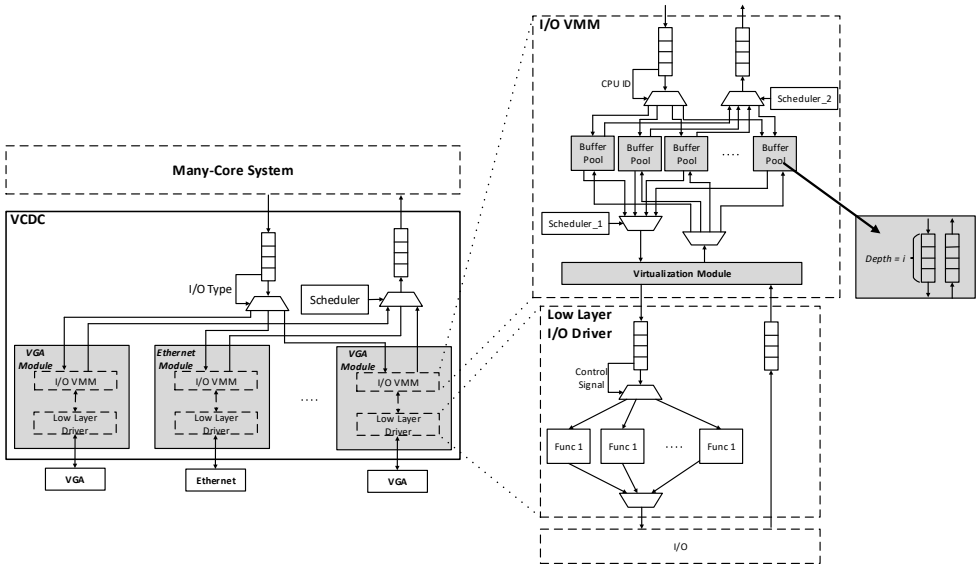


Figure 6.6: Structure of VCDC

SPI NOR-flash).

The I/O VMM has two main responsibilities: 1) Interpreting I/O requests (sent from a guest OS) to the actual I/O instructions (used to control a physical I/O); 2) Scheduling and allocating the interpreted I/O instructions to physical I/O. Considering that the functionalities and features of I/O devices are different, it is very difficult to build a general purpose module to achieve virtualization for all kinds of I/O devices. Therefore, we create some specific-purpose I/O VMM for those commonly used I/O devices, including UART, VGA, DMA, Ethernet, etc. Additionally, users can also easily add their customized I/O VMM into VCDC via our provided interfaces (see Chapter 4).

Note that, in this section, we only demonstrate the VCDC from high level, more details can be found in Chapter 4 and [72].

6.2.3 GPIO Command Processor (GPIOCP) [120]

As described in Chapter 5 and [120], the GPIO Command Processor (***GPIOCP***) was proposed. It is a resource efficient programmable I/O controller, which permits applications to instigate complex sequences of I/O operations at an exact time, so achieving timing-accuracy of a single clock cycle. This is achieved by loading application specific programs into the GPIOCP. Applications then

are able to invoke a specific program at run-time by sending the GPIO command, e.g. *Run command X at time t (at a future time).*

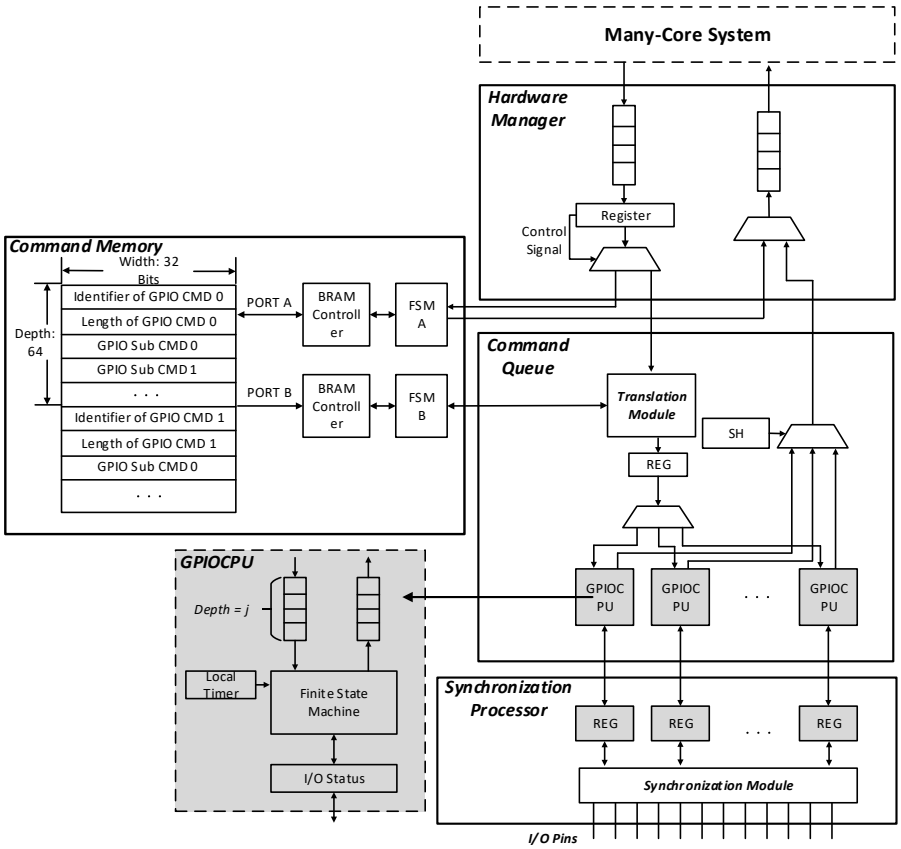


Figure 6.7: Structure of GPIOCP

The GPIOCP achieves cycle level timing-accuracy as the latencies of the I/O virtualization and communication bus are eliminated. For example, a periodic read of a sensor value by an application can be achieved by loading the GPIOCP with an appropriate program, then at run-time the GPIOCP issues a command such as *run command X at time t and repeat with period Z* — the values are read at exact times, with the latency of moving the data back to the application considered within that application's execution time.

The GPIOCP can be physically connected to a many-core system or VCDC, which is composed by four main components (see Figure 6.7):

- Hardware manager: Communicates with application CPUs, allocating incoming messages to either the command memory controller (to store

new commands) or the command queue (to initiate an existing command).

- Command memory controller: Stores a new GPIO command into the storage units; and accesses an existing GPIO command for execution by a GPIO CPU (within the command queue).
- Command queue: Allocates GPIO commands to GPIO CPUs for execution (cooperate with command memory controller). Each GPIO CPU is a simple finite state machine, with guaranteed execution time so achieving timing-accuracy.
- Synchronization processor: Synchronises the values of I/O pins, which may be written by different GPIO CPUs and I/O devices.

Further details can be seen in Chapter 5, [120] and [8].

6.2.4 BlueTree [62]

BlueTree is a tree-like memory interconnect built for many-core systems, which enables time-predictable memory read/write from a scaled number of CPUs and I/Os [62] [60]. BlueTree memory interconnect is designed to support the memory requirements of modern systems, leaving the TDM-based NoC for core-to-core communication only. BlueTree distributes memory arbitration across a set of 2-into-1 full-duplex multiplexers, each with a small arbiter (see Figure 6.8), rather than using a large monolithic arbiter next to memory, which allows the BlueTree to fulfill the scalability requirements of the system, and enable a larger number of requesters at a higher clock frequency than would be available with a single monolithic arbiter.

In order to prevent a single core dominating the tree, and to be able to satisfy the requirement that the memory subsystem can be time-predictable, each multiplexer contains a *blocking counter* which encodes the number of times that a high-priority packet (i.e., a packet from the left) has blocked a low-priority packet (i.e., a packet from the right). When this counter becomes equal to a fixed value m, the counter is reset and a single low-priority packet is given service. This then allows providing an upper bound of the WCET for a memory transaction. The specific timing analysis of BlueTree can be viewed in [62] [60] and [104].

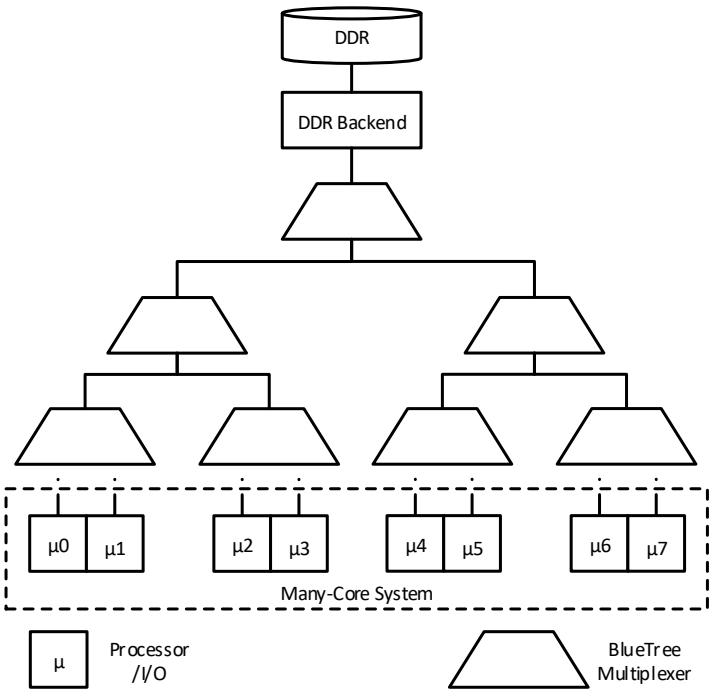


Figure 6.8: BlueTree Memory Hierarchy

6.3 Hardware Consumption Analysis

In this section, hardware consumption of BlueIO is analysed regarding its scalability. Firstly, the analysis is given to describe the hardware consumption of BlueIO; secondly, actual hardware consumption of BlueIO in VLSI (logic gates) and FPGA (LUTs, registers and BRAMs) is given.

In this hardware consumption analysis, we assume:

- Only BlueIO is included – hence BlueTree is not included (as the functionality of BlueTree is memory access, which is not necessary for all the I/Os).
- An independent I/O request buffer (buffer pool in VCDC) and an independent I/O request execution unit (GPIOCPU in GPIOCP) is allocated to each CPU – therefore, the number of buffer pools and GPIOCPUs in BlueIO equals the number of CPUs in the whole system.

We introduce the following terms:

- Number of CPUs in the system: **m**

  - I/Os are indexed as from IO_1 to IO_n: UART — IO_1, SPI flash — IO_2, VGA — IO_3, and Ethernet — IO_4.

- Hardware consumption: C where $C_x^{m,n}$ gives the hardware consumption of module x dependent on the number of CPUs (m) and I/Os (n) respectively.

In the analysis, we define the hardware consumption of a 1-CPU BlueIO system with GPIOCP ($C_{BIO}^{m=1,n=0}$) as the basic BlueIO system. We also define the difference between the m-CPU and n-IO BlueIO ($C_{BIO}^{m,n}$) and the basic BlueIO system as $\Delta C_{BIO}^{m,n}$. Therefore, the hardware consumption of an m-CPU and n-IO BlueIO system can be calculated as:

$$C_{BIO}^{\mathbf{m,\ n}} = C_{BIO}^{\mathbf{m=1,\ n=0}} + \Delta C_{BIO}^{\mathbf{m,\ n}} \tag{6.1}$$

Similarly, the variation of hardware consumption of the m-CPU and n-IO VCDC and GPIOCP compared with the basic systems are: $\Delta C_{VCDC}^{m,n}$ and $\Delta C_{GPIOCP}^{m,n}$.

BlueIO is comprised of BlueGrass, VCDC and GPIOCP (See Figure 6.4). Since the hardware consumption of BlueGrass is constant, the variation of hardware consumption in BlueIO ($\Delta C_{BIO}^{m,n}$) equals the sum of the variation of hardware consumption occurred in VCDC ($\Delta C_{VCDC}^{m,n}$) and GPIOCP ($\Delta C_{GPIOCP}^{m,n}$):

$$\Delta C_{BIO}^{\mathbf{m,\ n}} = \Delta C_{VCDC}^{\mathbf{m,\ n}} + \Delta C_{GPIOCP}^{\mathbf{m,\ n}} \tag{6.2}$$

The hardware consumption of the VCDC (see Figure 6.6) is dominated by I/O VMMs and buffer pools (around 99% in our design). Hence we consider VCDC hardware consumption as the summation of I/O VMMs (C_{VIO_i}) and buffer pools (C_{BP}), and ignore the effects from the other variables. In our design, the hardware consumption of an I/O VMM (C_{VIO_i}) and a buffer pool (C_{BP}) is constant. Additionally, the number of I/O VMMs equals the number of I/Os, meanwhile, the number of buffer pools equals the number of CPUs. Therefore, the increased hardware consumption of VCDC ($\Delta C_{VCDC}^{m,n}$) is calculated as:

$$\Delta C_{VCDC}^{\mathbf{m,\ n}} \approx \sum_{i=1}^{\mathbf{n}} (C_{VIO\_i} + \mathbf{m} * C_{BP}) \tag{6.3}$$

In GPIOCP (see Figure 6.7), the only variation related to its hardware consumption is the number of GPIOCPUs (C_{GCPU}), equalling to the number of

CPUs in the whole system. Therefore, the variation of hardware consumption of GPIOCP (ΔC_{GPIOCP}) is calculated as:

$$\Delta C^{\mathbf{m}}_{GPIOCP} = (\mathbf{m} - 1) * C_{GCPU} \tag{6.4}$$

Combining equations 6.1, 6.2, 6.3, and 6.4 gives the hardware consumption of BlueIO to be:

$$C^{\mathbf{m,\ n}}_{BIO} = C^{\mathbf{m=1,\ n=0}}_{BIO} + \sum_{i=1}^{\mathbf{n}} (C_{VIO\_i} + \mathbf{m} * C_{BP}) + (\mathbf{m} - 1) * C_{GCPU} \tag{6.5}$$

Expanding gives:

$$C^{\mathbf{m,\ n}}_{BIO} = C^{\mathbf{m=1,\ n=0}}_{BIO} + \sum_{i=1}^{\mathbf{n}} C_{VIO\_i} + (\mathbf{m} - 1) * C_{GCPU} + \mathbf{m} * \mathbf{n} * C_{BP} \tag{6.6}$$

Equation 6.6 shows the hardware consumption of implementing BlueIO is:

- Linearly scaled with the number of I/Os (n), while the number of CPUs (m) is constant;
- Linearly scaled with the number of CPUs (m), while the number of I/Os (n) is constant.

6.3.1 Implementing BlueIO in VLSI

This section shows that the implementation of BlueIO in VLSI has scalable hardware consumption at the gate level.

Firstly, we use Cadence RTL encounter compiler (v11.20) [5] to synthesis and provide gate level hardware consumption of each basic component in BlueIO respectively, i.e. $C^{m=1,n=0}_{BIO}$, C_{GCPU}, C_{BP}, and C_{VIO_n} (see Table 6.1). Secondly, we synthesise BlueIO with different number of CPUs and I/Os respectively, and exhibit their gate level hardware consumption in Table 6.2. Note that $OSU\_SOC\_v2.5$ [1] is the open source MOSIS SCMOS TSMC 0.25um library used in the synthesis.

The consumption of logic gates may be varied by a specific synthesis compiler and adopted synthesis library.

Table 6.1 shows I/O VMM (C_{VIO_n}) consumes more gates resources when compared with GPIOCPU (C_{GCPU}) and buffer pool (C_{BP}). Therefore, even

Table 6.1: Hardware Consumption of Basic Modules (Gate Level)

Component	$C_{BIO}^{m=1,n=0}$	C_{GCPU}	C_{BP}	C_{VIO_1}	C_{VIO_2}	C_{VIO_3}	C_{VIO_4}
AND	201	64	47	328	621	512	981
AOI	1,085	369	36	1,502	2,381	2,201	4,523
DFFPOS	1,020	382	54	1,196	2,021	1,981	3,708
HA	12	6	1	13	18	15	60
INV	1,346	666	59	1,621	2,531	2,512	5,128
MUX2	7	5	0	10	14	16	80
NAND	745	477	70	1,253	1,573	1,789	3,001
NOR	572	248	25	7,61	1,221	1,201	2,401
OAI	633	420	35	1,066	1,652	1,602	3,101
OR	115	35	2	62	141	142	250
XNOR	9	10	0	26	40	36	32
XOR	10	6	3	21	20	20	52
Total	5,755	2,688	332	7,859	12,233	12,027	23,317

though the hardware consumption of BlueIO is linearly scaled by the number of CPUs (m) and I/Os (n) respectively (see equation 6.5), the number of I/Os (n) and the specific implementation of the corresponding I/O VMMs (C_{VIO_n}) dominates the hardware consumption.

Table 6.2 shows that the hardware consumption of BlueIO is linearly increased with the number of CPUs (m) and I/Os (n) respectively. Specifically, if the number of I/Os (n) is fixed, the hardware consumption may be slightly linearly increased with the number of CPUs (m). Similarly, if m is fixed, the hardware consumption may be obviously linearly increased with the addition of I/Os (n). Additionally, the types of added I/Os can also affect the hardware consumption — the required logic gates of a simple I/O (e.g. $C_{BIO}^{m=1,n=0}$ with IO_1) is far less than a complicated I/O (e.g. $C_{BIO}^{m=1,n=0}$ with IO_4).

6.3.2 Hardware Consumption in RTL Level (FPGA)

Vivado (v2016.2) was used to synthesise and implement BlueIO on Xilinx VC709 FPGA board [20] with increasing numbers of I/Os and CPUs. The hardware consumption of BlueIO was recorded at the RTL level in terms of LUTs, registers, BRAMs, power consumption and maximum working frequency.

The resource efficiency of BlueIO is shown by Table 6.3 and 6.4, e.g. a full featured 2-CPU BlueIO only consumes 2.24% LUTs and 1.04% Registers

Table 6.2: Hardware Consumption of BlueIO (Gate Level)

[!h]	$C_{BIO}^{m=1,n=0}$	+ IO_1			+ IO_2			+ IO_3			+ IO_4		
Numb. CPUs	1	1	2	4	1	2	4	1	2	4	1	2	4
AND	201	292	381	482	529	550	680	1,006	1,205	1,379	1,921	2,025	2,150
AIO	1,085	1,579	1,996	2,852	2,573	2,988	3,925	4,769	5,268	6,233	9,222	9,852	10,850
DEEPOS	1,020	1,288	1,695	2,512	2,188	2,776	3,752	3,988	4,520	5,425	7,588	7,992	8,895
HA	12	47	52	68	27	34	48	39	46	59	98	106	120
INV	1,346	1,801	2,623	4,156	2,909	3,650	5,125	5,371	6,210	7,685	10,307	11,125	12,650
MUX2	7	16	21	32	16	20	33	31	38	48	113	125	141
NAND	745	972	1,525	2,487	1,876	2,501	3,602	3,449	4,000	5,153	6,330	6,952	8,053
NOR	572	729	1,051	1,753	1,233	1,666	2,325	2,350	3,052	3,752	4,661	5,125	6,002
OAI	633	775	1,325	2,423	1,694	2,050	3,112	3,241	3,825	4,057	6,337	6,925	8,125
OR	115	83	125	193	182	252	388	312	388	412	579	628	755
XNOR	9	7	19	43	29	41	65	64	79	102	96	113	141
XOR	10	16	28	49	27	39	57	46	55	71	91	102	115
Total	5,755	7,605	10,841	17,050	13,283	16,567	23,112	24,666	28,686	34,376	47,343	51,070	57,997

of the VC709 FPGA board. As shown, DSP slices are not required by the implementation of BlueIO on FPGA. Additionally, the number of LUT slices and registers linearly increase as the number of I/Os and CPUs increase respectively. Furthermore, the increased hardware consumption also leads to a linear increment in power consumption; and a decrease in the maximum working frequency. The maximum frequency is inversely proportional to the size of the RTL design [112], specifically described in Section 2.3.

Table 6.3: Hardware Consumption of 2-CPU BlueIO with Different I/Os on FPGA (RTL Level)

Added I/O	Hardware Consumption								Power (mW)	Maximum Frequency (Mhz)
	LUTs	% of VC709	Register	% of VC709	BRAMs	% of VC709	DSP	% of VC709		
+ UART	2192	0.12%	1471	0.17%	0	0%	0	0%	13	221.8
+ VGA	4566	0.51%	2315	0.27%	0	0%	0	0%	19	221.8
+ SPI Flash	6120	1.41%	4225	0.49%	0	0%	0	0%	29	221.8
+ Ethernet	9723	2.24%	9035	1.04%	0	0%	0	0%	75	192

Table 6.4: Hardware Consumption of BlueIO (+GPIOCP) with Different Number of CPUs on FPGA (RTL Level)

Number of CPUs	Hardware Consumption								Power (mW)	Maximum Frequency (Mhz)
	LUTs	% of VC709	Register	% of VC709	BRAMs	% of VC709	DSP	% of VC709		
1	632	0.146%	962	0.111%	16	1.09%	0	0%	19	318
2	886	0.205%	1156	0.113%	16	1.09%	0	0%	20	303
4	1314	0.303%	1468	0.169%	16	1.09%	0	0%	22	291
8	1942	0.448%	2094	0.242%	16	1.09%	0	0%	25	284
16	3236	0.747%	3346	0.386%	16	1.09%	0	0%	31	249
32	5065	1.169%	5311	0.613%	16	1.09%	0	0%	37	236
64	8698	2.008%	8449	0.975%	16	1.09%	0	0%	50	204

6.4 Evaluation

The BlueIO was implemented using Bluespec [4] and synthesised for Xilinx VC709 development board [20] (further implementation details are given in Appendix A and B).

The BlueIO system was connected to a 4 x 5 2D mesh type open source NoC [95] containing 16 Microblaze CPUs [11] running the modified guest OS (FreeRTOS v9.0.0) in the guest VM (see Section 6.1.3). The architecture is shown in Figure 6.9.

To enable comparison, a similar hardware architecture without the BlueIO system was built - note that this architecture requires I/O operations requested

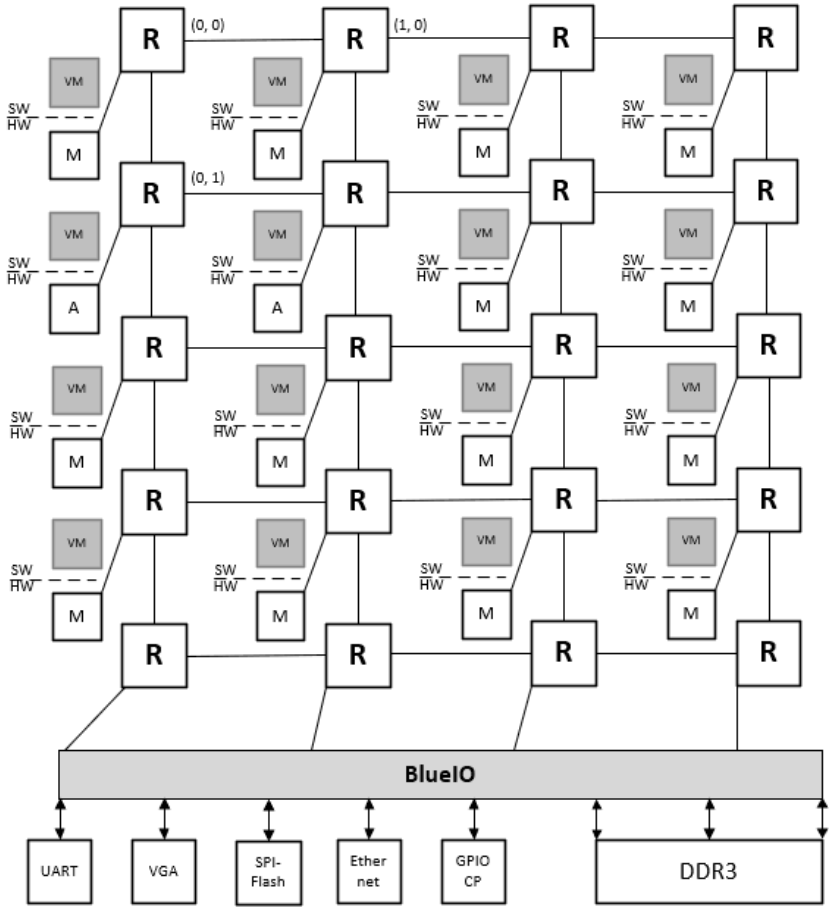


Figure 6.9: Experimental Platform
(M - Microblaze; A - ARM Processor;
VM - Guest VM; R - Router / Arbiter)

by Mircoblazes to pass through the mesh to the I/O rather than being controlled by a BlueIO. Both architectures run at 100 MHz.

6.4.1 Memory Footprint

In this section, we evaluate the memory footprint of BlueIO, as well as different versions of FreeRTOS running on Microbalze CPU, via the size tool of the Xilinx Microblaze GNU Tool chain. In the measurement, the native version of FreeRTOS (*nFreeRTOS*) is full-featured [7], which is the foundation of the

other versions [1] [2] [3]. Table 6.5 presents the collected measurements.

Table 6.5: BlueIO Memory Footprint (Bytes)

Software	Memory Footprint			
	.text	.data	.bss	Total
BlueIO	0	0	0	0
nFreeRTOS	121,309	1,728	35,704	158,741
nFreeRTOS + I/O	179,652	1,852	36,250	217,754
vFreeRTOS + I/O	189,556	1,882	36,450	227,888
BV_vFreeRTOS + I/O	131,969	1,732	35,723	169,424

As it can be seen, the memory overhead introduced by the hypervisor(BlueIO) is *zero*, resulting from its pure hardware implementation. The native full-featured FreeRTOS (*nFreeRTOS*) requires 158741 bytes – with I/O module added, the memory footprint increases 37.18%, owing to the addition of I/O manager and I/O drivers. When it comes to the *vFreeRTOS + I/O*, the introduction of software implemented virtualization increases the memory footprint to 227, 888 bytes. However, the *BV_vFreeRTOS + I/O* only consumes 169, 424 bytes of memory, which is 6.73% increased compared to the native FreeRTOS, as well as 77.81% and 74.35% of the *nFreeRTOS + I/O* and *vFreeRTOS + I/O*, respectively. The main reason behind such a low memory footprint is the implementation of para-virtualization (described in Section 6.1.3), has removed the software overhead significantly.

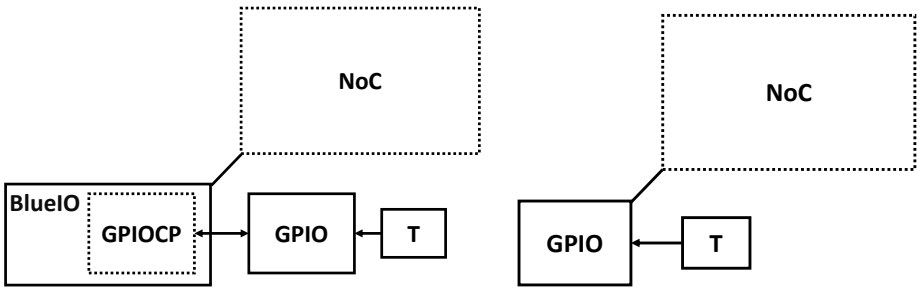
6.4.2 Real-time Features

This experiment aims to evaluate the predictability and timing accuracy of the I/O operations in a BlueIO and a non-BlueIO system. In both architectures, 9 CPUs are active, whose coordinates are from (0, 0) to (0, 2), (1, 0) to (1, 2) and (2, 0) to (2, 2). When CPUs are required to access and read the GPIO at a specific time, then for a non-BlueIO architecture the CPU has to instigate the I/O operation, for the BlueIO architecture, this can be delegated to the BlueIO (GPIOCP) to achieve timing accuracy. This was shown by connecting

[1] *FreeRTOS + I/O* involves UART, VGA and corresponding drivers.

[2] *vFreeRTOS* is a simply implemented software virtualized FreeRTOS for many-core systems, see [72].

[3] *BV_vFreeRTOS* is the virtualized FreeRTOS in BlueIO system.



(a) BlueIO System (b) Non-BlueIO System

Figure 6.10: Experimental Setup for the Timing Accuracy of I/O Operations (T - Timer)

a timer to the GPIO (updating its value every cycle), with every CPU needing to read the value simultaneously.

The results of 1000 experiments are given in Table 3.4 and 5.1, showing that the latencies and variance for the non-BlueIO architecture (baseline system) are significant (errors calculated according to equation 3.5); in contrast, the BlueIO architecture is predictable and accurate at the cycle level, which is totally the same as the GPIOCP architecture evaluated in Chapter 5. This results from the employment of the real-time I/O controller (i.e. GPIOCP).

6.4.3 Performance Features — I/O Performance

The I/O performance evaluation considers I/O response time and I/O throughput separately in the following sections.

6.4.3.1 I/O Response Time

This experiment is designed to evaluate the I/O response time whilst CPUs and measured I/O are fully loaded within a BlueIO and non-BlueIO system. In both architectures, all the active CPUs have an independent application that is set to be running, which continuously reads data from an SPI NOR-flash (model: S25FL128S). Specifically, the experiment is divided into four groups, depending on the number of reading bytes: 1, 4, 64 and 256 bytes. All experiments are implemented 1,000 times. We name the experiments according to the scheduling policy and bytes of read data in once I/O request. For example, *non-BlueIO-RR-4B* stands for a non-BlueIO system with Round-Robin global scheduling policy; and 4 bytes of data read from the NOR-flash in once I/O

request.

In the non-BlueIO architecture, we modify the FreeRTOS to be suitable for many-core systems[4]. In both architectures, while the user applications on different CPUs are requesting the I/O at the same time point, the scheduling policy can be set as local FIFO (non-BlueIO-FF and BlueIO-FF) and global Round-Robin (non-BlueIO-RR and BlueIO-RR) respectively. Due to the non-readability, the table with entire experimental results is shown in [21, 72]. Instead, a summarized version of experimental results showing the worst case and variation of each group of experiments are demonstrated in Table 6.6 and 6.7.

Table 6.6: I/O Response Time in Non-BlueIO Systems (unit: clock cycle) (Summarized Version)

Written Bytes	Non-BlueIO (FIFO)		Non-BlueIO (Round-Robin)	
	Worst Case	Variation	Worst Case	Variation
1	9,357	1,541	65,885	59,736
4	58,844	7,061	327,813	286,733
8	936,166	98,026	4,555,159	3,823,104
16	3,702,565	284,142	17,345,151	15,475,355

Table 6.7: I/O Response Time in BlueIO Systems (unit: clock cycle) (Summarized Version)

Written Bytes	BlueIO (FIFO)		BlueIO (Round-Robin)	
	Worst Case	Variation	Worst Case	Variation
1	532	57	403	46
4	1,785	368	1,569	276
8	25,053	3,667	23,032	3,542
16	92,153	15,225	89,708	13,711

Table 6.6 shows that the worst case response time of I/O requests in the non-BlueIO architecture is significantly high for the reading of 1, 4, 64 or 256

[4]FreeRTOS is designed for a single-core system; in our experiments, we modify it to be suitable for many-core systems [72]

byte(s) from the NOR-flash, especially while global Round-Robin scheduling policy being employed – noting that a lower I/O response time indicates a higher I/O performance. In experiments with the number of read bytes increased (see Table 6.7), BlueIO system maintains its superior performance. Additionally, when it comes to the variation, the BlueIO systems also always have a better performance than the non-BlueIO systems. For example, in the non-BlueIO-FF-1B, the variation is greater than $1,500$ clock cycles; and in non-BlueIO-RR-1B, the variation reaches $60,000$ clock cycles. Conversely, in both BlueIO-FF-1B and BlueIO-RR-1B, the highest variance is less than 60 clock cycles.

Therefore, the evaluation results reveal that a system with BlueIO provides more predictable I/O operations with lower response time.

6.4.3.2 I/O Throughput

We evaluate the I/O throughput in two architectures (with BlueIO and without BlueIO). In the experiments, we use the same NOR-flash illustrated in the previous section as our tested I/O. Additionally, the scheduling policy in both architectures is set as local FIFO and global Round-Robin respectively.

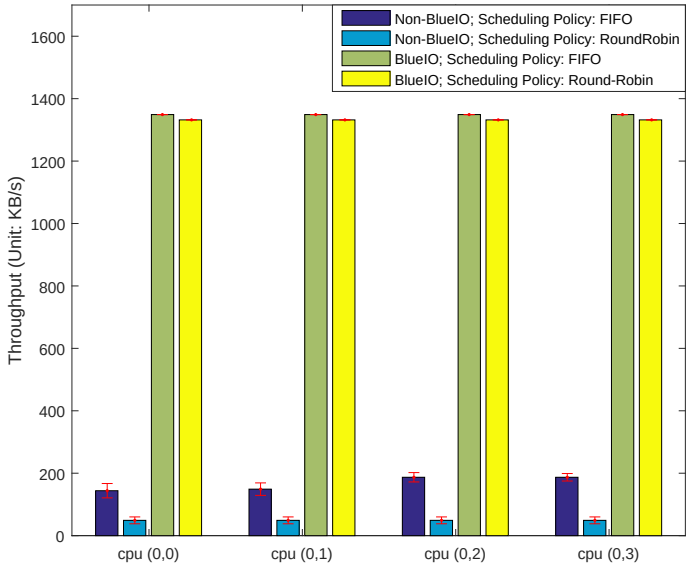


Figure 6.11: I/O Throughput

In both architectures, an independent application is set to be running on each of 4 Microblaze CPUs, whose coordinates are from (0,1) to (0,3))

and continuously writing to the NOR-flash — one byte can be written in a single I/O request. The number of bytes written from each CPU per second is recorded as the I/O throughput (unit: KB/s). The result of higher I/O throughput implies a better performance. All the evaluations are implemented 1,000 times. The evaluation results are shown in Figure 6.11.

As shown, four groups of bar charts present the average I/O throughput in the BlueIO system and the non-BlueIO system; and the error bar on each bar chart presents the variance of the I/O throughput during these 1,000 experiments. As shown, on all CPUs considered, no matter which scheduling policy is deployed, the BlueIO system always provides a better performance on I/O throughput (nearly 7 times), and less variance.

6.4.4 Performance Features — Timing Scalability

In this section, we evaluate the timing scalability of the BlueIO system via a connected complex device — Ethernet. The evaluation is implemented by measuring the I/O response time of Ethernet packets sent from different CPUs in single-core, 4-core, 8-core and 16-core systems, respectively. The implementation of the Ethernet virtualization in BlueIO system can be found in [72,72].

The experiment is divided into two parts, dependent on the global scheduling policy of the BlueIO: Round-Robin (named BlueIO-RR) and fixed priority (named BlueIO-FP). In BlueIO-RR and BlueIO-FP, the experiments can be further divided into four parts, according to the number of active CPUs. In these four parts of the experiments, we activate 1, 4, 8 and 16 Microblaze CPUs respectively. We name these experiment parts according to the global scheduling policy of the experiment plus the number of active CPUs. For example, in a 4-core BlueIO system with Round-Robin global scheduling policy, the experiment is labelled as BlueIO-RR-4.

The software application running on each active CPU is the same, and is designed to continuously send 1 KB Ethernet packets via BlueIO to a dedicated component. The 1 KB Ethernet packets sent from different CPUs are exactly the same. The dedicated component is designed to monitor the response time of these Ethernet packets by recording the reach time and analysing the virtual source IP address of the packets. All the experiments were implemented 1000 times; and the experiment results and Δr (described in eq 3.4) are depicted in tables.

In BlueIO-FP, CPU (0, 0) is always set as the highest priority, followed

Table 6.8: Average Response Time of Loop Back 1KB Ethernet Packets in BlueIO System (Global Scheduling Policy: Fixed Priority; Unit: us)

Number of CPUs	CPU Coordinate																Δr
	(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)	(0,3)	(1,3)	(2,3)	(3,3)	
1	11.5	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0
4	12.0	25.5	36.9	48.3	X	X	X	X	X	X	X	X	X	X	X	X	1.29
8	12.1	25.5	36.9	48.3	59.7	71.2	82.6	94.0	X	X	X	X	X	X	X	X	0.96
16	12.0	25.5	36.9	48.3	59.7	71.1	82.6	95.0	105.4	116.9	128.3	139.7	151.1	162.5	174.0	185.4	0.8

Table 6.9: Average Response Time of Loop Back 1KB Ethernet Packets in BlueIO System (Global Scheduling Policy: Round Robin; Unit: us)

Number of CPUs	CPU Coordinate																Δr
	(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)	(0,3)	(1,3)	(2,3)	(3,3)	
1	11.0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0
4	46.7	47.2	47.6	48.1	X	X	X	X	X	X	X	X	X	X	X	X	0.84
8	90.5	90.8	91.2	91.5	91.9	92.2	92.6	92.9	X	X	X	X	X	X	X	X	0.44
16	180.1	180.7	179.9	180.6	180.0	180.7	180.0	180.7	180.0	180.7	180.0	180.7	180.0	180.7	180.0	180.7	0.25

by CPU (1, 0), (2, 0), (3,0) and (1, 0) etc. The experiment results are shown in Table 6.8. As shown, for all many-core systems, the I/O response time from the CPU with the highest priority is always fixed around 12 us; and the I/O requests from the CPUs with the lower priorities are always blocked by the I/O requests with higher priorities, which guarantees the execution of the I/O requests with higher priorities. For example, in BlueIO-FP-8, the average response time of the I/O requests from CPU (0,0) (the highest priority) is kept to 12 us, which means it can never be blocked by others. When it comes to the I/O requests from CPU (3, 1) (the lowest priority), the I/O response time is always around 94 us, which is 8 times of the highest priority I/O requests. In an 8-core system, the theoretical optimal response time of the lowest priority I/O request should be 8 times the highest priority I/O request, which means that the BlueIO system does not introduce an extra delay for the lowest priority I/O request; and our experiment results obtain this. In addition, with the number of CPUs increased, there is no obvious increment in Δr, which implies the loss of I/O performance is not significant, while the number of CPUs being increased, as well as the good scalability of the BlueIO system (with the fixed priority scheduling policy).

In BlueIO-RR, the global arbiter is set to start from operating a random I/O request. The experiment results are shown in Table 6.9. As shown, with an increment in the number of CPUs, the I/O response time of each CPU is proportional to the number of CPUs. For example, the average response time of an I/O request in BlueIO-RR-4, BlueIO-RR-8 and BlueIO-RR-16 are close to their theoretical optimal values, which are around 4, 8 and 16 times of the one in a single-core system (BlueIO-RR-1). In addition, with the number of CPUs increased, there is no obvious increment in Δr, which also shows the good scalability of the BlueIO system (with the Round-Robin scheduling policy).

6.4.5 On-chip Communication Overhead and Scalability

In NoC-based many-core systems, all the I/O requests are transmitted as on-chip packets. A larger requirement for on-chip packets means a higher on-chip communication overhead. In this section, we compare the on-chip communication overhead while invoking commonly used I/O requests in a BlueIO and non-BlueIO system by recording the number of packets on the NoC. In the NoC [95], the width of all the on-chip packets is 32 bits. The

evaluation results are demonstrated in Table 6.10. Results show that whilst the invoked I/O request is simple, the on-chip communication overhead is similar in all the systems, e.g. displaying one pixel via the VGA in a single-core system. When the I/O operations become complicated or the number of CPUs is increased, the on-chip communication overhead in non-BlueIO architecture is significant; in contrast, the BlueIO architecture has a lower on-chip communication overhead, for example, reading 10 bytes data from the SPI flash in 10-core systems.

Table 6.10: On-chip Communication Overhead

I/O Device	I/O Operation		Number of on-chip Packets (Each Packet: 32-bit)		
			Non-VCDC FIFO	Non-VCDC Round-Robin	VCDC
VGA	Display 1 Pixel	1 CPU	6	6	3
		4 CPUs	24	33	12
		10 CPUs	60	87	30
	Display 10 Pixels	1 CPU	60	60	30
		4 CPUs	240	357	120
		10 CPUs	600	897	300
SPI Flash	Read 1 Byte	1 CPU	12	12	4
		4 CPUs	48	57	16
		10 CPUs	120	237	40
	Read 10 Bytes	1 CPU	120	120	40
		4 CPUs	480	597	160
		10 CPUs	1200	1497	400

6.5 Summary

In this chapter, we have presented a scalable hardware-implemented real-time I/O virtualization system for multi-core and many-core systems — BlueIO. It simultaneously enables improved performance features (i.e. I/O performance and scalability) compared to baseline systems, real-time features (i.e. predictability and timing-accuracy) and protection features (i.e. parallel accesses and isolation). BlueIO is designed based on previous research presented in this thesis — VCDC, GPIOCP and BlueTree, integrating most of the functionalities of I/O virtualization, low layer I/O drivers and the clock cycle level timing-accurate I/O controller (GPIOCP) in hardware layer, meanwhile providing abstracted high-layer access interfaces to software layers (Guest VMs).

Evaluation reveals that BlueIO can support virtualization of a physical

I/O device to multiple virtual I/O devices with good performance features, including faster I/O response time, higher I/O throughput, less on-chip communication overhead and good scalability. In additional, BlueIO can also handle multiple I/O operations with clock cycle accuracy, in many cases totally timing-accurate and predictable. In the hardware consumption analysis, we demonstrate the hardware consumption of BlueIO linearly scales with the number of CPUs and I/Os respectively, evidenced by our implementation in VLSI and FPGA.

The major contributions of the chapter follow. Firstly, Section 6.1 proposed BlueIO, and gave the general architecture and the corresponding I/O virtualization. Specifically, Section 6.1.1 briefly introduced the general architecture of BlueIO, followed by the system context introduced in Section 6.1.2. Section 6.1.3 describes the I/O virtualization in the BlueIO-based system — bare-metal virtualization and para-virtualization. Secondly, Section 6.2 presented the specific design and implementation details of BlueIO, which included the high level structure (shown in Figure 6.4), as well as the detailed introduction of internal components — BlueGrass, VCDC, GPIOCP and BlueTree (from Section 6.2.1 to 6.2.4). Thirdly, Section 6.3 demonstrated the hardware consumption analysis of BlueIO, which showed that it linearly scales with the number of CPUs and I/O devices respectively. The results are evidenced by our implementation which targets both VLSI (see Section 6.3.1) and FPGA (see Section 6.3.2). Finally, Section 6.4 evaluated BlueIO. Section 6.4.1 evaluated the memory footprint (software overhead) of BlueIO and BlueIO-based systems. Due to the hardware implementation of BlueIO, the software overhead in a BlueIO-based system is significantly lower than a conventional solution. Moreover, Section 6.4.2 evaluated the real-time features of BlueIO by measuring the error in timing-accuracy (E) of two architectures and corresponding variances (with and without BlueIO). The evaluation results revealed that a BlueIO-based system can handle multiple I/O devices with clock cycle timing-accuracy and predictability. Section 6.4.3 evaluated the I/O performance via I/O throughput and I/O response time. The evaluation results revealed that BlueIO significantly enhances the I/O performance compared to a non-BlueIO architecture — increased I/O throughput and reduced I/O response time. Meanwhile, Section 6.4.4 evaluatd timing scalability via measuring the I/O response time of a BlueIO architecture with a different number of processors. As shown in the evaluation results, the BlueIO system

achieves better scalability compared to a non-BlueIO architecture.

Overall, BlueIO simultaneously provides good real-time features and performance features on I/O virtualized systems. Meanwhile, the deployment of I/O virtualization also brings protection features (i.e. parallel accesses and isolation). Therefore, BlueIO has successfully solved the research question 3.

The following chapter proposes a real-time hypervisor (case study) built upon VCDC (see Chapter 4), GPIOCP (see Chapter 5), and BlueIO (see Chapter 6), in order to show our methodologies can be expanded to different system architectures and platforms, with kept features on real-time, performance and protection.

Chapter 7

BlueVisor: A Scalable Real-Time Hardware Hypervisor for Many-core Embedded Systems

VCDC (see Chapter 4) has solved the research question 1; GPIOCP (see Chaprter 5) has solved the research question 2; and BlueIO (see Chapter 6) has solved research question 3. The main aim of this chapter is to solve the fourth research problem: "*How to integrating the ready-built I/O system to the complete system with the expected features inherited?*", see Chapter 1, Section 1.4.

Currently, virtualization technology is widespread in real-time embedded systems [65, 87, 91, 93, 102, 121], resulting from the availability of hardware support. Hardware assistance allows the penalties suffered by traditional software virtualization technologies to be alleviated, e.g., significant software overhead [109]. However, current technologies are not necessarily applicable to real-time systems as they are not designed to satisfy strict performance and timing requirements and constraints [73].

In this chapter, we propose a scalable real-time hardware hypervisor for multi-core and many-core embedded system, termed BlueVisor, which enables predictable virtualization on CPU, memory, and I/O, as well as fast interrupt handler, and inter-VM communication.

We propose the design idea and specific implementation of the real-time hy-

pervisor, as well as demonstrate how a BlueVisor-based virtualization system can be adequately exploited to meet the real-time requirements with significant improvements on system performance, while presenting a low performance cost executing different operating systems (*OSs*).

This chapter is organized as follows: Section 7.1 proposes an overview of BlueVisor, which describes the general architecture. Section 7.2 demonstrates the design and implementation details of BlueVisor. Furthermore, Section 7.3 evaluates BlueVisor via multiple metrics. At last Section 7.4 and 7.5 discusses the drawbacks of BlueVisor, and draws the conclusion, respectively.

7.1 Overview

The design of the proposed real-time hardware hypervisor relies upon real-time hardware assistance (i.e. VCDC (see Chapter 4 [72]), GPIOCP (see Chapter 5 [120]), and virtualized BlueTree [35]) to move virtualization and low layer drivers from software to hardware, including the virtualization of CPU, memory, I/O and interrupts, as well as the inter-VM communication, and I/O drivers, while providing abstracted high layer access interfaces for guest VMs/OSs.

The hardware implemented hypervisor offloads the majority of the overhead of virtualization (see Section 7.2) to hardware. This enables guest OSs to execute in ring 0 with full privilege and removes application latency to the OS, the buses/routers. Also, indirection and interposition of privileged instructions are not required. Therefore, the real-time properties can be improved. Additionally, the hardware implemented low layer drivers and the abstracted high layer access interfaces (in software layer) significantly improve the system performance.

7.1.1 General Architecture

Figure 7.1 shows the proposed embedded virtualization architecture. The RTOS kernel in each VM can be executed in kernel mode (ring 0) to achieve full functionality. Meanwhile, it can provide a real-time environment for applications that need to guarantee deadlines. Finally, the hypervisor, running in hardware, is responsible for system virtualization, physical isolation between VMs, and providing high layer access interfaces for user applications (in guest VMs).

The architecture uses existing technologies: VCDC (see Chapter 4 [72]), GPIOCP (see Chapter 5 [120]) and BlueTree (see Section 6.2.4 [62]).

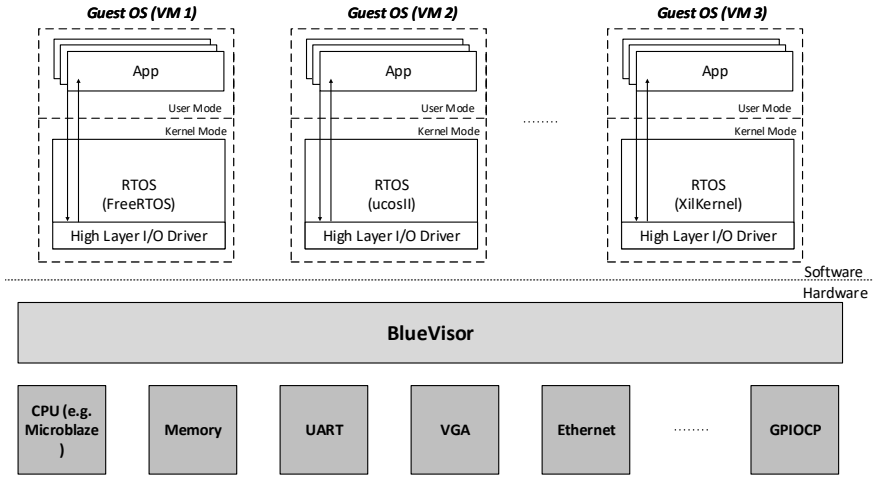


Figure 7.1: Embedded Virtualization Architecture

7.2 BlueVisor: Implementation

In this chapter, VCDC (see Chapter 4 [72]), GPIOCP (see Chapter 5 [120]), BlueTree (see Section 6.2.4 [62]) and the memory virtualization extension are supplemented with the real-time hardware hypervisor (***BlueVisor***). The motivation is to enhance the real-time features and performance features of whole system. BlueVisor is mounted to a 2D mesh type open source NoC, termed BlueTiles [95]. Use of a NoC is not required by BlueVisor, because it is a general-purpose hypervisor, which is agnostic to the type of bus and the software running on CPUs. To support a complete BlueVisor system, the platform requires communication channels between BlueVisor and CPUs and a global synchronization timer.

The use of BlueVisor within BlueTiles is shown in Figure 7.2. BlueVisor is physically connected to the home port (via the physical link) of a router, as well as the global timer.

7.2.1 CPU Virtualization and Guest VM

In our proposed approach, each processor (whatever the architecture) is set as an individual guest VM. The virtualization in the system has the following

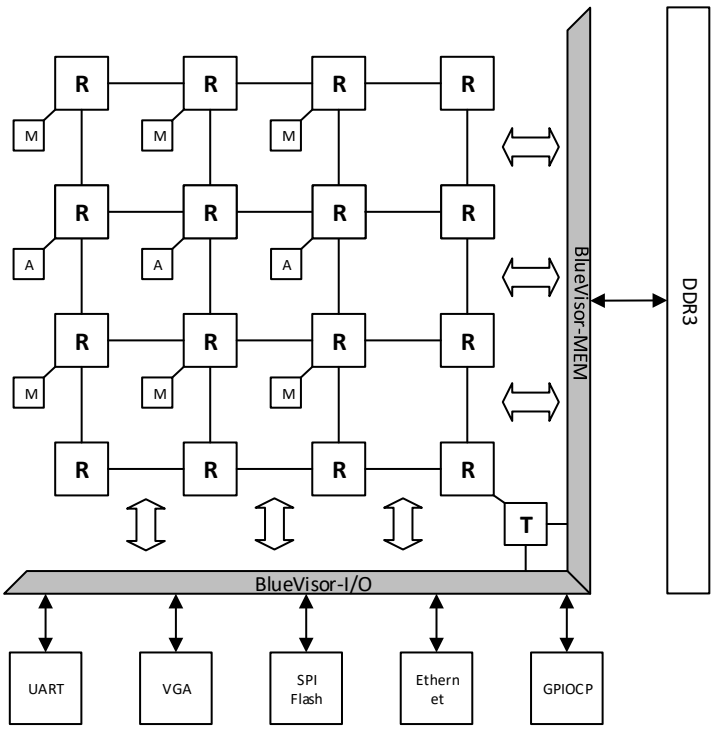


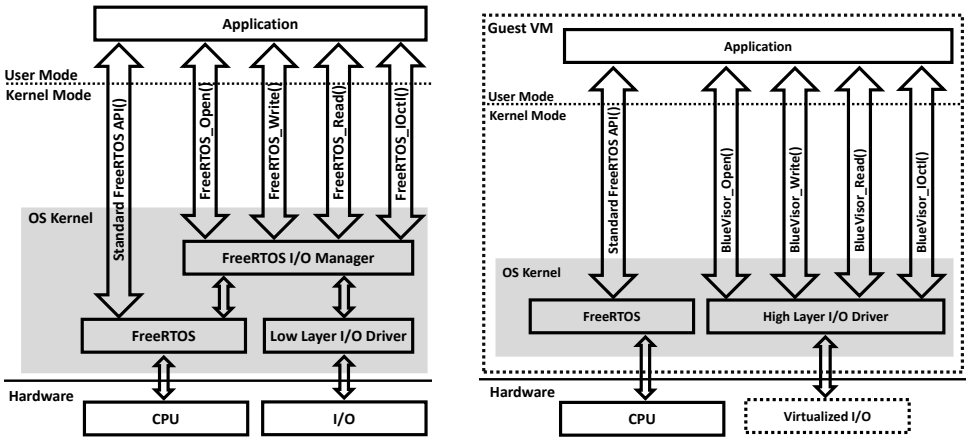
Figure 7.2: Platform Overview
M - Microblaze; A - ARM Processor;
R - Router / Arbiter; T - Global Timer

features:

- *Bare-metal virtualization [102]* - A guest OS can be executed on a processor directly, without host OS. Therefore, a guest OS is able to execute in kernel mode to achieve full privilege.

- *Para-virtualization [102]* - Parts of the guest OSs (e.g. I/O management, interrupts handler etc.) have to be replaced by our high layer drivers, which aims to achieve a smaller software, and improved performance.

Currently, in our proposed design, three OS kernels have been modified to support the virtualization [21], i.e. FreeRTOS [7], uCosII [19] and Xilkernel [29]. In Figure 7.3, we use FreeRTOS kernel as an example to demonstrate the modification of I/O parts. Compared with the original FreeRTOS kernel (Figure 7.3(a)), the user applications in the modified kernel(Figure 7.3(b)) are able to access and operate the I/O via provided high layer drivers directly.



(a) Traditional FreeRTOS Kernel (b) Modified FreeRTOS Kernel

Figure 7.3: Traditional and Modified FreeRTOS Kernels

Additionally, user applications running on the original FreeRTOS kernel can be ported to the modified kernel directly in a BlueVisor system (without any modification), since we have not modified the OS interfaces.

7.2.1.1 Timing Isolation

In our proposed design, VMs are logically isolated, which means the applications executed in one guest VM can never affect the other VMs, even if it breaks down. Moreover, isolation can be divided into spatial and temporal isolation. Specifically, with spatial isolation, a partition (i.e. VM) is completely allocated in a unique address space (e.g. code, virtual I/O resources, etc.). This address space is not accessible by other partitions (i.e. VMs). With temporal isolation, a partition (i.e. VM) is executed under a cyclic policy. The execution of partitions is not impacted by others [109]. Note that, this chapter mainly focuses on temporal isolation (see Chapter 7).

7.2.2 Memory Virtualization

Traditional hardware-assisted memory virtualization relies on Memory Management Unit (***MMU***) support for 2-level address translation, mapping a guest virtual address to a guest physical address, and then to a host physical address.

BlueVisor provides a single level mapping between individual CPU host-physical addresses and memory physical addresses using a MMU. In a BlueVi-

sor system, there are three types of physical memory allocated to each processor - local memory, individual external memory and shared external memory. The BlueVisor virtualizes these three types of physical memory to one virtual memory to each processor with continuous linear address, which is always started from *0x0000 0000*. For example, each processor in a BlueVisor system implemented on Xilinx ZC706 FPGA board [31] is allocated with 128 MB virtual memory, composed by 1 MB Block RAMs, 63 MB independent DDR3 and 64 MB shared DDR3, see Figure 7.4.

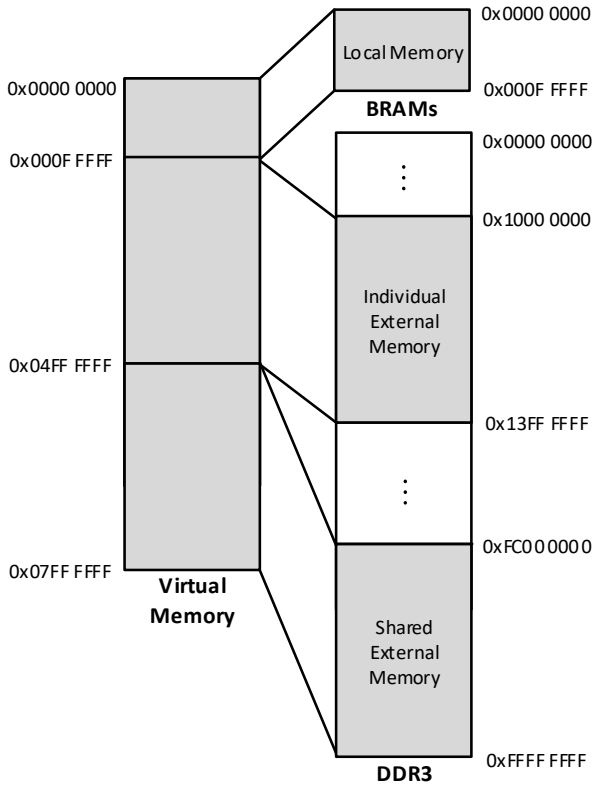


Figure 7.4: Memory Configuration

Memory segments can be configured with a specific granularity depending on the number of CPUs in the system, which enables good scalability. In addition, the memory address translation is calculated based on the CPU ID, which can be completed at fixed 1 clock cycle. As described in [62] [60] and [104], the response time of memory access provided by BlueTree is predictable. Therefore, the virtualized version BlueTree (with address translation added) is also predictable.

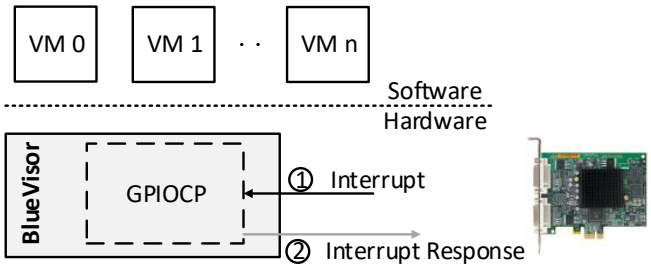
7.2.3 I/O Virtualization

VCDC inside BlueVisor virtualizes a physical I/O to multiple virtual I/Os, and provides high level access to the guest VMs (see Chapter 4). This hardware feature allows the partition of devices between guest VMs enforcing isolation at the device level, as well as shorter I/O access paths from guest VMs (bypassing guest OS kernel, VMM and host OS). In addition, VCDC also integrates low layer I/O drivers, which decreases software overhead significantly [72]. The I/O access path is shown in Figure 6.3(b).

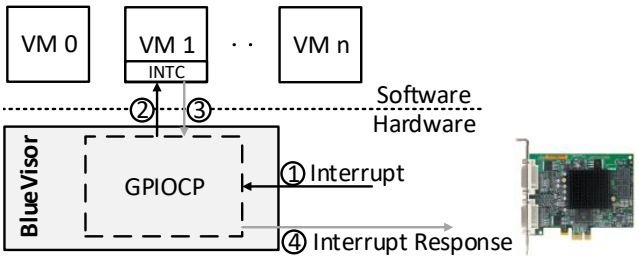
Clock cycle level timing-accurate I/O operations can be achieved by connecting the GPIOCP [120]. In [120], we have shown that deployment of GPIOCP can guarantee the clock cycle level granularity of I/O operations. In BlueVisor system, GPIOCP is integrated as an I/O controller to VCDC, in order to achieve both I/O virtualization and cycle level timing-accurate I/O operations.

7.2.4 Interrupt Management

We build two types of interrupt management in the BlueVisor system based on GPIOCP (see Figure 7.5):



(a) Type 1: Fast Interrupt Handler



(b) Type 2: Normal Interrupt Handler

Figure 7.5: Two Types of Interrupt Handlers in BlueVisor System

- Type 1: Fast interrupt handler[1] — User applications in different guest VMs are allowed to pre-program GPIOCP to respond and handle to an interrupt without sending it back to the software layer.
- Type 2: Normal interrupt handler — If the interrupt handler has not been pre-programmed, the GPIOCP will send the interrupt back to the guest VM according to its provided CPU ID.

7.2.5 Inter-VM Communication

Inter-VM communication is achieved via shared memory and interrupts. Specifically, 64 MB shared memory is allocated to each processor (described in Section 7.2.2) and is used as a communication buffer. Additionally, an interrupt is used as the notification of the occurrence of inter-VM communication among two VMs.

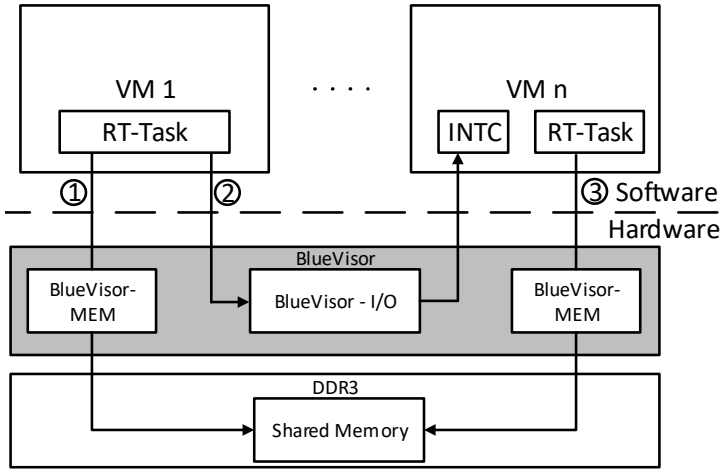


Figure 7.6: Inter-VM Communication

As shown in Figure 7.6, an inter-VM communication can be divided into three steps. Firstly, the sender guest VM writes the communication message into the shared memory. Afterwards, the sender guest VM generates an interrupt to the receiver guest VM, notifying the occurrence of the inter-VM communication. Finally, the receiver guest VM reads the communication message from the shared memory.

[1]In this chapter, the fast interrupt handler is different from the Fast Interrupt Request (***FIQ***) in the ARM architecture.

Different from the traditional inter-core communication in NoC-based many-core systems [95] (relying on on-chip communication), our proposed communication model reduces on-chip communication traffic significantly (only an interrupt required to be transferred between guest VMs). In addition, because of the predictable memory access (see Section 7.2.2) and fast interrupt handler (see Section 7.2.4) provided by BlueVisor, the predictability of the inter-VM communication can be also guaranteed.

7.3 Evaluation

The BlueVisor was implemented using Bluespec [4] and synthesised for a Xilinx ZC706 development board [31] (further implementation details are given in Appendix A and B).

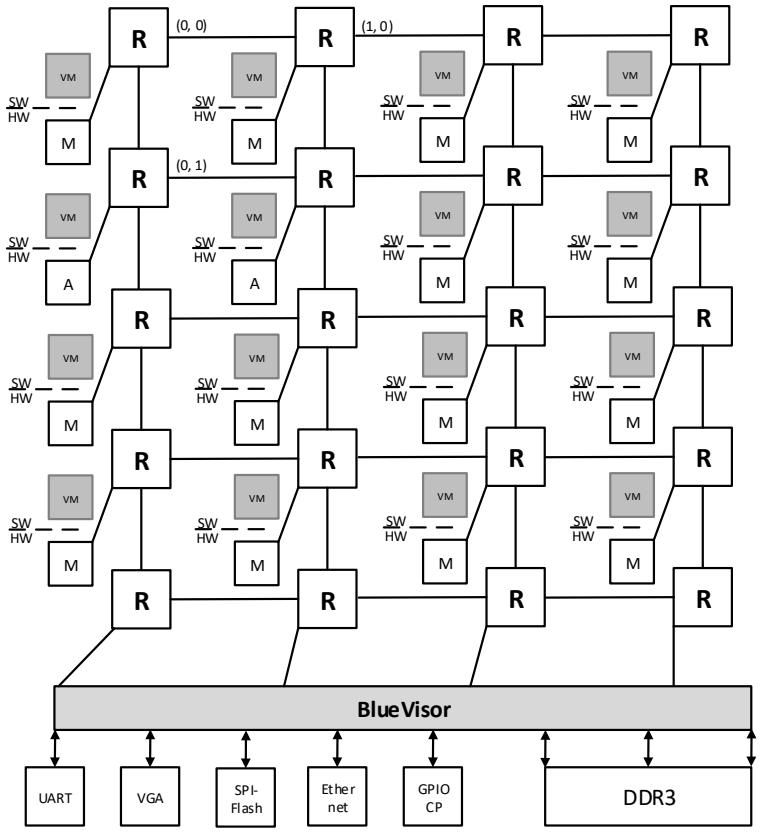


Figure 7.7: Experimental Platform
(M - Microblaze; A - ARM Processor;
VM - Guest VM; R - Router / Arbiter)

In the evaluation, the BlueVisor system is connected to a 4 x 5 2D mesh type open source NoC [95] containing 14 Microblaze CPUs [11] and 2 ARM Cortex-A9 CPUs [3], running the modified guest OS (kernel version, FreeRTOS v9.0.0) in the guest VM (described in Section 7.2.1). The architecture is shown in Figure 7.7. The software on Microblaze CPUs and ARM CPUs are compiled using the Xilinx Microblaze GNU tool chain [11] (version 5.2), and ARM GNU tool chain [3] (version 5.5) respectively. To enable comparison, a similar hardware architecture without the BlueVisor system was built. Both architectures run at 100 MHz.

Our evaluation focused on four metrics: 1) memory footprint, 2) hardware overhead, 3) I/O performance, and 4) interrupt latency.

7.3.1 Memory Footprint

In this section, we evaluate the memory footprint of BlueVisor, as well as different versions of FreeRTOS running on Microblaze CPU, via the size tool of the Xilinx Microblaze GNU Tool chain. In the measurement, the native version of FreeRTOS (*nFreeRTOS*) is full-featured [7], which is the foundation of the other versions [2] [3] [4]. Table 7.1 presents the collected measurements.

As it can be seen, there is no memory overhead introduced by the hypervisor, resulting from its pure hardware implementation. The native full-featured FreeRTOS (*nFreeRTOS*) requires 158, 741 bytes, with I/O module added, the memory footprint increases 37.18%, owing to the addition of I/O manager and I/O drivers. When it comes to the *vFreeRTOS + I/O*, the introducing of software implemented virtualization increases the memory footprint to 227, 888 bytes. However, the *BV_vFreeRTOS + I/O* only consumes 169, 424 bytes memory footprint, which is 6.73% increased compared to the native FreeRTOS, as well as 77.81% and 74.35% of the *nFreeRTOS + I/O* and *vFreeRTOS + I/O*, respectively. The main reason behind such a low memory footprint is the implementation of para-virtualization (described in Section 7.2.1), which has removed the software overhead significantly.

[2] *FreeRTOS + I/O* involves UART, VGA and corresponding drivers.

[3] *vFreeRTOS* is a simply implemented software virtualized FreeRTOS for many-core systems, see [72].

[4] *BV_vFreeRTOS* is the virtualized FreeRTOS in BlueVisor system.

Table 7.1: BlueVisor Memory Footprint (Bytes)
(I/O: UART + VGA)

Software	Memory Footprint			
	.text	.data	.bss	Total
BlueVisor	0	0	0	0
nFreeRTOS	121,309	1,728	35,704	158,741
nFreeRTOS + I/O	179,652	1,852	36,250	217,754
vFreeRTOS + I/O	189,556	1,882	36,450	227,888
BV_vFreeRTOS + I/O	131,969	1,732	35,723	169,424

7.3.2 Hardware Consumption

In this section, we use Vivado (v2016.2) to synthesize and implement BlueVisor on Xilinx ZC706 FPGA board [31], with increased number of I/Os and CPUs respectively, which aims to demonstrate the hardware consumption of BlueVisor scaled in RTL level (i.e. LUTs, registers, BRAMs, power and maximum working frequency).

As shown in Table 7.2 and 7.3, DSP slices are not required by the implementation of BlueVisor on FPGA. Additionally, the number of LUTs and registers increase linearly in the number of I/Os and CPUs respectively. Note that the increased hardware consumption leads to the linear increment in power consumption and a decrease in maximum working frequency.

The resource efficiency of BlueVisor is also shown by the tables, e.g. a full featured 2-CPU BlueVisor only consumes 2.24% LUTs and 1.04% registers of the ZC706 FPGA board; a 64-CPU BlueVisor (with GPCIOCP mounted) consumes 2.008% LUTs, 0.975% registers and 10.9% BRAMs of the ZC706 FPGA board respectively.

Table 7.2: Hardware Consumption of 2-CPU BlueVisor with Different I/Os on FPGA (RTL Level)

Added I/O	Hardware Consumption								Power (mW)	Maximum Frequency (Mhz)
	LUTs	% of ZC706	Register	% of ZC706	BRAMs	% of ZC706	DSP	% of ZC706		
+ UART	2,192	0.12%	1,471	0.17%	0	0%	0	0%	13	221.8
+ VGA	4,566	0.51%	2,315	0.27%	0	0%	0	0%	19	221.8
+ SPI Flash	6,120	1.41%	4,225	0.49%	0	0%	0	0%	29	221.8
+ Ethernet	9,723	2.24%	9,035	1.04%	0	0%	0	0%	75	192

Table 7.3: Hardware Consumption of BlueVisor (+GPIOCP) with Different Number of CPUs on FPGA (RTL Level)

Number of CPUs	Hardware Consumption								Power (mW)	Maximum Frequency (Mhz)
	LUTs	% of ZC706	Register	% of ZC706	BRAMs	% of ZC706	DSP	% of ZC706		
1	632	0.146%	962	0.111%	16	1.09%	0	0%	19	318
2	886	0.205%	1,156	0.113%	16	1.09%	0	0%	20	303
4	1,314	0.303%	1,468	0.169%	16	1.09%	0	0%	22	291
8	1,942	0.448%	2,094	0.242%	16	1.09%	0	0%	25	284
16	3,236	0.747%	3,346	0.386%	16	1.09%	0	0%	31	249
32	5,065	1.169%	5,311	0.613%	16	1.09%	0	0%	37	236
64	8,698	2.008%	8,449	0.975%	16	1.09%	0	0%	50	204

7.3.3 Real-time Features

This experiment aims to evaluate the predictability and timing accuracy of the I/O operations in a BlueVisor and a non-BlueVisor system.

This experiment aims to evaluate the predictability and timing accuracy of the I/O operations in a BlueVisor and a non-BBlueVisor system. As introduced in Equation 3.5, Chapter 3, Section 3.3.2, smaller E implies a higher timing-accuracy of the I/O operation. If E equals to 0, this I/O operation occurs at the expected time - i.e. totally timing-accurate. Moreover, a smaller variance of E implies more predictability of I/O operations.

In this evaluation, we evaluate the timing accuracy of the I/O operations in a BlueVisor and a non-BlueVisor system. In both architectures, 9 CPUs (7 Microblaze CPUs and 2 ARM CPUs) are active, whose coordinates are from (0, 0) to (0, 2), (1, 0) to (1, 2) and (2, 0) to (2, 2). When CPUs are required to access and read the GPIO at a specific time, then for a non-BlueVisor architecture the CPU has to instigate the I/O operation, for the BlueVisor architecture, this can be delegated to the BlueVisor (GPIOCP) to achieve timing accuracy. This is shown by connecting a timer to the GPIO (updating its value every cycle), with every CPU needing to read the value simultaneously.

In this evaluation, we evaluate the timing accuracy of the I/O operations in a BlueVisor and a non-BlueVisor system. In both architectures, 9 CPUs (7 Microblaze CPUs and 2 ARM CPUs) are active, whose coordinates are from (0, 0) to (0, 2), (1, 0) to (1, 2) and (2, 0) to (2, 2). When CPUs are required to access and read the GPIO at a specific time, then for a non-BlueVisor architecture the CPU has to instigate the I/O operation, for the BlueVisor

architecture, this can be delegated to the BlueVisor (GPIOCP) to achieve predictability and timing accuracy. This is shown by connecting a timer to the GPIO (updating its value every cycle), with every CPU needing to read the value simultaneously.

The result of 1,000 experiments is given in Table 3.4 and 5.1, showing that the latencies and variances for the non-BlueVisor architecture (baseline system) are significant (errors calculated according to Equation 3.5); in contrast, the BlueVisor architecture is accurate at the cycle level with good predictability, similar to the GPIOCP architecture evaluated in Chapter 5. This results from the employment of the real-time I/O controller (i.e. GPIOCP).

Note that, in the experiments, the maximum resolution of the timer is 10 ns. Therefore, while the measured E is less than 10 ns (1 clock cycle), we conclude that I/O operations exhibit high timing accuracy.

7.3.4 I/O Performance

The I/O performance evaluation is split into two different test case scenarios: 1) I/O response time, and 2) I/O throughput.

7.3.4.1 I/O Response Time

This experiment is designed to evaluate the I/O response time while CPUs and the evaluated I/Os are fully loaded in a BlueVisor and non-BlueVisor system. In both architectures, all the active CPUs have an independent application that is set to be running, which continuously reads data from an SPI NOR-flash (model: S25FL128S). Specifically, the experiment is divided into four groups, depending on the number of reading bytes: 1, 4, 64 and 256 bytes. All experiments are implemented 1,000 times and recorded in tables. A lower I/O response time indicates a higher I/O performance. We name the experiments according to the global scheduling policy and bytes of read data in once I/O request. For example, *non-BlueVisor-RR-4B* stands for a non-BlueVisor system with Round-Robin global scheduling policy; and 4 bytes of data read from the NOR-flash in once I/O request.

In the non-BlueVisor architecture, we modify the FreeRTOS to be suitable for many-core systems[5]. In both architectures, while the user applications on different CPUs are requesting the I/O at the same time instant, the scheduling

[5]FreeRTOS is designed for a single-core system; in our experiments, we modify it to be suitable for many-core systems [72]

policy can be set as FIFO (non-BlueVisor-FF and BlueVisor-FF) and Round-Robin (non-BlueVisor-RR and BlueVisor-RR), respectively. A summarized version of experimental results showing the worst case and variation of each group of experiments are demonstrated in Table 6.6 and 6.7.

Because the I/O virtualization is achieved via the VCDC inside BlueIO, the evaluation results in this chapter are essentially the same as the results in Chapter 4. The worst case response time of I/O requests in the non-BlueVisor architecture is significantly high for the reading of 1, 4, 64 or 256 byte(s) from the NOR-flash, especially while Round-Robin scheduling policy being employed. In experiments where the number of read bytes is increased, the BlueVisor system maintains its superior performance. Additionally, when it comes to the variation, BlueVisor systems always have a better performance than the non-BlueVisor systems. For example, in the non-BlueVisor-FF-1B, the variation is greater than $1,500$ clock cycles; and in non-BlueVisor-RR-1B, the variation reaches $60,000$ clock cycles. Conversely, in both BlueVisor-FF-1B and BlueVisor-RR-1B, the highest variance is less than 60 clock cycles.

Therefore, the evaluation results reveal that a system with BlueVisor provides more predictable I/O operations with lower response time.

7.3.4.2 I/O Throughput

We evaluate the I/O throughput in two architectures (with BlueVisor and without BlueVisor). In the experiments, we use the same NOR-flash illustrated in the previous section as our tested I/O. Additionally, the scheduling policy in both architectures is set as FIFO and Round-Robin respectively.

In both architectures, an independent application is set to be running on each of 4 CPUs, (3 Microblaze CPUs and 1 ARM CPU, whose coordinates are from (0,1) to (0,3)) and continuously writing to the NOR-flash - one byte can be written in one I/O request. We record the bytes written from each CPU per second as the I/O throughput (unit: KB/s). The result of higher I/O throughput implies a better performance. All the evaluations are implemented 1,000 times. The evaluation results are shown in Figure 7.8.

As demonstrated in Figure 7.8, four groups of bar charts present the average I/O throughput in the BlueVisor system and the non-BlueVisor system; and the error bar on each bar chart presents the variance of the I/O throughput during these 1,000 experiments. As shown, on all CPUs considered, no matter which scheduling policy deployed, the BlueVisor system always provides a

better performance on I/O throughput (nearly 7 times), and less variance.

7.3.5 Interrupt Handling

In this section, we evaluate the response time of interrupt handling in BlueVisor and non-BlueVisor architectures. In the measurements, a fixed interval timer [28] is programmed to send an interrupt to the active Microblaze CPU, with coordinate (0, 0), at a fixed frequency. We recorded and measured the amount of time elapsed between interrupt occurrence and conclusion of the handling routine. The 1,000 times experimental results are shown in Table 7.4.

Table 7.4: Interrupt Handling (Unit: Clock Cycles)

	Best Case	Worst Case	Mean
Native FreeRTOS	520	652	577
BS_vFreeRTOS (Fast IRQ)	10	10	10
BS_vFreeRTOS (Normal IRQ)	544	682	592

As shown, in 1,000 times experiments, fast IRQs in the BlueVisor system

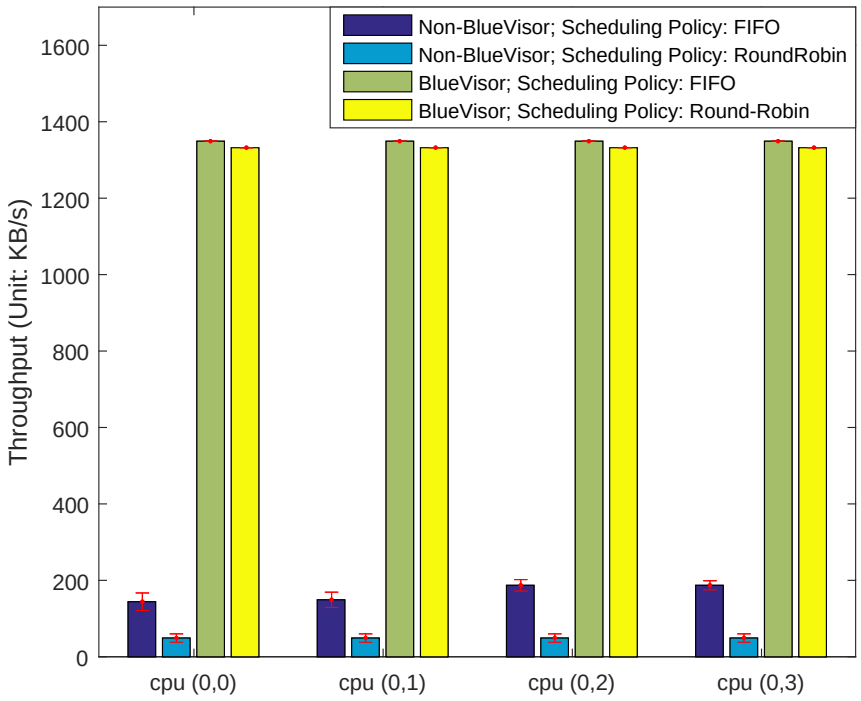


Figure 7.8: I/O Throughput

can be always completed in 10 clock cycles, which is fast and predictable. However, the response time of normal IRQs in a BlueVisor system is a little higher than the IRQs in the native FreeRTOS (including best case, worst case and mean value), mainly resulting from the need for a more complicated interrupt handler, ie. to deal with the complicated interrupt handling path (See Figure 7.5(b)).

7.4 Limitations of BlueVisor

BlueVisor requires significant hardware overhead (noting the gained predictability) as the implementation of BlueVisor is completely in hardware. Also, there is significant communication overhead at the communication interfaces of BlueVisor (Section 7.2): as all requests sent from Guest VMs have to be handled via the communication interfaces. With the number of processors increasing, some requests will be blocked and then may miss their deadlines, when the system reaches the maximum capacity of BlueVisor. The maximum capacity can be determined by different factors, e.g. the number of processors, the types of I/O devices, etc. The analysis of the maximum capacity of BlueVisor and the worst case of the system remains for future work (see Chapter 8.2). Furthermore, the hardware implementation of BlueVisor restricts the upgrade of the virtualization logic and I/O drivers. In FPGA-based systems, the hardware can be changed. However, this hardware implementation may be restrictive for production systems (i.e. VLSI).

7.5 Summary

In this chapter, we have presented the concept of predictable hardware hypervisor (VMM) for NoC many-core systems — BlueVisor. It enables a guest OS running in kernel mode to achieve full privilege (CPU virtualization), as well as predictable memory virtualization, I/O virtualization, faster interrupt handler, and inter-VM communication.

Evaluation reveals that BlueVisor can achieve virtualization with significant performance improvements, including reduced memory footprint, improved real-time features (i.e. predictability and timing-accuracy), performance features (i.e I/O performance and scalability), predictable fast IRQ. When it comes to the system overhead, the BlueVisor represents a trade-off

between software and hardware, decreasing the software usage but requiring a greater consumption of hardware.

The major contributions detailed in the chapter follow. Firstly, Section 7.1 proposed BlueVisor including design and implementation of BlueVisor using ready-built hardware components (i.e. GPIOCP, VCDC and BlueIO). Secondly, Section 7.2 introduced the specific design and implementation details of BlueVisor, with the virtualization of CPU, memory, I/O, as well as interrupt management and internal-VM communication described from Section 7.2.1 to Section 7.2.5, respectively. Thirdly, Section 7.3 evaluated the BlueVisor with multiple metrics. Specifically, Section 7.3.1 evaluated the memory footprint of BlueVisor and BlueVisor-based systems. Because of the hardware implementation of BlueVisor, the software overhead in a BlueVisor-based system is significantly lighter than a conventional solution. Then, Section 7.3.2 evaluated the resource efficiency of BlueVisor when implemented on the Xilinx ZC706 FPGA development board. The introduction of BlueVisor introduced extra hardware overhead. However, the BlueVisor occupies at most 1% of the FPGA board (in a 64-core system). Furthermore, Section 7.3.3 evaluated the real-time features of BlueVisor by measuring the error in timing-accuracy (E) of two architectures and corresponding variances (with and without BlueVisor). The evaluation results reveal that a BlueVisor-based system can always handle multiple I/O devices with clock cycle timing-accuracy and predictability. Moreover, Section 7.3.4 evaluated the I/O performance via I/O throughput and I/O response time. The evaluation results reveal that BlueVisor significantly enhances the I/O performance compared to a non-BlueVisor architecture — increased I/O throughput and reduced I/O response time. Then, Section 7.3.5 evaluated the response time to handle an external interrupt. The evaluation resulted in Table 7.4 showing that fast IRQs in the BlueVisor system can be always completed at 10 clock cycles — fast and predictable. Finally, Section 7.4 discussed the current drawback of BlueVisor, including extra hardware overhead, traffic congestion at its interface and difficulties in upgrade (which on IC rather than FPGA).

The design and implementation of BlueVisor evidence that our ready-built hardware components can be applied and expanded into different architectures and platforms, i.e. VCDC (see Chapter 4), GPIOCP (see Chapter 5), and BlueIO (see Chapter 6). The evaluation results demonstrate that the expanded system can also maintain lower software overhead, better real-time

features (i.e. predictability and timing-accuracy), performance feature (i.e. I/O performance and scalability) and protection features (parallel accesses and isolation).

Chapter 8

Conclusion and Future Work

This thesis has proposed a real-time I/O virtualization system for multi-core and many-core embedded systems, where the I/O system simultaneously enables the following features (the proposed I/O system is architecture agnostic, and can be easily applied to different architectures with a various number of processors):

- **Performance features:**
  - Enhanced I/O performance;
  - Scalability.
- **Real-time feature:**
  - Predictability;
  - Timing-accuracy.
- **Protection feature:**
  - Parallel accesses;
  - Isolation.

The research questions described in Chapter 1, Section 1.2, 1.3 and 1.4 have all been answered and discussed below, respectively.

> *Research Question 1: How can I/O performance in real-time systems be enhanced by an increased number of cores? (compared to a traditional system)*

Chapter 4 proposes a hardware-implemented I/O virtualization system — i.e. Virtualized Complicated Device Controller (***VCDC***). VCDC enables user applications to access and operate I/O devices directly from a guest VM, bypassing the guest OS, the VMM, and low layer I/O drivers.

The evaluation results in Section 4.3 demonstrate that VCDC is able to virtualize a physical I/O device to multiple virtual I/O devices with significant performance improvements compared to baseline systems (i.e. I/O performance and scalability), containing shorter I/O response time, greater I/O throughput, and less on-chip communication overhead. Chapter 4 has satisfied the success criteria SC-1.

> *Research Question 2: Apart from performance features, how can the predictability and timing-accuracy of I/O operations in multi-core and many-core real-time systems be guaranteed?*

Chapter 5 has proposed a resource efficient programmable I/O controller, termed the GPIO Command Processor (***GPIOCP***). GPIOCP permits applications to instigate complicated sequences of I/O operations at an exactly specific clock cycle, so good real-time features (i.e. predictability and timing-accuracy). Moreover, the I/O operations can be programmed to occur at some point in the future, periodically, or reactively.

The evaluation results in Section 5.4 provide evidence that GPIOCP can handle multiple I/O operations with predictability and clock cycle accuracy. Furthermore, its hardware overhead was 50% less compared to a tested with the same functionality build using a minimalistic version of the soft core microprocessor; Microblaze instead of GPIOCP. Chapter 5 has satisfied the success criteria SC-2.

> *Research Question 3: How can performance features and real-time features for I/O systems be achieved when I/O virtualization is deployed (to achieve protection features)?*

Chapter 6 has integrated GPIOCP (see Chapter 5) and VCDC (see Chapter 4) as a real-time I/O virtualization system, termed BlueIO.

The evaluation results in Section 6.4 demonstrate that BlueIO inherits

the benefits brought by GPIOCP and VCDC. Specifically, the deployment of VCDC enables enhanced performance features, including faster I/O response time, greater I/O throughput, good scalability and less on-chip communication overhead. Moreover, the employment of GPIOCP enables BlueIO achieving real-time features while handling multiple I/O operations in parallel, both predictability and clock cycle timing-accuracy. Furthermore, the implementation of I/O virtualization brings significant protection features to the whole system — i.e. parallel accesses and isolation. Chapter 6 has satisfied the success criteria SC-3.

> Research Question 4: How to integrate the ready-built I/O system to the complete system with the expected features inherited?

Chapter 7 establishes a scalable real-time hardware hypervisor for multi-core and many-core embedded architectures, termed BlueVisor, which is built upon GPIOCP, VCDC and BlueIO. BlueVisor enables predictable virtualization on CPU, memory, and I/O, as well as fast interrupt handler, and inter-VM communication. The establishment of BlueVisor aims to show our methodologies can be applied and expanded to different architectures and platforms, with maintained features on real-time, performance and protection — evidenced by the evaluation results in Section 7.3. Chapter 7 has satisfied the success criteria SC-4.

These research questions have been resolved, and demonstrated the thesis hypothesis (stated in Section 1.5)

> ***Effective real-time I/O and virtualization can be achieved by moving virtualization, I/O drivers and I/O operations into hardware.***
>
> ***The thesis will show that moving the virtualization layer and I/O drivers from software layer to hardware layer significantly increases I/O performance compared to traditional virtualized and non-virtualized systems. Also, it will show that a programmable I/O controller contained in the virtualization system permits applications to instigate complex sequences of I/O operations at an exact time (the output values can be both***

static and dynamic), so achieving timing-accurate and predictable I/O operations with I/O virtualization.

Moreover, The design of the real-time I/O virtualization system is generic, which can be ported to different platforms with a scaled number of processors and I/O devices. Therefore, it can be directly applied to a real-time system, with the inherited performance features, real-time features and protection features.

8.1 Major Contributions and Key Findings

This section summarises major contributions and key findings in the thesis. The findings are grouped under four headings: VCDC, GPIOCP, BlueIO and BlueVisor.

Chapter 4: VCDC

The VCDC proposed within Chapter 4 enables:

- ***Better Performance Feature (compared to baseline systems)*** — Includes the lower response time of I/O operations and higher I/O throughput.
- ***Good Scalability (Performance Feature)*** — We propose a distributed implementation. When the VCDC is employed, to add one more CPU into a system, the users are only required to add one group of dedicated CPU FIFO (see section 4.2.4.2), which provides an interface between the added CPU and the VCDC.
- ***Predictability (Real-time Feature)*** — I/O operations requested from a guest OS are more predictable than under conventional virtualization.
- ***Lower Software Overhead*** — Moves the VMM and low level I/O drivers from kernel mode (at the software level) to the VCDC.
- ***Abstracted High Layer Access*** — The user application in a guest virtual machine is able to request and operate an I/O device via invoking simple high layer drivers. For example, a user application can request

to read a series of data from an SPI-Flash by sending a request with parameters to the VCDC: "*Read SPI-Flash* (instruction), from the *start address* to the *end address* (parameters)".

- ***Global Arbitration*** — We propose a modularized implementation, whereby the scheduling policy of the arbiter can be switched easily between round robin, fixed priority and customized scheduling policies [21].

The design and implementation of VCDC have successfully answered the research question 1.

Chapter 5: GPIOPCP

The GPIOCP proposed within Chapter 5 enables:

- ***Predictability and Timing-accuracy (i.e. Real-time Features)*** — All I/O operations over the GPIO pins can be predictably issued with an accuracy of a single cycle.
- ***Programmability*** — The GPIOCP holds small programs designed to control connected devices. They are loaded into GPIOCP memory by the application during system initialisation (so that loading does not interfere with normal execution and timeliness of the system). Importantly, commands within the program can be executed at exact times (cf. conventional CPU instructions).
- ***Parallel Controls*** — Multiple I/O devices connected to the GPIO pins can be controlled in parallel, whilst maintaining predictability and timing-accuracy of a single clock cycle.

The design and implementation of GPIOCP have successfully answered the research question 2.

Chapter 6: BlueIO

Chapter 6 proposes the design and implementation details of BlueIO, achieving the following contributions:

- A scalable hardware-implemented real-time I/O virtualization system, with the following features:

1. ***Parallel Accesses and Isolation (i.e. Protection Features)*** — BlueIO enables I/O virtualization, so that I/O operations requested from different VMs are isolated, and able to access different I/O devices simultaneously.
2. ***Predictability and Timing-accuracy (i.e. Real-time Features)*** — BlueIO integrates the real-time timing-accurate I/O controller GPIOCP, to enable predictable and timing-accurate I/O operations, whilst maintaining isolation and parallel accesses.
3. ***Enhanced I/O Performance (i.e. Performance Feature)*** — BlueIO integrates I/O drivers, and provides abstracted high-layer access interfaces to software (Guest VMs), which simplify the I/O access paths, and improve I/O performance.

- A hardware consumption analysis of BlueIO, in order to show that it is linearly scaled by the number of CPUs and I/O devices respectively.

The design and implementation of BlueIO have successfully answered the research question 3.

Moreover, the hardware consumption analysis has implied that its hardware consumption is linearly scaled with the number of processors and I/O devices respectively.

Chapter 7: BlueVisor

The BlueVisor proposed within Chapter 7 is a hardware hypervisor, which enables:

- ***Predictable and Timing-accurate Virtualization (i.e. Real-time Features)*** — BlueVisor enables virtualization with real-time features, i.e. CPU virtualization, memory virtualization, I/O virtualization.
- ***Improved I/O Performance (i.e. Performance Feature)*** — Due to the integration of I/O drivers (in hardware layer), the I/O response time is reduced and the I/O throughput is increased.
- ***Fast and Predictable Interrupt Handling*** — Due to the deployment of GPIOCP, the fast IRQ in BlueVisor can always handle interrupts at a fixed time.

The design and implementation of BlueVisor have successfully answered the research question 4. It verifies that our contributions (i.e. GPIOCP, VCDC and BlueIO) can be applied and expanded to different system architectures and platforms, with kept performance features, real-time features and protection features.

8.2 Future Work

There are several possible areas of future research based on the work presented in the thesis.

8.2.1 Supporting SMP OS

Currently, with our proposed components (i.e. GPIOCP, VCDC, BlueIO and BlueVisor), one guest VM is always mapped to only one core. This implies that one user application is not able to utilize more than one core simultaneously. In order to overcome this drawback, we are considering a new architecture among multiple cores, a router and a guest OS (see Figure 8.1).

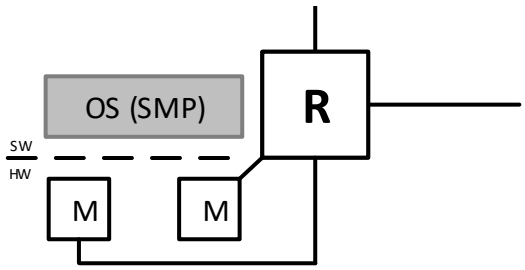


Figure 8.1: Supporting SMP OS (M - Microblaze; R - Router / Arbiter)

With the new architecture, multiple processors can be used to support an SMP OS. However, the development of this new architecture is still under progress.

8.2.2 Timing Analysis — Hard Real-time

As reviewed in Section 2.1.2, two commonly used methodologies are adopted to achieve the WCET of tasks — static analysis and a measurement-based

analysis.

In this thesis, we only adopted measurement-based analysis to derive the WCET of the I/O operations, including in Section 5.4.1, 4.3.1, 4.3.1, 6.4.2, 7.3.3. However, in order to make the analysis be sound, the worst-case path of the program and the worst-case conditions of the system have to be accurately found. However, it is difficult to determine the worst-case conditions via observing, even though all the experiments have been repeated 1,000 times. Hence, our systems can not be directly fitted in a hard real-time system.

In order to solve this issue and make sure our system is hard real-time, we are considering to figure out the WCET of I/O operations in our systems via schedulability analysis based on [36] and [70]. Once the WCET is found, our methodologies can be fitted in a hard real-time system.

8.2.3 Supporting More I/O Drivers

As described in Section 2.2, the number of currently popular I/O devices is countless. Due to the time limit, only a few types of I/O devices are supported in the thesis.

In the future work, a number of complicated I/O devices are proposed to be supported, e.g. USB hot devices.

8.3 Closing Remarks

In modern real-time embedded systems, I/O operations often simultaneously require performance features (i.e. enhanced I/O performance and scalability), real-time features (i.e. predictability and timing-accuracy), and protection features (i.e. parallel accesses and isolation).

In this thesis, we have proposed a scalable hardware-implemented real-time I/O system for multi-core and many-core systems — BlueIO, which satisfies the requirements at the same time. BlueIO system integrates most of the functionalities of I/O virtualization, low layer I/O drivers (i.e. VCDC) and the clock cycle level timing-accurate I/O controller (i.e. GPIOCP) in hardware layer, meanwhile providing abstracted high-layer access interfaces to the Guest VMs in software layer.

Evaluation reveals that BlueIO can virtualize a physical I/O device to multiple virtual I/O devices with significant performance improvements, including faster I/O response time, greater I/O throughput, and good scalability. In ad-

dition, BlueIO can also handle multiple I/O operations with clock-cycle-level accuracy, in many cases totally timing-accurate and predictable. Due to the employment of I/O virtualization, I/O operations requested from different VMs are isolated, and able to access different I/O devices simultaneously.

At last, a scalable real-time hardware hypervisor is established, termed BlueVisor. It is built upon GPIOCP, VCDC and BlueIO. BlueVisor enables predictable virtualization on CPU, memory, and I/O, as well as fast interrupt handler, and inter-VM communication. The establishment of BlueVisor shows that our methodologies can be applied and expanded to different architectures and platforms, with maintained features on real-time, performance and protection.

Appendices

Appendix A

Implementing a GPIOCP/VCD-C/BlueIO/BlueVisor

This chapter mainly describes the implementation steps of our proposed components in hardware, i.e. GPIOCP, VCDC, BlueIO and BlueVisor. Because the steps of implementing the components are same, we implement GPIOCP on the Bluetile NoC in Xilinx VC709 FPGA board as an example. All the source code can be accessed via link: *https://github.com/RTSYork*.

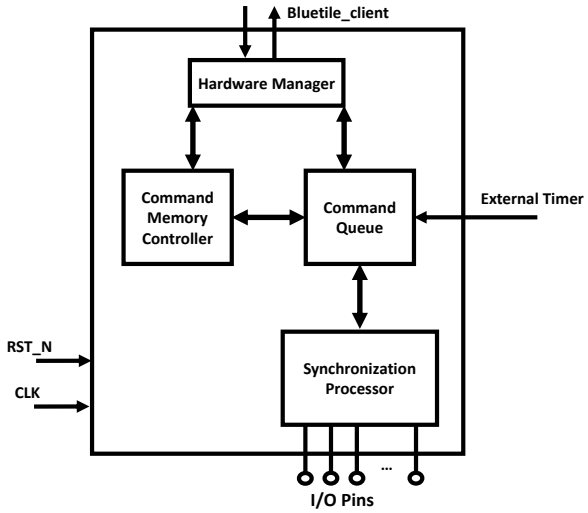


Figure A.1: Top Level Architecture of GPIOCP

GPIOCP is comprised by four modules: hardware manager, command memory controller, command queue and synchronization processor, which are implemented via *Bluespec System Verilog* [4]. The interconnected system is illustrated in Figure A.1.

Corresponding to these four modules, the source code can be found in the root folder *IP_GPIOCP* respectively: *GPIOCMD_hw_manager.bsv*, *GPIOCMD_cmd_memory.bsv*, *GPIOCMD_cmd_q.bsv* and *GPIOCMD_cmd_processor.bsv*. Users can execute the script *build.sh* in *wrap* folder to compile the source files to *verilog* files of the GPIOCP. The top level of the GPIOCP can be found as *BS_GPIOProcessor.v*. This top level of GPIOCP in VIVADO is shown in Figure A.2.

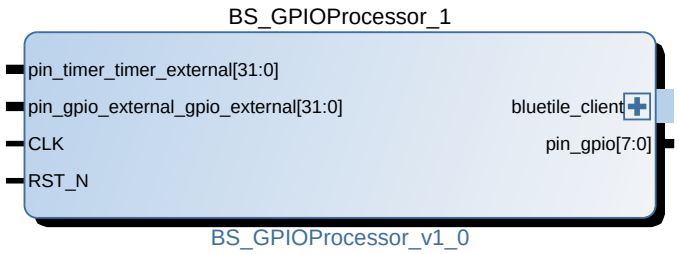


Figure A.2: The Toplevel of the IP Core - GPIOCP

As shown, the top level has 4 input ports, 1 output port and 1 system interface. Among these ports, the port *CLK* and port *RST_N* should be respectively connected to the clock source and the reset of the whole system. The port *pin_gpio_external_gpio_external[31:0]* and port *pin_gpio[7:0]* should be connected to the output and input GPIO pins of I/O devices respectively, which connects the GPIOCP and peripherals. The port *pin_timer_timer_external[31:0]* should be connected to the global timer of the whole system, whose resolution is 31-bit. This global timer provides a synchronization among GPIOCP and the whole system. Finally, the port *bluetile_client* should be connected to a router on the Bluetile system, which provides a communication interface between GPIOCP and the processors mounted on the NoC. In Bluetile NoC system, all the communication are transmitted as packets. The format of the packets follows uniform rules illustrate in Section **??**.

The steps of wrapping other components (i.e. VCDC, BlueIO and BlueVisor) are same as GPIOCP.

A.1 Generic Number of Processors

As described in the thesis, the designs of proposed components are generic (i.e. GPIOCP, VCDC, BlueIO and BlueVisor). This means the designs can be fitted into systems with a scaled number of processors, which is achieved via modifying the pre-defined macro in each top module — *numb_CPU*. *numb_CPU* indicates the number of processors in the whole system. For example, to build a GPIOCP with *9* CPUs, the macro should be modified as the following Listing.

```
1 Integer  numb_CPU = 9;
```

Listing A.1: Modifying the macro to fit a 9-core system

The methods of changing macros in the other components are same as GPIOCP.

Appendix B

Connecting GPIOCP/VCD-C/BlueIO/BlueVisor to a Bluetile Many-core System

This chapter mainly describes the steps of setting our experimental platform — Bluetile Many-core systems [95]. Blueile system is a Manhattan grid (mesh) interconnect for a network on chip (NoC) built using Bluespec System Verilog [95]. The interconnect enables a large number of CPUs and other processing elements to exchange messages in the form of network packets, the more details of Bluetile can be found in the website:
https://rtslab.wikispaces.com/Bluetiles.

Bluetile system implements a Manhattan grid interconnect. Two sorts of component are important:

- A router: Each router has five connections - each a bidirectional 32-bit channel of type "BlueBits" (defined in *Bluetiles.bsv*). Four of these are named North, East, South and West and are connected to other routers (or, at the edge of the grid, nothing at all). The fifth is named Home and connects to a local component. Each router has an address expressed in the form (x, y): these are Cartesian co-ordinates representing a grid location. The address is used when packets are routed. The router compares its own address against the destination address in a network packet, then directs the packet to one of the five interfaces accordingly.

- A local component: This could be a CPU, an I/O device, or a co-processor. It implements the other side of the Home connection, which allows it to send and receive messages over the network. GPIOCP is one of the local component.

The source code related to Bluetile systems can be found in folder *System_Bluetile*, which is accesed via the link:
https://github.com/RTSYork/GPIOCP/tree/master/System_Bluetile.

B.1 Building Bluetile system

The source code of the router and local components, e.g. UART and mutex, are written via *Bluespec System Verilog*. There are four steps are compulsory while building a Bluetile system: 1) Compiling the source code of each components to *verilog* files; 2) Encapsulating the *verilog* files as the Vivado IP cores; 3) Building the NoC via connecting routers; 4) Adding local components and connecting them on the NoC, including CPUs, UART ,etc.. The flow chart of these steps are shown in Figure B.1.

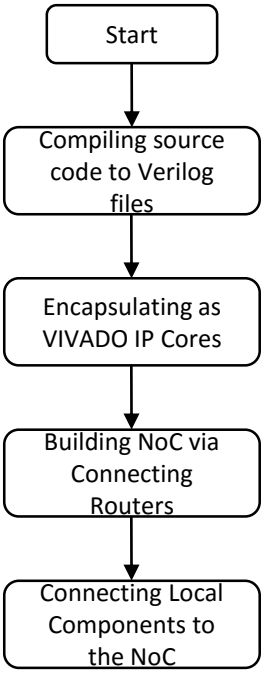


Figure B.1: Flow of Building Bluetile System

B.1.1 Compiling Bluespec System Verilog Files

To compile the source code of all the components, users can run the script *build_all.sh* located in the *System_Bluetile* folder.

B.1.2 Encapsulating Verilog Files as IP cores

Afterwards, users can run the script *launch_vivado.sh* to encapsulate the verilog files as the Vivado IP cores. After the IP cores being built, users can invoke these components in Vivado directly. The IP cores are listed in Figure B.2.

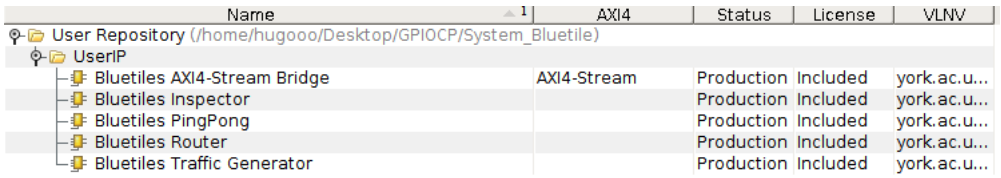


Figure B.2: Encapsulated Bluetile System IP Cores

B.1.3 Building the NoC

We provide two methods for users to build a Bluetile NoC:

- Manual Building: Invoking the routers inside Vivado and connect corresponding communication ports.
- Automatic Building: Executing the provided tcl script to build a NoC with particular size. For example:

```
1  bs::create_bluetiles_net_hier 2 3 BuleTile_NoC
```

Listing B.1: Building a 2*3 NoC via tcl script

After this script being executed, a size *2*3* NoC will be built, which named as *BlutTile_NoC*. Figure B.3 illustrates this NoC.

As it shown, the top level of a NoC has a clock signal port, a reset signal port and some home ports. Each home port belongs to a corresponding router, which can be used to connect local components.

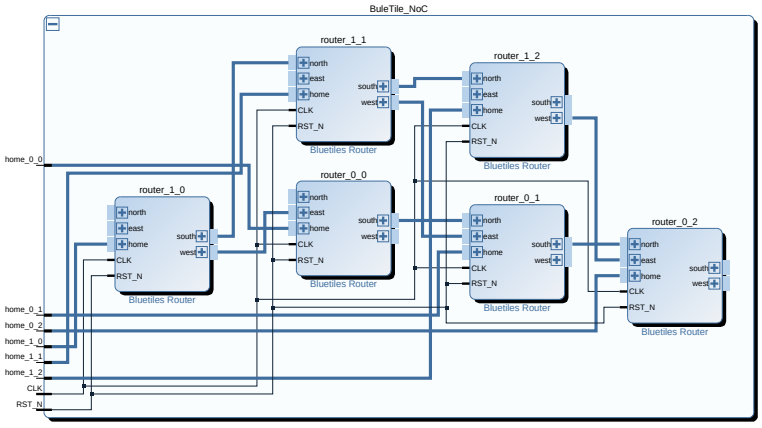


Figure B.3: Size 2*3 BlutTile NoC

B.1.4 Connecting Local Components

The communication method in Bluetile system are implemented via an communication interface provided by Bluespec System Verilog named ClientServer interface. The ClientServer interface provides two interfaces - Client interface and Server interface that can be used to define modules which have a request-response type of interface. In Bluetile system, we set the communication interfaces of all the routers are Server; and set the communication interfaces of all the local components are Client. Therefore, to connect local components, users are only required to connected the client interfaces of local components to the Server interfaces of the NoC. Figure B.4 illustrates an example of connecting the UART to the router whose coordinate is (0, 0) in the NoC.

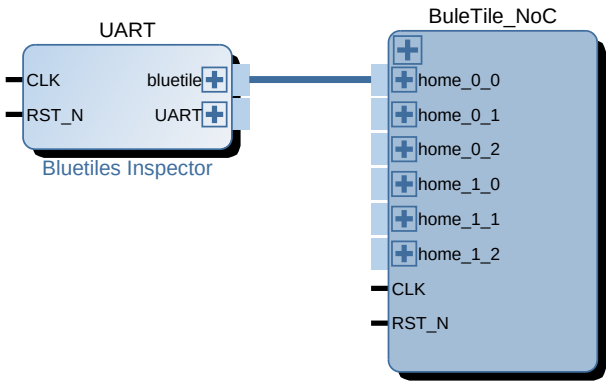


Figure B.4: Connecting an UART to the NoC

B.1.5 Building a Bluetile System with Script

In the folder *zedboard_example*, we provide a script *create_project.tcl*, which can build an example Bluetile system with commonly used IP cores.

B.2 Connecting GPIOCP/VCDC/BlueIO/BlueVisor to a Bluetile System

The methods of connecting our proposed components (i.e. GPIOCP, VCDC, BlueIO and BlueVisor) to a Bluetile system are same. In this chapter, connecting GPIOCP to a Bluetile system is demonstrated as an example.

Following the steps in Chapter A, users can build a GPIOCP shown as Figure A.2. Same as other local components, to connect a GPIOCP, users are only required to connect the Client interface of the GPIOCP to the Server interface on one of the router, which is shown in Figure B.5.

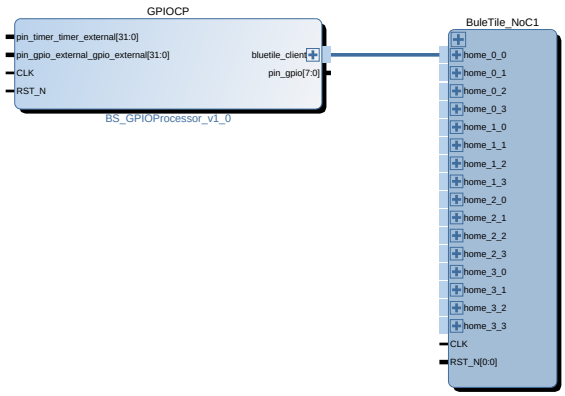


Figure B.5: Connecting the GPIOCP on the NoC

Appendix C

Running FreeRTOS/uCosII/Xilinx Kernel

As introduced in the thesis, our proposed systems enable para-virtualization. Currently, the systems support three real-time kernels: FreeRTOS, uCosII, and XilinxKernel. Due to the steps of executing the kernels are same, we introduce the steps of running FreeRTOS as an example.

In our approaches, the official version of the FreeRTOS kernel are used, which can be accessed via *http://www.freertos.org/*. Additionally, we also modify the official I/O module of FreeRTOS, the official I/O module can be accessed via
http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_IO.html.

C.1 Building BSP of FreeRTOS

In the Github, the folder *OS_bsp* stores the BSPs of different OSs: *ucos II (v1.41)* and *FreeRTOS (v9.0)*, see Figure C.1.

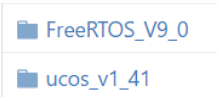


Figure C.1: BSP for different OSs

To invoke a BSP into a project, users only need to click *Xilinx Tool*, *repos-*

itories and add the BSP, see Figure C.2.

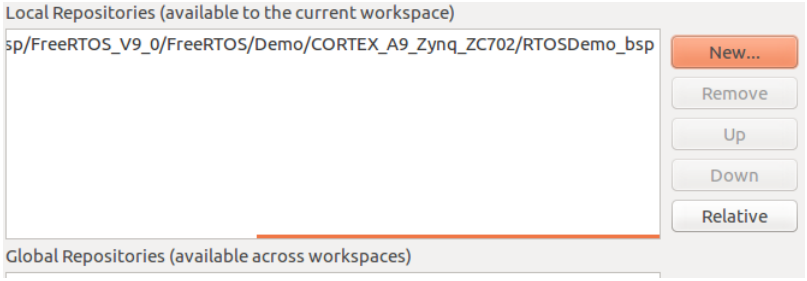


Figure C.2: Add the BSP

After that, users can build a project with FreeRTOS via using the FreeRTOS BSP.

C.2 Adding the I/O Manager

As mentioned in the thesis, an I/O manager is not required in our approaches. However, we still provide a modified I/O manager for the real-time OS kernels, which can be access in the folder *FreeRTOS-Plus-IO*, see Figure C.3.

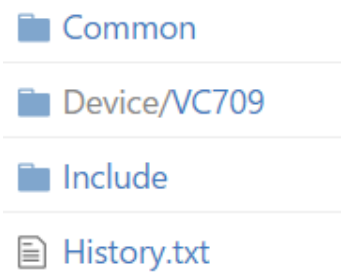


Figure C.3: I/O manager in FreeRTOS

The folder *VC709* stores the drivers for the Xilinx FPGA board VC709.

C.3 Invoking High Layer I/O Drivers

We provide the high layer I/O drivers in the folder *I/O* drivers, which is shown in Figure C.4.

Users can just invoke this high layer I/O drivers in the project directly. For example, *BS_NoC.c* includes the drivers for the BlueTile system; *BS_rtc.c*

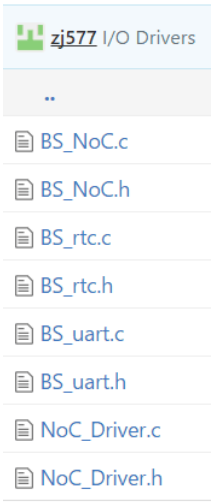


Figure C.4: I/O Drivers in FreeRTOS

contains the drivers for the real-time clock; *BS_uart.c* includes the drivers for the UART.

Bibliography

[1] https://vlsiarch.ecen.okstate.edu/flows/?C=N;O=D, note = Accessed Otc 16, 2018, title = Encounter RTL Compiler.

[2] 8080 user manual. http://altairclone.com/downloads/manuals/8080%20Programmers%20Manual.pdf. Accessed August 27, 2018.

[3] Arm cortex-a9 cpu official website. https://developer.arm.com/products/processors/cortex-a/cortex-a9. Accessed April 12, 2017.

[4] Bluespec inc. bluespec system verilog (bsv). http://www.bluespec.com/products/. Accessed September 27, 2017.

[5] Encounter rtl compiler. https://www.cadence.com/. Accessed Otc 16, 2018.

[6] Freertos i/o official website. http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_IO/FreeRTOS_Plus_IO.shtml. Accessed September 27, 2016.

[7] Freertos official website. http://www.freertos.org/. Accessed September 27, 2017.

[8] Gpiocp technical report. https://github.com/RTSYork/GPIOCP/blob/master/GPIOCP%20GPIO%20Command%20Processor%20(v1.0).pdf. Accessed August 27, 2018.

[9] Intel Knight Landing website. https://ark.intel.com/products/codename/48999/Knights-Landing. Accessed April 12, 2017.

[10] Microblaze gnu tool chain. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug1043-embedded-system-tools.pdf. Accessed September 23, 2018.

[11] Miroblaze user manual. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf. Accessed August 27, 2016.

[12] Parallella official website. https://www.parallella.org/. Accessed August 27, 2017.

[13] Reprap website. https://reprap.org/wiki/RepRap. Accessed August 27, 2018.

[14] Rtems official website. https://www.rtems.org/. Accessed August 26, 2017.

[15] Solar website. https://www.solarflare.com/ultra-low-latency. Accessed August 27, 2018.

[16] Sr-iov official website. http://pcisig.com/. Accessed September 27, 2016.

[17] Tilera official website. http://www.tilera.com/. Accessed August 27, 2017.

[18] Tpu website. http://www.nxp.com/products/microcontrollers-and-processors/power-architecture-processors/mpc5xxx-5xxx-32-bit-mcus/mpc56xx-mcus/enhanced-time-processor-unit:eTPU?uc=true&lang_cd=en. Accessed August 27, 2017.

[19] ucos official website. https://www.micrium.com/rtos/kernels/. Accessed September 27, 2017.

[20] Vc709 official website. https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html. Accessed August 27, 2017.

[21] Vcdc technical report. https://github.com/RTSYork/BlueIO. Accessed January 27, 2017.

[22] Vxworks official website. http://windriver.com/products/vxworks/. Accessed August 27, 2017.

[23] Vxworks timer library. http://www.vxdev.com/docs/vx55man/vxworks/ref/timerLib.html. Accessed August 27, 2017.

[24] x86 user manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures.pdf. Accessed January 26, 2018.

[25] Xen official website. https://www.xenproject.org/. Accessed Otc 16, 2017.

[26] Xilinx 1g/2.5g ethernet subsystem manual. https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v7_0/pg138-axi-ethernet.pdf. Accessed August 27, 2016.

[27] Xilinx axi fifo user manual. https://www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo-mm-s.pdf. Accessed August 27, 2018.

[28] Xilinx fixed interval timer website. https://www.xilinx.com/support/documentation/ip_documentation/fit_timer/v2_0/pg110-fit-timer.pdf. Accessed September 23, 2018.

[29] Xilinx official website. https://www.Xilinx.com. Accessed July 5, 2017.

[30] Xilinx official website - the introduction of axi bus. http://www.xilinx.com/products/intellectual-property/do-axi-bfm.html. Accessed July 5, 2017.

[31] Zc706 official website. https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html. Accessed April 12, 2017.

[32] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital systems testing and testable design*, volume 2. Computer Science Press New York, 1990.

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

[34] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 248–259. ACM, 2000.

[35] N. Audsley. Memory architectures for noc-based real-time mixed criticality systems. *Proc. WMC, RTSS*, pages 37–42, 2013.

[36] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[37] J. Bacon and T. Harris. *Operating systems: concurrent and distributed software design*. Pearson Education, 2003.

[38] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The vmware mobile virtualization platform: is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review*, 44(4):124–135, 2010.

[39] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn. The price of safety: Evaluating iommu performance. In *The Ottawa Linux Symposium*, pages 9–20, 2007.

[40] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time i/o management system with cots peripherals. *IEEE Transactions on Computers*, 62(1):45–58, 2013.

[41] V. Betz and J. Rose. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[42] R. Birkett. Enhancing real-time capabilities with the pru. In *Embedded Linux Conference*, 2015.

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

[44] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[45] M. Caldari, M. Conti, M. Coppola, P. Crippa, S. Orcioni, L. Pieralisi, and C. Turchetti. System-level power analysis methodology applied to the AMBA AHB bus [soc applications]. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 32–37. IEEE, 2003.

[46] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in Xen. *SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[47] P. Cousot. Abstract interpretation. *ACM Computing Surveys (CSUR)*, 28(2):324–328, 1996.

[48] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[49] D. Dimitrov, W. Van Wijck, K. Schreve, and N. De Beer. Investigating the achievable accuracy of three dimensional printing. *Rapid Prototyping Journal*, 12(1):42–52, 2006.

[50] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[51] Y. Dong, Z. Yu, and G. Rose. SR-IOV Networking in Xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, 2008.

[52] A. Duller, G. Panesar, and D. Towner. Parallel processing-the picochip way. *Communicating Processing Architectures*, 2003:125–138, 2003.

[53] T. El-Ghazawi and G. Frieder. Input-output operations. 2003.

[54] J. Engblom. *Processor pipelines and static worst-case execution time analysis*. PhD thesis, Acta Universitatis Upsaliensis, 2002.

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

[56] A. Farzadi, M. Solati-Hashjin, M. Asadi-Eydivand, and N. A. A. Osman. Effect of layer thickness and printing orientation on mechanical properties and dimensional accuracy of 3d printed porous samples for bone tissue engineering. *PloS one*, 9(9):e108252, 2014.

[57] C. Fetzer, U. Schiffel, and M. Süßkraut. An-encoding compiler: Building safety-critical systems with commodity hardware. In *International Conference on Computer Safety, Reliability, and Security*, pages 283–296. Springer, 2009.

[58] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong. Device scaling limits of si mosfets and their application dependencies. *Proceedings of the IEEE*, 89(3):259–288, 2001.

[59] M. García-Valls, T. Cucinotta, and C. Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726–740, 2014.

[60] J. Garside and N. C. Audsley. Prefetching across a shared memory tree within a network-on-chip architecture. In *ISSoC*, pages 1–4, 2013.

[61] M. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Transactions on Computers*, 2016.

[62] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A generic, scalable and globally arbitrated memory tree for shared dram access in real-time systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 193–198. EDA Consortium, 2015.

[63] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.

[64] C. D. Graziano. A performance analysis of xen and kvm hypervisors for hosting the xen worlds project. 2011.

[65] Z. Gu and Q. Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization. *Journal of Software Engineering and Applications*, 5(04):277, 2012.

[66] G. Heiser and B. 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.

[67] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[68] R. Hiremane. Intel virtualization technology for directed i/o (intel vt-d). *Technology@ Intel Magazine*, 4(10), 2007.

[69] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia. I/o scheduling model of virtual machine based on multi-core dynamic partitioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 142–154. ACM, 2010.

[70] L. S. Indrusiak. End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration. *Journal of systems architecture*, 60(7):553–561, 2014.

[71] J.-W. Jeong, S. Yoo, and C. Yoo. Parfait: A new scheduler framework supporting heterogeneous xen-arm schedulers. In *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*, pages 1192–1196. IEEE, 2011.

[72] Z. Jiang and N. Audsley. Vcdc: The virtualized complicated device controller. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[73] A. N. Jiang Zhe and P. Dong. Bluevisor: A scalable real-time hardware hypervisor for heterogeneous many-core embedded systems.

[74] M. A. Khan, S. Saeed, A. Darwish, and A. Abraham. *Embedded and Real Time System Development: A Software Engineering Perspective: Concepts, Methods and Principles*, volume 520. Springer, 2013.

[75] Y. Kinebuchi, H. Koshimae, and T. Nakajima. Constructing machine emulator on portable microkernel. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1197–1198. ACM, 2007.

[76] J. Kiszka. Towards linux as a real-time hypervisor. In *Proceedings of the 11th Real-Time Linux Workshop*, pages 215–224. Citeseer, 2009.

[77] J. A. Landis, T. V. Powderly, R. Subrahmanian, A. Puthiyaparambil, and J. R. Hunter Jr. Computer system para-virtualization using a hypervisor that is implemented in a partition of the host system, July 19 2011. US Patent 7,984,108.

[78] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *ACM Sigplan Notices*, volume 45, pages 97–108. ACM, 2010.

[79] W. F. Lee. *Verilog coding for logic synthesis*. Wiley Online Library, 2003.

[80] Y. Li, M. Danish, and R. West. Quest-v: A virtualized multikernel for high-confidence systems. *arXiv preprint arXiv:1112.5136*, 2011.

[81] J. Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.

[82] M. Masmano, I. Ripoll, A. Crespo, and J. Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272, 2009.

[83] A. Masrur, S. Drossler, T. Pfeuffer, and S. Chakraborty. Vm-based real-time services for automotive control applications. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 218–223. IEEE, 2010.

[84] A. Masrur, T. Pfeuffer, M. Geier, S. Drössler, and S. Chakraborty. Designing vm schedulers for embedded real-time applications. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 29–38. ACM, 2011.

[85] D. May. The xmos architecture and xs1 chips. *IEEE Micro*, 32(6):28–37, 2012.

[86] H. Mei. *Real-Time Stream Processing in Embedded Systems*. PhD thesis, University of York, 2018.

[87] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in xen. In *USENIX Annual Technical Conference*, number LABOS-CONF-2006-003, 2006.

[88] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *INTEGRATION, the VLSI journal*, 38(1):69–93, 2004.

[89] J. Mossinger. Software in automotive systems. *IEEE software*, 27(2):92, 2010.

[90] D. Muench, O. Isfort, K. Mueller, M. Paulitsch, and A. Herkersdorf. Hardware-based i/o virtualization for mixed criticality real-time systems using pcie sr-iov. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 706–713. IEEE, 2013.

[91] D. Pan, B. Alan, J. Zhe, and L. Xiangke. Tzdks: A new trustzone-based dual-criticality system with balanced performance. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2018 IEEE 24th International Conference on*, pages 1–10. IEEE, 2018.

[92] J. L. Peterson and A. Silberschatz. *Operating system concepts*, volume 2. Addison-Wesley Reading, MA, 1985.

[93] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares. Towards a lightweight embedded virtualization architecture exploiting arm trustzone. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–4. IEEE, 2014.

[94] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. Ltzvisor: Trustzone is the key. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

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

[96] D. Plummer. Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware, nov 1982.

[97] J. Polkinghorne and M. Desnoyers. Application specific integrated circuit, Mar. 28 1989. US Patent 4,816,823.

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

[99] I. Present. Cramming more components onto integrated circuits. *Readings in computer architecture*, 56, 2000.

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

[101] A. Rubini and J. Corbet. *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[102] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226. IEEE, 2010.

[103] H. Schild, A. Lackorzynski, A. Warg, et al. Faithful virtualization on a real-time operating system. *RTLWS11*, 2009.

[104] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, et al. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[105] M. J. S. Smith. *Application-specific integrated circuits*, volume 7. Addison-Wesley Reading, MA, 1997.

[106] J. A. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, 1990.

[107] S. Stopp, T. Wolff, F. Irlinger, and T. Lueth. A new method for printer calibration and contour accuracy manufacturing with 3d-print technology. *Rapid Prototyping Journal*, 14(3):167–172, 2008.

[108] K. Thompson. Unix time-sharing system: Unix implementation. *Bell Labs Technical Journal*, 57(6):1931–1946, 1978.

[109] S. Trujillo, A. Crespo, and A. Alonso. Multipartes: Multicore virtualization for mixed-criticality systems. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 260–265. IEEE, 2013.

[110] P. W. Tsai, H. Y. Chou, M. Y. Luo, and C. S. Yang. Design a flexible software development environment on netfpga platform. In *Applied Mechanics and Materials*, volume 411, pages 1665–1669. Trans Tech Publ, 2013.

[111] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[112] J. F. Wakerly. *Digital design*. Pearson Australia Pty Limited, 2016.

[113] C. Waldspurger and M. Rosenblum. I/o virtualization. *Communications of the ACM*, 55(1):66–73, 2012.

[114] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[115] W. H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, 1994.

[116] S. Xi, J. Wilson, C. Lu, and C. Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 39–48. IEEE, 2011.

[117] S. Yoo, Y.-P. Kim, and C. Yoo. Real-time scheduling in a virtualized ce device. In *Consumer Electronics (ICCE), 2010 Digest of Technical Papers International Conference on*, pages 261–262. IEEE, 2010.

[118] P. Yu, M. Xia, Q. Lin, M. Zhu, S. Gao, Z. Qi, K. Chen, and H. Guan. Real-time enhancement for xen hypervisor. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, pages 23–30. IEEE, 2010.

[119] J. Zhang, K. Chen, B. Zuo, R. Ma, Y. Dong, and H. Guan. Performance analysis towards a kvm-based embedded real-time virtualization architecture. In *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*, pages 421–426. IEEE, 2010.

[120] N. A. Zhe Jiang. Gpiocp: Timing-accurate general purpose i/o controller for many-core real-time systems. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2017.

[121] N. A. Zhe Jiang. Blueio: A scalable real-time hardware i/o virtualization system for many-core embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2019.

[122] R. W. Zhuoqun Cheng and Y. Ye. Building Real-Time embedded applications on QduinoMC: A web-connected 3d printer case study. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*. IEEE, 2017.

[123] B. Zuo, K. Chen, A. Liang, H. Guan, J. Zhang, R. Ma, and H. Yang. Performance tuning towards a kvm-based low latency virtualization system. In *Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on*, pages 1–4. IEEE, 2010.