UNIVERSITÁ DI ROMA
"TOR VERGATA"

FACOLTÁ DI INGEGNERIA

DIPARTIMENTO DI INFORMATICA, SISTEMI e PRODUZIONE

P H D T H E S I S

to obtain the title of

PhD of Science

Specialty: Computer Science

Satisfying hard real-time constraints using COTS components

Defended by

Emiliano Betti

Course: XXII

A.A. 2009/2010

Thesis Advisor
Dott. Marco Cesati

Coordinator
Prof. Daniel P. Bovet

Acknowledgments

It is a pleasure to thank those who have made this thesis possible and helped me during my PhD program.

I gratefully thank Marco Cesati, who always guided me since 2003, starting with my Bachelor thesis, and then through my Master's, and now my PhD. His guidance and support drastically impacted the development of my knowledge, notably improving the results I could obtain during my studies.

I also would like to thank Daniel P. Bovet for his precious advice and supervision, and Roberto Gioiosa, whose encouragement and support made a big difference to me. Along with them, I gratefully acknowledge all the other members of the System Programming Research Group at the University of Rome "Tor Vergata", with whom working has been a real pleasure.

Last, but not least, many sincere thanks go to Marco Caccamo and all the members of the "Networked Real Time and Embedded Systems Laboratory" at the University of Illinois at Urbana-Champaign, where I worked over the past two years. Most of this work could not have been done without their knowledge, support, and experience. In particular, I would like to thank Lui Sha, Rodolfo Pellizzoni, Stanley Bak, Bach Dui Bui, and Gang Yao.

Emiliano, May 23, 2010

Satisfying hard real-time constraints using COTS components

Real-time embedded systems are increasingly being built using Commercial Off-The-Shelf (COTS) components such as mass-produced peripherals and buses to reduce costs, time-to-market, and increase performance. Unfortunately, COTS hardware and operating systems are typically designed to optimize average performance, instead of determinism, predictability, and reliability, hence their employment in high criticality real-time systems is still a daunting task.

In this thesis, we addressed some of the most important sources of unpredictability which must be removed in order to integrate COTS hardware and software into hard real-time systems. We first developed ASMP-Linux, a variant of Linux, capable of minimizing both operating system overhead and latency. Next, we designed and implemented a new I/O management system, based on real-time bridges, a novel hardware component that provides temporal isolation on the COTS bus and removes the interference among I/O peripherals. A multi-flow real-time bridge has been also developed to address interperipheral interference, allowing predictable device sharing. Finally, we propose PREM, a new execution model for real-time systems which eliminates interference between peripherals and the CPU, as well as interference between a critical task and driver interrupts. For each of our solutions, we will describe in detail theory aspects, as well as prototype implementations and experimental measurements.

Keywords: real-time, COTS (Commercial Off-The-Shelf), operating system, Linux, computer architecture, scheduling, I/O drivers.

Contents

List of Figures

List of Tables

Introduction

The term *real-time* pertains to computer applications whose correctness is judged not only on whether the results are the correct ones, but also on the time at which the results are delivered. A *real-time system* is a computer system that is able to run real-time applications and meet the constraints of the task at hand. In particular, in this thesis, we focus on *hard real-time systems*, which are high criticality systems, where *missing a deadline* (in other words, delivering the computation results too late with respect of a predefined time) can lead to catastrophic events. For example, an airbag device is a hard real-time system. When a crash happens the airbag must deploy within a tight range of time: an early or late deployment would be unuseful or even dangerous.

Traditionally, because of these strict requirements where the system must not fail in any situation, hard real-time systems have been constructed from hardware and software components specifically designed for real-time. Developing these systems typically costs millions of dollars, in particular when they are employed in aircraft, and the whole development process can be very long. The main reason is that every single component (from the peripheral up to the operating system) has to be designed, implemented, tested, and integrated with the rest of the system. Moreover, nowadays many hard real-time systems make intense use of media streaming, thus they require significant performance in terms of bus and network bandwidth, processors, and memory. In this scenario, COTS (Commercial Off-

The-Shelf) components are every day becoming more attractive for real-time systems.

A COTS hardware or software component is a ready-to-use product, available in the market. Due to mass production, COTS components are significantly cheaper to produce than their application-specific peers. They also evolve very rapidly, improving in performance much faster than a custom real-time system. For example consider Intel or AMD processors, modern video cards, network interfaces, system buses like PCI-E (PCI-Express), and even operating systems like Linux.

Obviously, a custom component produced with a (relatively) small number of pieces cannot easily match the performance of its COTS equivalent. As a consequence COTS components are already used in real-time systems with low criticality (also called *soft real-time systems*[1]), but they are not yet typically employed for hard real-time. The reason is that COTS components are built having as a primary design goal their average performance, and not worst case behaviors and reliability. On the other hand, worst case behaviors and reliability are the most important features for a real-time system, in particular for high criticality systems, where failing or miss a time deadline is unacceptable. For example, modern buses can transfer data at speeds on the order of several Gigabyte per second, but in case more entities are accessing the bus at the same moment, there is no mechanism to prioritize a critical task, which may then be delayed for an excessively long time and even potentially miss its deadline.

[1] More accurate definitions of real-time systems are given later, in chapter 1.

Within this PhD program we started addressing some of the most important sources of unpredictability that must be removed in order to integrate COTS hardware and software in a hard real-time system. In particular, we focused on:

- operating system overhead and latency,
- interference among peripherals,
- interference within peripherals (peripheral sharing),
- interference among peripherals and CPU.

The **operating system overhead and operating system latency** are the first culprits for *software interference*[2] in the execution time of a task. To address this issue we designed and developed a variant of the Linux operating system, called ASMP-Linux [7], which is described in detail in chapter 2.

Analyzing the system behavior when using COTS hardware components, we identified as crucial the unpredictability caused by the I/O subsystem and main memory. In fact, although COTS processors still have many open problems, they have started to appear in hard real-time systems over the last few years. On the other hand, bus, peripherals, and main memory access can still be bottlenecks for the system, often introducing unpredictable delays.

Interference among peripherals is observable when more than one peripheral asynchronously accesses the bus to read or write data

[2] Actually, we will see in chapter 1 that also some hardware interference, very difficult to avoid, can slightly affect the measure of these delays.

from main memory. This problem will be treated in detail in chapter 3, together with our proposed solution: a novel real-time I/O management system, based on *real-time bridges* [4]. Such a system, transparent to end-user CPU applications, introduces two new types of components into the COTS system, *real-time bridges* and a *reservation controller*, to provide temporal isolation on the COTS bus and remove the possibility of interference.

A real-time bridge evolution has been also developed to handle **interference within peripherals**. *Multi-flow real-time bridges* [5], described in chapter 4, deals with peripherals shared among tasks with different criticalities, enforcing a real-time I/O scheduling within shared peripherals and on the PCI-E bus.

Finally, we are currently working on a solution for **interference between peripheral and the CPU**, observed when both access main memory at the same time [45, 44]. Our solution, described in chapter 5, proposes a new execution model, called PREM (PRedictable Execution Model) [48], that combines new compiler techniques and the real-time bridges and is able to enforce predictable memory accesses while avoiding bus interference between tasks running on the CPU and peripherals.

Before we start discussing the challenges just introduced and our proposed solutions, chapter 1 gives an overview of some general concepts on real-time systems, COTS components, and definitions of hardware and software sources of unpredictability.

Chapter 1

Real-time systems and COTS components

This chapter introduces some key concepts regarding *real-time systems* and COTS (Commercial Off-The-Shelf) components (hardware and operating systems). The main goal is giving the reader an overview of the basic knowledge on these topics. Moreover the main requirements of the real-time systems will be described in the context of the limitations of the COTS components. This will give a general idea of the challenges that we address later in this work.

1.1 Real-time systems

In the literature there are many different definitions of real-time systems. The following is the canonical definition from Donald Gillies [2, 26]:

Definition 1. *"A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."*

Along with definition 1, POSIX Standard 1003.1 defines "real-time" for operating systems as:

Definition 2. *"Real-time in operating systems is the ability of the operating system to provide a required level of service in a bounded response time."*

In both definitions is evident how *time* is one of the most important aspects of a real-time system. The main reason for this is that a real-time task is usually running on a, so called, *Cyber-Physical System*, which, in other words, is a system where system's computational interacts with physical elements [67]. It follows that such a task must react to events that take place in the real world, and that the *response time* is the key element of correctness for a real-time system. If the answer to the event is delivered too late, it may be useless, or worse, catastrophic events could happen in the real world.

Finally it is also important to emphasize that "react within a deadline" needs not mean "react fast". The *deadline* within the real-time system must complete its computation and deliver its results could be in the order of milliseconds, seconds, or minutes as well. The only important thing is to complete the task before the deadline is missed.

1.1.1 Hard and soft real-time

Real-time systems and applications can be classified in several ways. One classification scheme divides them into two main classes: "hard" real-time and "soft" real-time.

A *hard real-time* system is characterized by the fact that meeting the application's deadlines is the primary metric of success. In other words, failing to meet the application's deadlines (timing requirements, quality of service, latency constraints, and so on) is a catastrophic event that must be absolutely avoided.

Conversely, a *soft real-time* system is suitable to run applications whose deadlines must be satisfied "most of the time", that is, the work carried out by a soft real-time application retains some value even if a deadline is missed. In soft real-time systems some design goals, such as achieving high average throughput, may be as important, or more important, than meeting application deadlines.

An example of a hard real-time application is a missile defense system: whenever a radar system detects an attacking missile, the real-time system has to compute all the information required for tracking, intercepting, and destroying the target missile. If it fails, catastrophic events may follow.

A very common example of soft real-time application is a video stream decoder: the incoming data have to be decoded on the fly. If, for some reason, the decoder is not able to translate the stream before its deadline and a frame is lost, nothing catastrophic happens: the user will likely not even take notice of the missing frame (the human eye cannot distinguish images faster than one tenth of a second).

Needless to say, hard real-time applications put many more time and resource constraints on the system than do soft real-time applications; thus hard real-time systems are difficult to design and implement.

In this thesis, we focus on hard real-time systems.

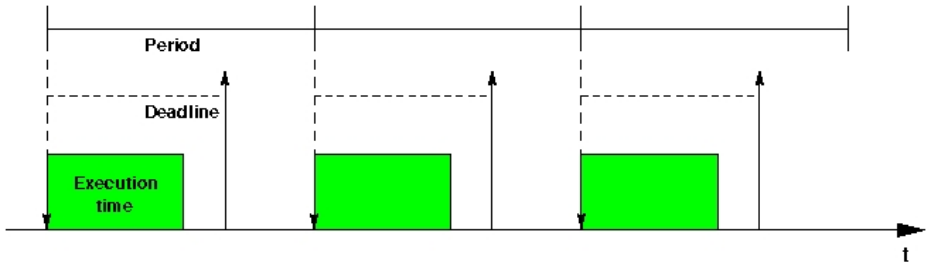

Figure 1.1: A real-time periodic task

1.1.2 Periodic and event-driver real-time applications

There exists also a classification scheme that divides real-time applications into "periodic" and "event-driven" classes.

As the name suggests, *periodic* applications execute a task periodically, and have to complete their job by a predefined deadline within each period. In figure 1.1, we can note that the deadline need not coincide with the end of the period.

A nuclear power plant monitor is an example of a periodic hard real-time application, while a multimedia decoder is an example of a periodic soft real-time application.

Conversely, *event-driven* applications give rise to processes that spend most of the time waiting for some event. When an expected even occurs, the real-time process waiting for that event must wake up and handle it in such a way as to satisfy the predefined time constraints.

The aforementioned missile defense system is an example of an event-driven hard real-time application, while a network router might

be an example of an event-driven soft real-time application.

Dealing with real-time applications, the operating system must guarantee a sufficient amount of each resource (processor time, memory, network bandwidth, and so on) to each application so that it succeeds in meeting its deadlines. Essentially, in order to effectively implement a real-time system for periodic or event-driven applications a resource allocation problem must be solved. Clearly, the operating system should assign resources to processes according to their priorities, so that, for instance a high-priority task will never be delayed by a low-priority task. Real-time scheduling theory is a widely explored field; for an up-to-date survey of real-time theory see [60].

In this thesis we focus on both event-driven and periodic hard real-time applications. In particular we will study the case where hard real-time applications run on COTS hardware and in a general-purpose COTS operating system, like Linux.

In the following sections we will introduce the concept of *jitter* (section 1.2) and how it affects the system's predictability. In section 1.3, we will describe the characteristics of *COTS* components and why they are used in real-time systems. After that, in section 1.4, *operating system overhead and latency* will be also discussed as main software sources of jitter. Finally, section 1.5 discusses COTS hardware impact on system's predictability.

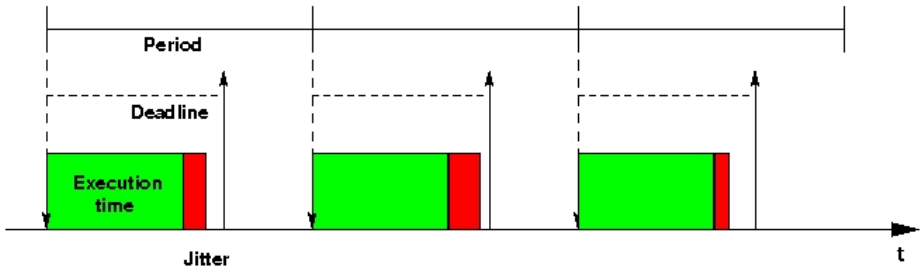

Figure 1.2: An example of jitter in a real-time periodic task

1.2 Jitter and predictability

In an ideal world a real-time process would run undisturbed from the beginning to the end, directly communicating with the device it is intended to control. In this situation a real-time process will require always the same amount of time T to complete its task. Such a system is said to be *deterministic*.

In the real world, however, there are several software layers between the device and the real-time process, and other processes and services could be running at the same time competing for hardware resources. Moreover, other devices might require attention and interrupt the real-time execution. As a result, the amount of time required by the real-time process to complete its task is actually $T_x = T + \varepsilon$, where $\varepsilon \geq 0$ is a delay caused by the system (hardware and software). The variation in the values of ε is defined as the system's *jitter*, a measure of the non-determinism of a system. In figure 1.1 an ideal case for a periodic task is showed, instead in figure 1.2 we can see a more realistic case where a variable jitter increases the execution time in a non-deterministic way.

Determinism is a key factor for hard real-time systems: the larger the jitter, the less deterministic the system's response. Thus, jitter is also an indicator of the hard real-time capabilities of a system: if the jitter is greater than some critical threshold, the system can be unsuitable for real-time. As a consequence, a real-time operating system and the associated hardware components must be designed so as to minimize the jitter observed by the real-time applications.

Jitter, by its nature, is not constant and makes the system behavior unpredictable; for this reason, real-time application developers must provide an estimated *Worst Case Execution Time* (WCET), which is an upper bound (often quite pessimistic) of the real-time application's execution time. A real-time application meets its deadline if $T_x \leq$ WCET. If it is not possible to estimate an upper bound for the execution time, then the system is unpredictable and cannot be used for real-time applications. In other words, hardware components and operating systems for real-time must be designed to minimize the jitter, as we already said, but also to follow predictable behaviors, in order to have a computable WCET.

1.3 Commercial Off-The-Shelf components

With the term Commercial Off-The-Shelf (COTS) we refer to software or hardware products ready-made and available for sale, lease, or license to the general public [66].

Even tough traditionally, hardware and software for real-time systems have been specifically designed, lately real-time embedded systems are increasingly being built using Commercial Off-The-Shelf

(COTS) hardware. Also open-source COTS operating system are used, usually after some modifications, and nowadays many commercial real-time operating systems are Linux-based.

Motivations for using COTS components include reduction of overall system development time and costs (as components can be bought or licensed instead of being developed from scratch) and reduced maintenance costs.

Since several years ago, most of the real-time embedded systems that require high computational power make use of COTS processors, mainly for their high performance/cost ratio. Moreover integrating high-speed COTS peripherals within a real-time system offers substantial benefits in terms of cost reduction, time-to-market, and overall performance. Since COTS components are already designed, a system's time-to-market can be reduced by reusing existing components instead of creating new ones. Additionally, overall performance of mass produced components is often significantly higher than custom made systems. For example, a PCI-Express bus [42] can transfer data three orders of magnitude faster than the real-time SAFEbus [31].

1.3.1 COTS components' challenges

COTS components are typically designed paying little or no attention to worst-case timing behaviors. Usually COTS designers aim to improve average performance, instead of determinism, predictability, and reliability.

Modern COTS processors are intrinsically parallel and so complex

that it is not possible to predict when an instruction will complete. For example cache misses, branch predictors, pipeline, out-of-order execution, and speculative accesses could significantly alter the execution time of an in-flying instruction.

COTS buses, like PCI-E, can be very fast, but also completely unpredictable. Bus arbitration policies and transactions' scheduling are "black-boxes", and their implementation details are known only by the original manufacturer.

Integrating COTS peripherals within a real-time system is also a daunting task. The main reason is the unpredictable timing of the I/O subsystem: some shared resources, such as the PCI-E bus or the memory controller, require an arbitration among the hardware devices, that is, a lock mechanism. Moreover, low priority data transferred on the bus could steal bandwidth to real-time applications with higher priority. There are also "intrinsic indeterministic buses" used to connect devices to the system or system to system, such as Ethernet or PCI buses [57].

It is worth to say that COTS components have also other kinds of drawbacks, other than the ones related to system's determinism. For example, as personal users we like to often update our system, following every new technology in the market. On the other hand, big companies that sell or use real-time systems can fear to be dependent on a third-party component vendor, because future changes to the product will not be under their control. Moreover components integration can require much more work than it would if every single component was designed for the same specific system.

However, at the end of the day, integration problems and compo-

nents availability on the market can be paid by the fact the producing your own system (hardware and operating system), especially in case of high performance requirements, can be even more expensive or sometimes impossible. Looking at the predictability of the system instead, using COTS components could seem implicitly giving up on strict determinism, but, as a matter of fact, commercial and industrial real-time systems often follow the *five-nines rule*:

Definition 3. *The system is considered hard real-time if a real-time application catches its deadline 99.999% of the times.*

Consequently, real-time industry is already using COTS components. And it is even more important that most of the companies are investing considerable amount of money in research, foreseeing a complete transaction from their dedicated systems to complete COTS based systems.

1.4 Operating system overhead and latency

This section gives an overview of the software jitter measured by an operating system. This will help to better understand the characteristics required by a real-time operating system and, consequently, the challenges that have to be faced using a COTS operating system, like Linux. Before starting tough, it must be clear that predictability is not the only feature required by a real-time operating system: it is definitely the most important one, but also other features are required, like fault tolerance, or system partitioning. Anyway, in this

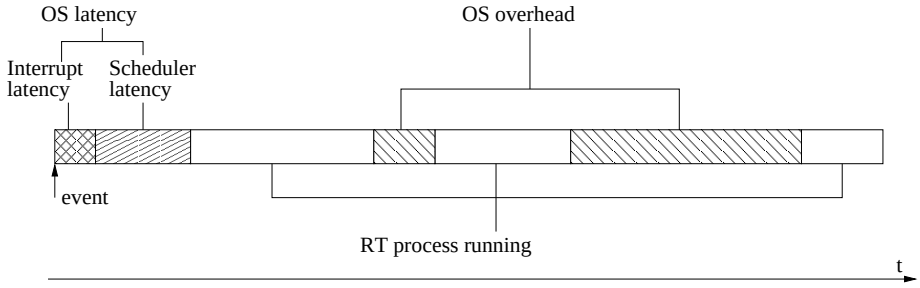

Figure 1.3: Jitter

work we mostly focus on assuring predictability, hence other real-time system's features will not be covered in detail.

As discussed in [29, 51, 72], short, unpredictable activities such as interrupts handling are the main causes of large jitter in computer systems. As shown in Figure 1.3, the jitter seen by an operating system is composed by two main components: the "operating system overhead" and the "operating system latency".

The *operating system overhead* is the amount of time the CPU spends while executing system's code—for example, handling the hardware devices or managing memory—and code of other processes instead of the real-time process' code.

The *operating system latency* is the time elapsed between the instant in which an event is raised by some hardware device and the instant in which a real-time application starts handling that event. Also periodic real-time applications suffer from operating system latency: for example, the operating system latency may cause a periodic application to wake up with some delay with respect to the beginning of its real-time period.

The definitions of overhead and latency are rather informal, be-

cause they overlap on some corner cases. For instance, the operating system overhead includes the time spent by the kernel in order to select the "best" process to run on each CPU; the component in charge of this activity is called *scheduler*. However, in a specific case the time spent while executing the scheduler is not accounted as operating system overhead but rather as operating system latency: it happens when the scheduler is going to select precisely the process that carries on the execution of the real-time application. On the other hand, if some non-real-time interrupts occur between a real-time event and the wake-up of the real-time applications, the time spent by the kernel while handling the non-real-time interrupts should be accounted as overhead rather than latency.

As illustrated in Figure 1.3, operating system latency can be decomposed in two components: the "interrupt latency" and the "scheduler latency".

The *interrupt latency* is the time required to execute the interrupt handler connected to the device that raised the interrupt, i.e., the device that detected an event the real-time process has to handle.

The *scheduler latency* is the time required by the operating system scheduler to select the real-time task as the next process to run and assign it the CPU.

Interrupt latency, scheduler latency, and operating system overhead reduce the system's determinism and, thus, its real-time capabilities. It follows that, the main goals of a real-time operating system must be to reduce software latency and overhead, and, over all, to make them predictable. The best optimized operating system could be able to manage operating system overhead with opportune

scheduling policies. Anyway, it will still suffer from some operating system latency, because the sources of those delays (interrupt and scheduler latencies) are code that can be executed only between the time when an event happens and its handling deadline expires.

1.5 Hardware jitter

As we just said, most of the system's jitter is due to the operating system, and in particular to:

- interrupts handling, and more in general devices related activities,
- hardware resources' sharing (i.e., overhead and latency introduce by a CPU scheduler, or locks needed to synchronize different users on the same resource),
- other periodic or sporadic activities important for the system (i.e., disk cache flushing).

Anyway, while measuring operating system overhead and operating system latency we have to consider that part of those delays could be also accounted to the underline hardware. This is specially true when our system is running on COTS hardware.

Several hardware resources are shared among different hardware components: caches, buses, and main memory, for example, can all be sources of unexpected delays when two different hardware entities are competing for the same resource. The execution time of a real-time process, e.g., could be delayed by a peripheral when both try to

access main memory at the same time: the process in consequence of some cache misses, and the peripheral performing DMA (Direct Memory Access) transactions. In this kind of situations the memory bus and the memory itself would be a bottleneck: one or both of the competing operations could indeed suffer for a delay, potentially missing their deadline.

The indeterminism caused by the hardware cannot be reduced by the software, thus no real-time operating system can have better performances than those of the underlying hardware. In other words, the execution time T_x of a real-time task will always be affected by some jitter, no matter of how good the real-time operating system is.

1.6 Proposed solutions

In this work, in order to obtain hard real-time performance with strict deadline using COTS componets, we propose:

- a variant of the Linux operating system, discussed in chapter 2, that significantly reduces operating system overhead and operating system latency;
- a simple and flexible architectural modification, described in chapters 3 and 4, that completely removes interferences among and within peripherals;
- a new execution model, treated in chapter 5, that faces interferences between CPU and peripherals, schedules drivers activities, and is meant to support and integrate all the previous solutions.

However, we will see how our new architecture does not mean to directly modify COTS hardware components (which will not make sense for COTS' nature), but just to integrate them, in order to make their behavior deterministic. Finally we developed prototypes for each proposed component and, with our experimental results, we can show how our solutions reduce, and in many cases remove, the most important sources of unpredictabilities in COTS systems.

Chapter 2

ASMP-Linux

Multiprocessor systems, especially those based on multi-core processors, and new operating system architectures could satisfy the ever increasing computational requirements of embedded systems. Unfortunately, every time the level of parallelism is increased, more resources have to be shared among different tasks. As we said in section 1.2, predictability is the first requirement of a real-time system, and instead shared resources are the main origin of indeterminism. For example, in [49] the authors experimented an increment of $\sim 200\%$ in the WCET when 2 cores and peripherals access memory at the same time. In uni-processor systems the measured increment was $\sim 44\%$ [44]. Hence, this increased unpredictability is among the most important reasons why multiprocessor systems are not very common yet in hard real-time environments.

In this chapter, we present ASMP-Linux, which stands for *ASymmetric MultiProcessor Linux*, and is a modified Linux kernel that can be used in COTS multiprocessor embedded systems with hard real-time requirements. It has been developed in 2007 within this PhD program, and in 2008 was presented in the EURASIP Journal on Embedded Systems [7].

ASMP-Linux provides a high responsiveness, open-source hard

real-time operating system for multiprocessor architectures, using an asymmetric kernel approach. We will see how ASMP-Linux is capable of providing hard real-time capabilities with COTS multiprocessor while maintaining the code simple and not impacting on the performance of the rest of the system. Moreover, ASMP-Linux does not require code changing or application re-compiling/re-linking. In a conventional (symmetric) kernel, I/O devices and CPUs are considered alike, since no assumption is made on the system's load. Asymmetric kernels, instead, consider real-time processes and related devices as privileged and shield them from other system activities.

The main advantages offered by ASMP-Linux to real-time applications are:

- Deterministic execution time (up to a few hundreds of nanoseconds).
- Very low system overhead.
- High performance and high responsiveness.

Clearly, all of the good things offered by ASMP-Linux have a cost, namely at least one processor core dedicated to real-time tasks. The current trend in processor design is leading to chips with more and more cores. Because the power consumption of single-chip multicore processors is not exceedingly higher than that of single-core processors, we can also expect that in a near future many embedded systems will make use of multi-core processors. Nowadays, in fact, most of the companies in the real-time industry are investigating multi-core solutions: soft real-time embedded systems are already

available, and satisfing hard real-time constraints in multi-core systems is a very hot topic for modern research. Thus, even though several issues have not to be solved yet, in a near future multi-core systems could become very common for hard real-time systems too, and the hardware requirements of real-time operating systems such as ASMP-Linux will become quite acceptable as well.

In the rest of this chapter we will first review some background information, so that the reader can better understand ASMP-Linux and the main idea behind it: Section 2.1 describes the evolution of single-chip multi-core processors; Section 2.2 gives an introduction to system partitioning; Section 2.3 illustrates briefly the main characteristics of other asymmetric kernels developed before ASMP-Linux. After that ASMP-Linux will be described in detail. Section 2.4 discusses ASMP-Linux implementation, and section 2.5 lists the tests performed on different computers and the results obtained.

2.1 Multiprocessor systems

As we discussed in the section 1.3, the increasing demand for computational power leaded real-time embedded systems developers to use general-purpose COTS processors, such as ARM, Intel, AMD, or IBM's POWER, instead of micro-controllers or Digital Signal Processors (DSPs).

Furthermore, many hardware vendors started to develop and market *system-on-chip* (SoC) devices, which usually include on the same integrated circuit one or more COTS CPUs, together with other specialized processors like DSPs, peripherals, communication buses, and

memory. System-on-chip devices are particularly suited for embedded systems because they are cheaper, more reliable, and consume less power than their equivalent multi-chip systems. Actually, power consumption can be considered as a very important constraint in some embedded systems [22].

In the quest for the highest CPU performances, hardware developers are faced with a difficult dilemma. On one hand, the Moore's Law does not apply to computational power any more, that is, computational power is no longer doubling every 18 months as in the past. On the other hand, power consumption continues to increase more than linearly with the number of transistors included in a chip, and the Moore's Law still holds for the number of transistors in a chip.

Several technology solutions have been adopted to solve this dilemma. Some of them try to reduce the power consumption by sacrificing computational power, usually by means of *frequency scaling*, *voltage throttling*, or both. For instance, the Intel Centrino processor [16] (first generation released in March 2003) has a variable CPU clock rate ranging between 600 MHz and 1.5 GHz, which can be dynamically adjusted according to the computational needs.

Other solutions try to get more computational power from the CPU without increasing power consumption. For instance, a key idea was to increase the *Instruction Level Parallelism* (ILP) inside a processor; this solution worked well for some years, but nowadays the penalty of a cache miss (which may stall the pipeline) or of a miss-predicted branch (which may invalidate the pipeline) has become way too expensive.

Chip-Multi-Thread (CMT) [38] processors aim to solve the problem from another point of view: they run different processes at the same time, assigning them resources dynamically according to the available resources and requirements. Historically the first CMT processor was a *coarse-grained multithreading* CPU (IBM RS64-II [8, 64]) introduced in 1998: in this kind of processor only one thread executes at any instance. Whenever that thread experiments a long-latency delay (such as a cache miss), the processor swaps out the waiting thread and starts to execute the second thread. In this way the machine is not idle during the memory transfers and, thus, its utilization increase.

Fine-grained multithreading processors improve the previous approach: in this case the processor executes the two threads in successive cycles, most of the time in a round-robin fashion. In this way the two threads are executed at the same time but, if one of them encounters a long-latency event, its cycles are lost. Moreover, this approach requires more hardware resources duplication than the coarse-grained multithreading solution.

In *Symmetric MultiThreading* (SMT) processors two threads are executed at the same time, like in the fine-grained multithreading CPUs; however, the processor is capable of adjusting the rate at which it fetches instructions from one thread flow or the other one dynamically, according to the actual environmental situation. In this way, if a thread experiments a long-latency event, its cycles will be used by the other thread, hopefully without loosing anything.

Yet another approach consists of putting more processors on a chip rather than packing into a chip a single CPU with a higher

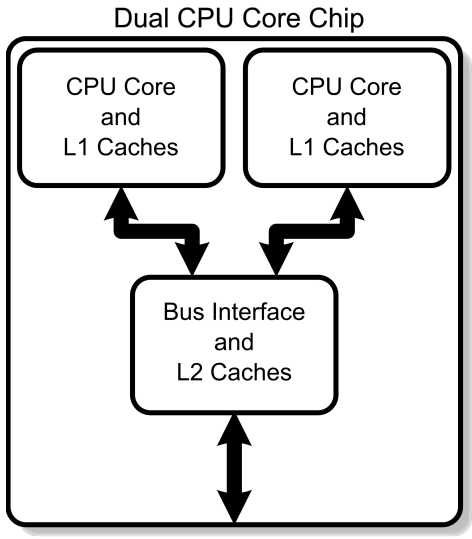

Figure 2.1: An example of dual core processor with shared L2 cache and bus interface [70].

frequency. This technique is called *chip-level multiprocessing* (CMP), but it is also known as "chip multiprocessor"; essentially it implements symmetric multiprocessing (SMP) inside a single VLSI integrated circuit. Multiple processor cores typically share a common second- or third-level cache and interconnections. Figure 2.1 shows an example of a dual core processor with shared L2 cache and bus interface.

In 2001 IBM introduced the first chip containing two single-threaded processors (cores): the POWER4 [65]. Since that time, several other vendors have also introduced their multi-core solutions: dual-core and quad-core processors are nowadays widely used (e.g., Intel Dual Core 2 [69] introduced in 2006, or AMD Opteron [19] introduced in 2005) and single core processors are almost out of the market; hexa-core processors (sometimes called six-core) are already on the

shelves (like Intel Xeon Dunnington [14]), and eight-core processors (also called *octo-core*) are starting to appear (like Intel Xeon Beckton [71]).

In conclusion, McKenney's forecast [39] that in a near future many embedded systems will sport several CMP and/or CMT processors is becoming reality. In fact, the small increase in power consumption is justified by the large increment of computational power available to the embedded system's applications. Furthermore, the actual trend in the design of system-on-chip devices is showing that such chips starts including multi-core processors. Therefore, nowadays an embedded system designer can create boards having many processors almost "for free", that is, without the overhead of a much more complicated electronic layout or a much higher power consumption.

On the other hand real-time theory for multiprocessor systems (about schedulability, static analysis, etc) is not very solid yet and most of the big companies dealing with real-time systems are still afraid of migrating to multiprocessors systems. In this work we show how a *partitioned* multiprocessor system can be used in a real-time environment to reduce and in some case eliminate the unpredictability introduced by the operating system (see section 1.4).

2.2 System partitioning

The real-time operating system must guarantee that, when a real-time application requires some resource, that resource is made available to the application as soon as possible. The operating system should also ensure that the resources shared among all processes—

the CPU itself, the memory bus, and so on—are assigned to the processes according to a policy that take in consideration the priorities among the running processes.

As long as the operating system has to deal only with processes, it is relatively easy to preempt a running process and assign the CPU to another, higher-priority process. Unfortunately, as discussed in 1.4, external events and operating system critical activities, required for the correct operation of the whole system, occur at unpredictable times and are usually associated with the highest priority in the system. Thus, for example, an external interrupt could delay a critical, hard real-time application that, deprived of the processor, could eventually miss its deadline. Even if the application manages to catch its deadline, the operating system may introduce a factor of non-determinism that is tough to predict in advance.

Therefore, handling both external events and operating system critical activities while guaranteeing strict deadlines is the main problem in real-time operating systems. Multiprocessor systems make this problem even worse, because operating system activities are much more complicated.

In order to cope with this problem, real-time operating systems are usually partitioned horizontally or vertically. As illustrated in Figure 2.2, *horizontally partitioned operating systems* have a bottom layer (called *hardware abstraction layer*, or HAL) that virtualizes the real hardware; on top of this layer there are several *virtual machines*, or *partitions*, running a standard or modified operating system, one for each application's domain; finally, applications run into their own domain as they were running on a dedicated machine.

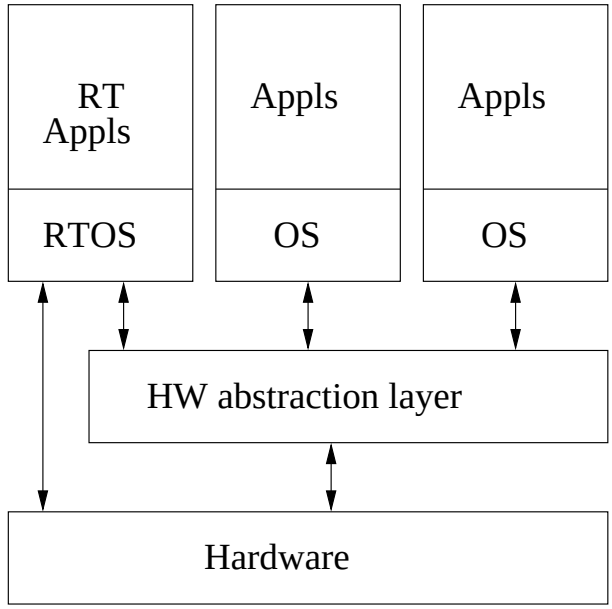

Figure 2.2: Horizontally partitioned operating system

In horizontally partitioned operating systems the real-time applications have an abstract view of the system; external events are caught by the hardware abstraction layer and propagated to the domains according to their priorities. While it seems counter-intuitive to use virtual machines for hard real-time applications, this approach works well in most of the cases, even if the hardware abstraction layer—in particular the partitions scheduler or the interrupt dispatcher—might introduce some overhead. Several Linux-based real-time operating systems such as RTAI [20] (implemented on top of *Adeos* [79]) and some commercial operating systems like Wind River's VxWorks [54] use this software architecture.

In contrast with the previous approach, in a *vertically partitioned*

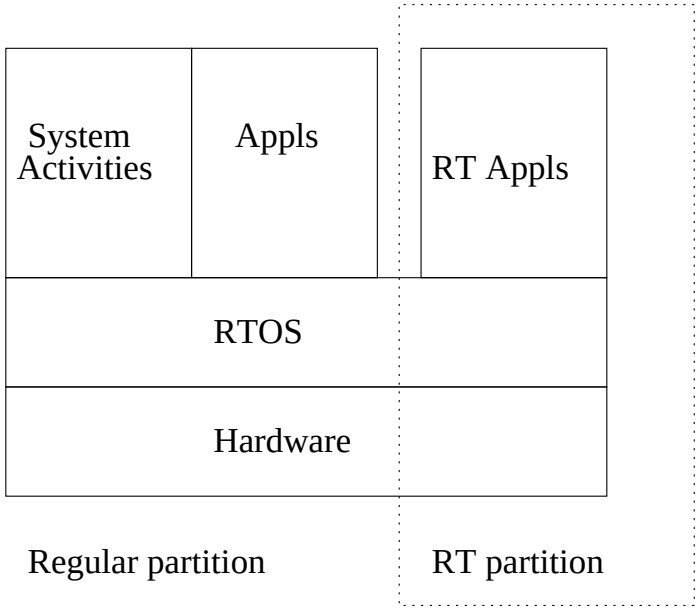

Figure 2.3: Vertically partitioned operating system

operating system the resources that are crucial for the execution of the real-time applications are directly assigned to the application themselves, with no software layer in the middle. The non-critical applications and the operating system activities not related to the real-time tasks are not allowed to use the reserved resources. This schema is illustrated in Figure 2.3.

Thanks to this approach, followed by ASMP-Linux, the real-time specific components of the operating system are kept simple, because they do not require complex partition schedulers or virtual interrupt dispatchers. Moreover, the performances of a real-time application are potentially higher with respect to those of the corresponding application in a horizontally partitioned operating system,

because there is no overhead due to the hardware abstraction layer. Finally, in a vertically partitioned operating system, the non-real-time components never slow down unreasonably, because these components always have their own hardware resources different from the resources used by the real-time tasks.

2.3 Asymmetric multiprocessor kernels

The idea of dedicating some processors of a multiprocessor system to real-time tasks is not new. In an early description of the ARTiS system included in [40], processors are classified as real-time and non-real-time. Real-time CPUs execute non-preemptible code only, thus tasks running on these processors perform predictably. If a task wants to enter into a preemptible section of code on a real-time processor, ARTiS will automatically migrate this task to a non-real-time processor.

Furthermore, dedicated load-balancing strategies allow all CPUs to be fully exploited. In a more recent article by the same group [37], processes have been divided into three groups: highest priority (RT0), other real-time Linux tasks (RT1+), and non-real-time tasks; furthermore, a basic library has been implemented to provide functions that allow programmers to register and unregister RT0 tasks. Since ARTiS relies on the standard Linux interrupt handler, the system latency may vary considerably: a maximum observed latency of 120 μsecs on a 4-way Intel Architecture-64 (IA-64) heavily loaded system has been reported in [37].

A more drastic approach to reduce the fluctuations in the latency

time has been proposed independently in [28] and [10]. In this approach, the source of real-time events is typically a hardware device that drives an IRQ signal not shared with other devices. The asymmetric multiprocessor (ASMP) system is implemented by binding the real-time IRQ and the real-time tasks to one or more "shielded" processors, which never handle non-real-time IRQs or tasks. Of course, the non-real-time IRQs and non-real-time tasks are handled by the other processors in the system. As discussed in [28] and [10], the fluctuations of the system latency are thus significantly reduced.

It is worth noting that, since version 2.6.9 released in October 2004, the standard Linux kernel includes a boot parameter (`isolcpus`) that allows the system administrator to specify a list of "isolated" processors: they will never be considered by the scheduling algorithm, thus they do not normally run any task besides the per-CPU kernel threads. In order to force a process to migrate on a isolated CPU, a programmer may make use of the Linux-specific system call `sched_setaffinity()`. The Linux kernel also includes a mechanism to bind a specific IRQ to one or more CPUs; therefore, it is easy to implement an ASMP mechanism using a standard Linux kernel.

However, the implementation of ASMP discussed in this thesis, ASMP-Linux, is not based on the `isolcpus` boot parameter.A clear advantage of ASMP-Linux is that the system administrator can switch between SMP and ASMP mode at run time, without rebooting the computer. Moreover, as explained in Section 2.4.2, ASMP-Linux takes care of avoiding load rebalancing for asymmetric processors, thus it should be slightly more efficient than a system based only on `isolcpus`.

2.3.1 Other asymmetric systems

Although in this work we will concentrate on the real-time applications of asymmetric kernels, it is worth mentioning that these kernels are also used in other areas. As an example, some multi-core chips include different types of cores and require thus an asymmetric kernel to handle each core properly. The IBM Cell Broadband Engine (BE) discussed in [12], for instance, integrates a 64-bit PowerPC processor core along with eight "synergistic processor cores". This asymmetric multi-core chip is the hearth of the Sony PS3 PlayStation console, although other applications outside of the video game console market, such as medical imaging and rendering graphical data, are been considered.

2.4 ASMP-Linux implementation

ASMP-Linux has been originally developed as a patch for the 2.4 Linux kernel series in 2002 [28]. After several revisions and major updates, ASMP-Linux was implemented as a patch for the Linux kernel 2.6.19.1, the latest Linux kernel version available when this implementation has been done.

One of the design goals of ASMP-Linux is simplicity: because Linux developers introduce quite often significant changes in the kernel, it would be very difficult to maintain the ASMP-Linux patch if it was intrusive or overly complex. Actually, most of the code specific to ASMP-Linux is implemented as an independent kernel module, even if some minor changes in the core kernel code—mainly in the

scheduler, as discussed in Section 2.4.2—are still required.

Another design goal of ASMP-Linux is architecture-independency: the patch can be easily ported to many different architectures, besides the IA-32 architecture that has been adopted for the experiments reported in Section 2.5.

It should be noted, however, that in a few cases ASMP-Linux needs to interact with the hardware devices (for instance when dealing with the local timer, as explained in Section 2.4.3). In these cases, ASMP-Linux makes use of the interfaces provided by the standard Linux kernel; those interfaces are, of course, architecture-dependent but they are officially maintained by the kernel developers.

It is also worth noting that what ASMP-Linux can or cannot do depends ultimately on the characteristics of the underlying system architecture. For example, in the IBM's POWER5 architecture disabling the in-chip circuit that generates the local timer interrupt (the so-called *decrementer*) also disables all other external interrupts. Thus, the designer of a real-time embedded system must be aware that in some general-purpose COTS architectures it might be simply impossible to mask all sources of system jitter.

ASMP-Linux is released under the version 2 of the GNU General Public License [25], and it is available at [6] to all the developers who wish to work with it. Actually it is very interesting how, after almost 3 years, we are still receiving feedback from people working on ASMP-Linux.

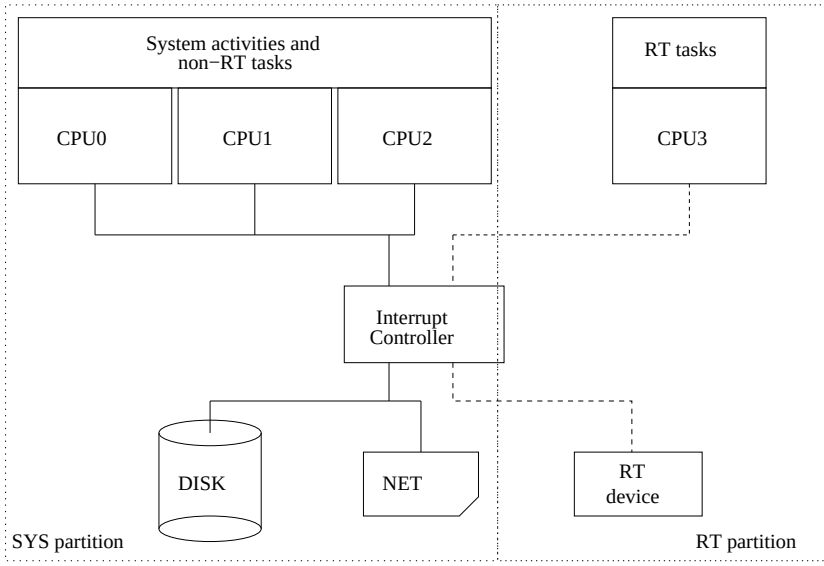

Figure 2.4: ASMP-Linux partitioning

2.4.1 System partitions

ASMP-Linux is a vertically partitioned operating system. Thus, as explained in Section 2.2, it implements two different kinds of partitions:

System partition It executes all the non-real-time activities, such as daemons, normal processes, interrupt handling for non critical devices, and so on.

Real-time partition It handles some real-time tasks, as well as any hardware device and driver that is crucial for the real-time performances of that tasks.

In an ASMP-Linux system there is exactly one system partition, which may consist of several processors, devices, and processes;

moreover, there should always exist at least one real-time partition (see Figure 2.4). Additional real-time partitions might also exist, each handling one specific real-time application.

Each real-time partition consists of a processor (called *shielded CPU*, or shortly S-CPU), $n_{irq} \geq 0$ IRQ lines assigned to that processor and corresponding to the critical hardware devices handled in the partition, and $n_{task} \geq 0$ real-time processes (there could be no real-time process in the partition; this happens when the whole real-time algorithm is coded inside an interrupt handler).

Each real-time partition is protected from any external event or activity that does not belong to the real-time task running on that partition. Thus, for example, no conventional process can be scheduled on a shielded CPU and no normal interrupt can be delivered to that processor.

2.4.2 Process handling

The bottom rule of ASMP-Linux while managing processes is the following:

> Every process assigned to a real-time partition must run only in that partition; furthermore, every process that does not belong to a real-time partition cannot run on that partition.

It should be noted, however, that a real-time partition always include a few peculiar non-real-time processes. In fact, the Linux kernel design makes use of some processes, called *kernel threads*, which

execute only in Kernel Mode and perform system-related activities. Besides the *idle process*, a few kernel threads such as *ksoftirqd* [9] are duplicated across all CPUs, so that each CPU executes one specific instance of the kernel thread. In the current design of ASMP-Linux, the per-CPU kernel threads still remain associated with the shielded CPUs, thus they can potentially compete with the real-time tasks inside the partition. As we shall see in Section 2.5, this design choice has no significant impact on the operating system overhead and latency.

The ASMP-Linux patch is not intrusive because the standard Linux kernel already provides support to select which processes can execute on each CPU. In particular, every process descriptor contains a field named `cpus_allowed`, which is a bitmap denoting the CPUs that are allowed to execute the process itself. Thus, in order to implement the asymmetric behavior, the bitmaps of the real-time processes are modified so that only the bit associated with the corresponding shielded CPU is set; conversely, the bitmaps of the non-real-time processes are modified so that the bits of all shielded CPUs are cleared.

A real-time partition might include more than one real-time process. Scheduling among the real-time partition is still achieved by the standard Linux scheduler, so the standard Linux static and dynamic priorities are honored. In this case, of course, it's up to the developer of the real-time application to ensure that the deadlines of each process are always caught.

The list of real-time processes assigned to a real-time partition may also be empty: this is intended for those applications that do

not need to do anything more than handling the interrupts coming from some hardware device. In this case, the device handled in a real-time partition can be seen as a *smart device*, that is, a device with the computational power of a standard processor.

The ASMP-Linux patch modifies in a few places the scheduling algorithm of the standard Linux kernel. In particular, since version 2.6, Linux support the so-called *scheduling domains* [9]: the processors are evenly partitioned in domains, which are kept balanced by the kernel according to the physical characteristics of the CPUs. Usually, the load in CMP and CMT processors will be equally spread on all the physical chips. For instance, in a system having two physical processors *chip0* and *chip1*, each of which being a 2-way CMT CPU, the standard Linux scheduler will try to put two running processes in such a way to assign one process to the first virtual processor of *chip0* and the other one to the first virtual processor of *chip1*. Having both processes running on the same chip, one on each virtual processor, would be a waste of resource: an entire physical chip kept idle.

However, load balancing among scheduling domains is a time-consuming, unpredictable activity. Moreover, it is obviously useless for shielded processors, because only predefined processes can run on each shielded CPU. Therefore, the ASMP-Linux patch changes the load balancing algorithm so that shielded CPUs are always ignored.

2.4.3 Interrupts handling

As mentioned in Section 1.2, interrupts are the major cause of jitter in real-time systems, because they are generated by hardware devices

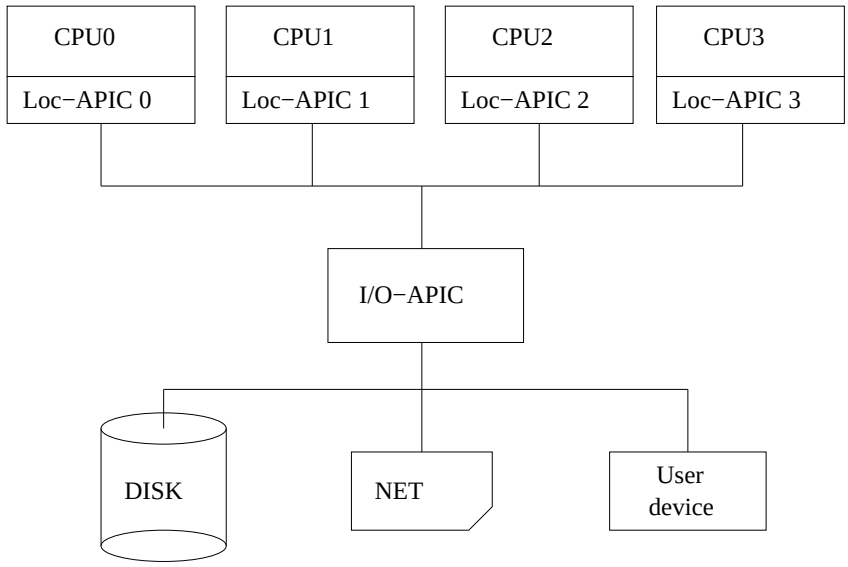

Figure 2.5: A SMP using Intel IO-APIC

asynchronously with respect to the process currently executed on a CPU. In order to understand how ASMP-Linux manages this problem, a brief introduction on how interrupts are delivered to processors is required.

Most uni-processor and multiprocessor systems include one or more *Interrupt Controller* chips, which are capable to route interrupt signals to the CPUs according to predefined routing policies. Two routing policies are commonly found: either the Interrupt Controller propagates the next interrupt signal to one specific CPU (using, for example, a round-robin scheduling), or it propagates the signal to all the CPUs. In the latter case, the CPU that first stops the execution of its process and starts to execute the interrupt handler sends an acknowledgment signal to the Interrupt Controller, which frees

the others CPUs from handling the interrupt. Figure 2.5 shows a typical configuration for a multiprocessor system based on the IA-32 architecture.

A shielded process must receive only interrupts coming from selected, crucial hardware devices, otherwise the real-time application executing on the shielded processor will be affected by some unpredictable jitter. Fortunately, recent Interrupt Controller chips—such as the *I/O Advanced Programmable Interrupt Controller* (I/O-APIC) found in the Intel architectures—can be programmed in such a way to forward interrupt signals coming from specific IRQ lines to a set of predefined processors.

Thus, the ASMP-Linux patch instruments the Interrupt Controller chips to forward general interrupts only to non-shielded CPUs, while the interrupts assigned to a given real-time partition are sent only to the corresponding shielded CPU.

However, a shielded processor can also receive interrupt signals that do not come from an Interrupt Controller at all. In fact, modern processors include an internal interrupt controller—for instance, in the Intel processors this component is called *Local APIC*. This internal controller receives the signals coming from the external Interrupt Controllers and sends them to the CPU core, thus interrupting the current execution flow. However, the internal controller is also capable to directly exchange interrupt signals with the interrupt controllers of the other CPUs in the system; these interrupts are said *Inter–Processor Interrupts*, or IPI. Finally, the internal controller could also generate a periodic self-interrupt, that is, a clock signal that will periodically interrupt the execution flow of its own

CPU core. This interrupt signal is called *local timer*.

In multiprocessor systems based on Linux, Inter–Processor Interrupts are commonly used to force all CPUs in the system to perform synchronization procedures or load balancing across CPUs, while the local timer is used to implement the time-sharing policy of each processor. As discussed in the previous sections, in ASMP-Linux load balancing never affects shielded CPUs. Furthermore, it is possible to disable the local timer of a shielded CPU altogether. Of course, this means that the time-sharing policy across the processes running in the corresponding real-time partition is no longer enforced, thus the real-time tasks must implement some form of cooperative scheduling.

2.4.4 Real-time inheritance

During its execution, a process could invoke kernel services by means of system calls. The ASMP-Linux patch slightly modifies the service routines of a few system calls, in particular those related to process creation and removal: `fork()`, `clone()`, and `exit()`. In fact, those system calls affect some data structures introduced by ASMP-Linux and associated with the process descriptor.

As a design choice, a process transmits its property of being part of a real-time partition to its children; it also maintains that property even when executing an `exec()`-like system call. If the child does not actually need the real-time capabilities, it can move itself to the non-real-time partition (see next section).

2.4.5 ASMP-Linux interface

ASMP-Linux provides a simple `/proc` file interface to control which CPUs are shielded, as well as the real-time processes and interrupts attached to each shielded CPU. The interface could have been designed as system calls but this choice would have made the ASMP-Linux patch less portable (system calls are universally numbered) and more intrusive.

Let's suppose that the system administrator wants to shield the second CPU of the system (CPU1), and that she wants to assign to the new real-time partition the process having PID `X` and the interrupt vector `N`. In order to do this, she can simply issue the following shell commands:

```
/bin/echo 1 > /proc/asmp/cpu1/shielded
/bin/echo X > /proc/asmp/cpu1/pids
/bin/echo N > /proc/asmp/cpu1/irqs
```

The first command makes CPU1 shielded.[1] The other two commands assign to the shielded CPU the process and the interrupt vector, respectively. Of course, more processes or interrupt vectors can be added to the real-time partition by writing their identifiers into the proper `pids` and `irqs` files as needed.

To remove a process or interrupt vector it is sufficient to write the corresponding identifier into the proper `/proc` file prefixed by the minus sign ("`-`"). Writing `0` into the file

[1] Actually, the first command could be omitted in this case, because issuing either the second command or the third one will implicitly shield the target CPU.

```
/proc/asmp/cpu1/shielded
```

turns the real-time partition off: any process or interrupt in the partition is moved to the non-real-time partition, then the CPU is unshielded.

The `/proc` interface also allows the system administrator to control the local timers of the shielded CPUs. Disabling the local timer is as simple as typing:

```
/bin/echo 0 > /proc/asmp/cpu1/localtimer
```

The value written in the `localtimer` file can be either zero (timer disabled) or a positive scaling factor that represents how many ticks—that is, global timer interrupts generated by the Programmable Interval Timer chip—must elapse before the local timer interrupt is raised. For instance, writing the value ten into the `localtimer` file sets the frequency of the local timer interrupt to 1/10 of the frequency of the global timer interrupts.

Needless to say, these operations on the `/proc` interface of ASMP-Linux can also be performed directly by the User Mode programs through the usual `open()` and `write()` system calls.

2.5 Experimental data

ASMP-Linux provides a good foundation for an hard real-time operating system on COTS multiprocessor systems. To validate this claim, we performed two sets of experiments.

The first test, described in Section 2.5.2, aims to evaluate the operating system overhead of ASMP-Linux: the execution time of

a real-time process executing a CPU-bound computation is measured under both ASMP-Linux and a standard Linux 2.6.19.1 kernel, with different system loads, and on several hardware platforms.

The second test, described in Section 2.5.3, aims to evaluate the operating system latency of ASMP-Linux: the local timer is reprogrammed in such a way to raise an interrupt signal after a predefined interval of time; the interrupt handler wakes a sleeping real-time process up. The difference between the expected wake-up time and the actual wake-up time is a measure of the operating system latency. The test has been carried on under both ASMP-Linux and a standard Linux 2.6.19.1 kernel, with different system loads, and on several hardware platforms.

2.5.1 Experimental environments

Two different platforms were used for the experiments; Table 2.1 summarizes their characteristics and configurations.

ID	**Architecture**	**CPUs**	**Freq.** GHz	**RAM** GB
S1	IA-32 SMP HT	8 virt.	1.50	3
S2	IA-32 SMP	4 phys.	1.50	3
S3	IA-32 CMP	2 cores	1.83	1

Table 2.1: Characteristics of the test platforms

The first platform is a 4-way SMP Intel Xeon HT [15] system running at 1.50 GHz; every chip consists of two virtual processors (HT

stands for *HyperThreading Technology* [13]). The operating system sees each virtual processor as a single CPU. This system has been tested with HT enabled (configuration "S1") and disabled (configuration "S2").

The second platform (configuration "S3") is a desktop computer based on a 2-way CMT Intel processor running at 1.83 GHz. The physical processor chip contains two cores [17]. This particular version of the processor was the one used in laptop systems, optimized for power consumption.

The Intel Xeon HT processor is a coarse-grained multithreading processor; on the other side, the Intel Dual Core is a multi-core processor (see Section 2.1). These two platforms covered the spectrum of modern CMP/CMT processors at the time this work has been done. However, nowadays, the situation would not be much different. The Intel Dual Core technology is still in the shelves, and, more in general, all modern CPUs are using a multi-core approach. Moreover, at least for the aspects that concern this work, we can consider that there are not significant differences between the tested multi-core platform and the modern ones. Instead, for what concern the 4-way SMP Intel Xeon HT, even though multithreading technology seems to not be interesting anymore for real-time systems, the multi-chip approach is still very common. So, actually, configurations S2 and S3 still reflect the most common options we can find in the market. On the other hand, configuration S1 gives us additional information about a multithreading solution, which can be still interesting, even though the market seems taking a different direction: we will in fact observe what happens when more resources are shared among differ-

ent cores.

Finally, we believe that mostly multi-core processors are of particular interest to real-time embedded system designers. In fact, multi-core CPUs performs better than the multi-chip ones, in term of power consuming and heat dissipation. These two aspects can be very important in some embedded systems, and, for this reason, low-power versions of COTS processors have often been used in embedded systems precisely because they make the heat dissipation problems much easier to solve.

System loads

For each platform, the following system loads have been considered:

IDL The system is mostly *idle*: no User Mode process is runnable beside the real-time application being tested. This load has been included for comparison with the other system loads.

CPU *CPU load*: the system is busy executing kp CPU-bound processes, where p is the number of (virtual) processors in the system, and k is equal to 16 for the first test, and to 128 for the second test.

AIO *Asynchronous I/O load*: the system is busy executing kp I/O-bound processes, where k and p are defined as above. Each I/O-bound process continuously issues non-blocking write operations on disk.

SIO *Synchronous I/O load*: the system is busy executing kp I/O-bound processes, where k and p are defined as above. Each

I/O-bound process continuously issues synchronous (blocking) write operations on disk.

MIX *Mixed load*: the system is busy executing $\frac{k}{2}p$ CPU-bound processes, $\frac{k}{2}p$ asynchronous I/O-bound processes, and $\frac{k}{2}p$ synchronous I/O-bound processes, where k and p are defined as above.

Each of these workloads has a peculiar impact on the operating system overhead. The CPU workload is characterized by a large number of processes that compete for the CPUs, thus the overhead due to the scheduler is significant. In the AIO workload, the write operations issued by the processes are asynchronous, but the kernel must serialize the low-level accesses to the disks in order to avoid data corruption. Therefore, the AIO workload is characterized by a moderate number of disk I/O interrupts and a significant overhead due to data moving between User Mode and Kernel Mode buffers. The SIO workload is characterized by processes that raise blocking write operations to disk: each process sleeps until the data have been written on the disk. This means that, most of the times, the processes are waiting for an external event and do not compete for the CPU. On the other hand, the kernel must spend a significant portion of time handling the disk I/O interrupts. Finally, in the MIX workload the kernel must handle many interrupts, it must move large amounts of data, and it must schedule many runnable processes.

For each platform, we performed a large number of iterations of the tests by using:

N A normal (`SCHED_NORMAL`) Linux process (just for comparison).

R_w A "real-time" (`SCHED_FIFO`) Linux process statically bound on a CPU that also gets all external interrupt signals of the system.

R_b A "real-time" (`SCHED_FIFO`) Linux process statically bound on a CPU that does not receive any external interrupt signal.

A_{on} A process running inside a real-time ASMP-Linux partition with local timer interrupts enabled.

A_{off} A process running inside a real-time ASMP-Linux partition with local timer interrupts disabled.

The IA-32 architecture could not reliably distribute the external interrupt signals across all CPUs in the system (this was the well-known "I/O APIC annoyance" problem). Therefore, two sets of experiments for real-time processes have been performed: R_w represents the worst possible case, where the CPU executing the real-time process handles all the external interrupt signals; R_b represents the best possible case, where the CPU executing the real-time process handles no interrupt signal at all (except the local timer interrupt). The actual performance of a production system is in some point between the two cases.

2.5.2 Evaluating the OS overhead

The goal of the first test is to evaluate the operating system overhead of ASMP-Linux. In order to achieve this, a simple, CPU-bound conventional program has been developed. The program includes a function performing n millions of integer arithmetic operations on a

tight loop (n has been chosen, for each test platform, so that each iteration last for about 0.5 sec); this function is executed 1000 times, and the execution time of each invocation is measured.

The program has been implemented in five versions (N, R_w, R_b, A_{on}, and A_{off}), and each program version has been executed on all platforms (S1, S2, and S3).

As discussed in the previous Section 1.2, the data T_x coming from the experiments are the real execution times resulting from the base time T effectively required for the computation plus any delay induced by the system. Generally speaking, $T_x = T + \varepsilon_h + \varepsilon_l + \varepsilon_o$, where ε_h is a non-constant delay introduced by the hardware, ε_l is due to the operating system latency, and ε_o is due to the operating system overhead. The variations of the values $\varepsilon_h + \varepsilon_l + \varepsilon_o$ give raise to the jitter of the system. In order to understand how the operating system overhead ε_o affects the execution time, estimations for T, ε_h, and ε_l are required.

In order to evaluate T and ε_h, the "minimum" execution time required by the function—the *base time*—has been computed on each platform by running the program with interrupts disabled, that is, exactly as if the operating system were not present at all. The base time corresponds to $T + \varepsilon_h$; however, the hardware jitter for the performed experiments is negligible (roughly, some tens of nanoseconds, on the average) because the test has been written so that it makes little use of the caches and no use at all of memory, it does not execute long latency operation, and so on. Therefore, we can safely assume that $\varepsilon_h \approx 0$ and that the base time is essentially the value T. On the S1 and S2 platforms, the measured base time was 466.649 msec,

while on the S3 platform the measured base time was 545.469 msec.

Finally, because the test program is CPU-bound and never blocks waiting for an interrupt signal, the impact of the operating system latency on the execution time is very small ($\varepsilon_l \approx 0$). One can thus assume that $T_x \approx T + \varepsilon_o$.

Therefore, in order to evaluate ε_o, the appropriate base times has been subtracted from the measured execution times. These differences are statistically summarized for the MIX workload in Tables 2.2, 2.3, and 2.4.[2] Note that in this way we also subtract the hardware overhead, ε_h, which is supposed to be roughly the same in all the iterations, from the data.

Proc	Avg	StDev	Min	Max
N	20275.784	6072.575	12.796	34696.051
R_w	28.459	12.945	10.721	48.837
R_b	27.461	9.661	3.907	42.213
A_{on}	30.262	8.306	8.063	41.099
A_{off}	27.847	7.985	6.427	38.207

Table 2.2: Operating system overheads for the MIX workload (in milliseconds) on configuration S1.

Platform S1 shows how the asymmetric approach does not provide real-time performance for HyperThreading architectures. In fact, in those processors, the amount of shared resources is significant, therefore a real-time application running on a virtual processor cannot be executed in a deterministic way regardless of the application running

[2]Results for all workloads are reported in Tables A.1, A.2, and A.3 included in appendix A.

Proc	Avg	StDev	Min	Max
N	18513.615	5996.971	1.479	33993.351
R_w	4.215	0.226	3.913	10.146
R_b	1.420	0.029	1.393	1.554
A_{on}	1.490	0.044	1.362	1.624
A_{off}	0.000	0.000	0.000	0.000

Table 2.3: Operating system overheads for the MIX workload (in milliseconds) on configuration S2.

Proc	Avg	StDev	Min	Max
N	20065.194	6095.807	0.606	32472.931
R_w	3.477	0.024	3.431	3.603
R_b	0.554	0.031	0.525	0.807
A_{on}	0.556	0.032	0.505	0.811
A_{off}	0.000	0.000	0.000	0.000

Table 2.4: Operating system overheads for the MIX workload (in milliseconds) on configuration S3.

on the other virtual processors.

The test results for platform S2 and S3, instead, clearly state that ASMP-Linux does an excellent job in reducing the impact of operating system overhead on real-time applications.

Platform S3 is the most interesting to us because provide a good *performance/cost* ratio (where *cost* is intended in both money and power consumption senses). For lack of space, in the following analysis we will focus on that platform, unless other platforms are clearly stated.

Figure 2.6 shows how platform S3 performs with the different

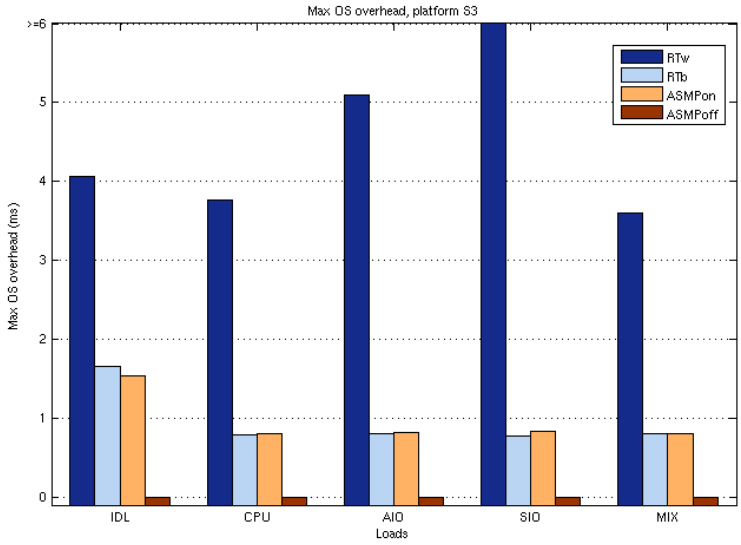

Figure 2.6: OS maximum overhead comparison

workloads and test cases R_w, R_b, A_{on}, and A_{off} (we do not show results from the N test case because its times are several orders of magnitude higher than the others). Each box in the figure represents the maximum overhead measured in all experiments performed on the specific workload and test case. Since the maximum overhead might be considered as a rough estimator for the real-time characteristics of the system, it can be inferred that all workloads present the same pattern: A_{off} is better than A_{on}, which in turn is roughly equivalent to R_b, which is finally much better than R_w. Since all workloads are alike, from now on we will specifically discuss the MIX workload—likely the most representative of a real-world scenario.

Figures 2.7, 2.8, 2.9, and 2.10 show the samples measured on system S3 with MIX workload. Each dot represents a single test;

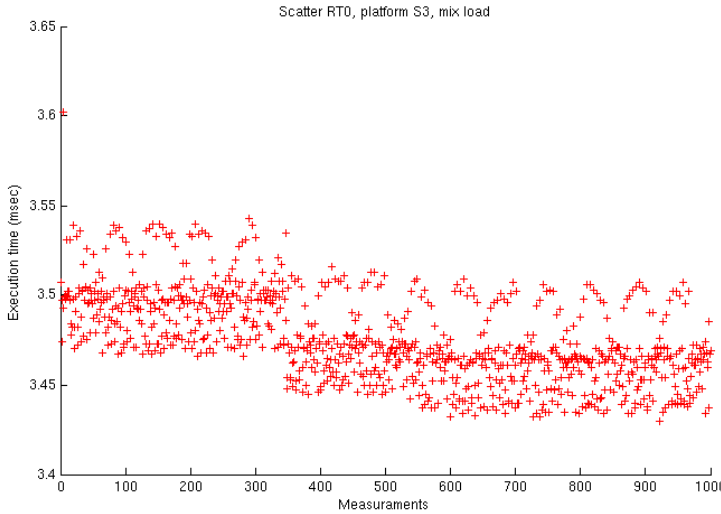

Figure 2.7: Scatter R_w graphic for system S3, MIX workload

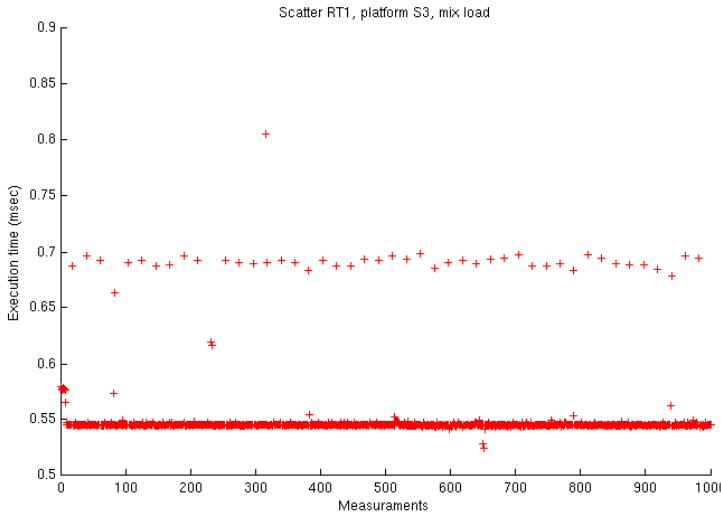

Figure 2.8: Scatter R_b graphic for system S3, MIX workload

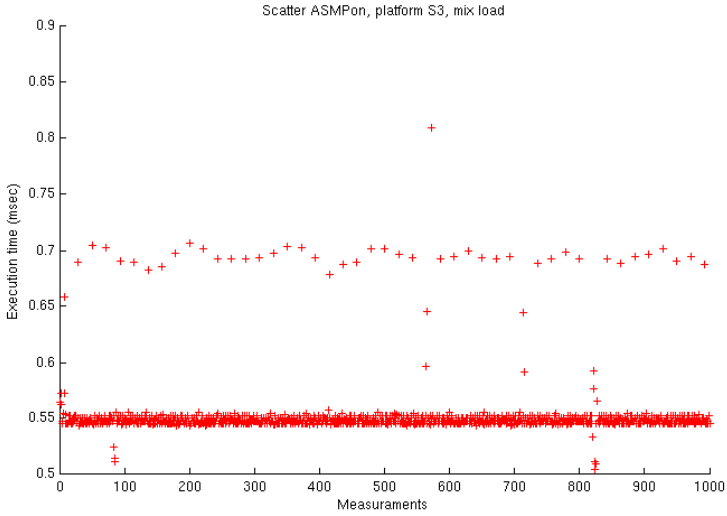

Figure 2.9: Scatter A_{on} graphic for system S3, MIX workload

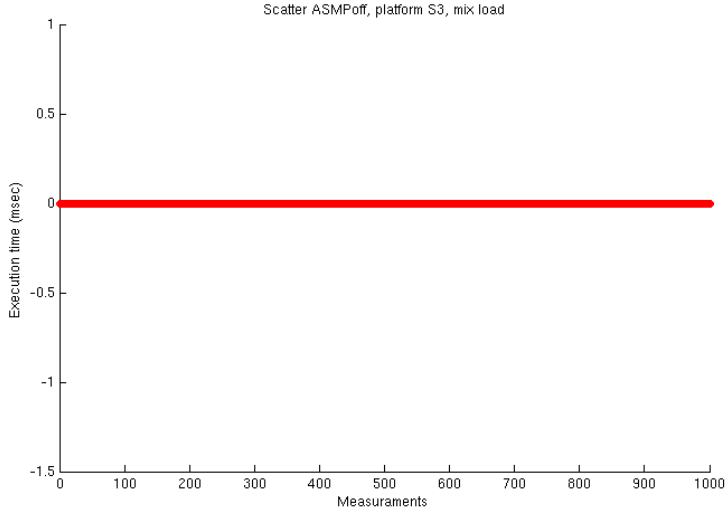

Figure 2.10: Scatter A_{off} graphic for system S3, MIX workload

its y-coordinate corresponds to the difference between the measured time and the base value, as reported in Table 2.4. (Notice that the y-axis in the four plots have different scales).

The time required to complete each iteration of the test varies according to the operating system overhead experimented in that measurement: for example, in system S3, with a MIX workload, each difference can be between 3.431 msec and 3.603 msec for the R_w test case.

Figure 2.7 clearly shows that at the beginning of the test the kernel was involved in some activity, which terminated after about 300 samples. We identified this activity in creating the processes that belonged to the workload: after some time all the processes have been created and that activity is no longer present. Figure 2.7 also shows how, for the length of the experiment, all the samples are affected by jitter, thus they are far from the theoretical performance of the platform.

Figures 2.8 and 2.9 show that the operating system overhead mainly consists of some short, regular activities: we identify those activities with the local timer interrupt (which, in fact, is not present in Figure 2.10). Every millisecond the local timer raised an interrupt (the highest priority kernel activity) and the CPU executed the interrupt handler instead of the real time application. It can be noticed that R_b performs slightly better than A_{on}. As a matter of fact, the two test cases are very similar: in R_b the scheduler always selects the test program because it has `SCHED_FIFO` priority, while in A_{on} the scheduler selects the "best" runnable process in the real-time ASMP partition—this normally means the test program itself, but in a few

cases it might also select a per-CPU kernel thread, whose execution makes the running time of the test program longer.

It is straightforward to see in Figure 2.10 how ASMP-Linux in the A_{off} version has a deterministic behavior, with no jitter, and catches the optimal performance that can be achieved on the platform (i.e., the *base time* mentioned above). On the other hand, using ASMP-Linux in the A_{on} version only provides soft real time performance, comparable with those of R_b.

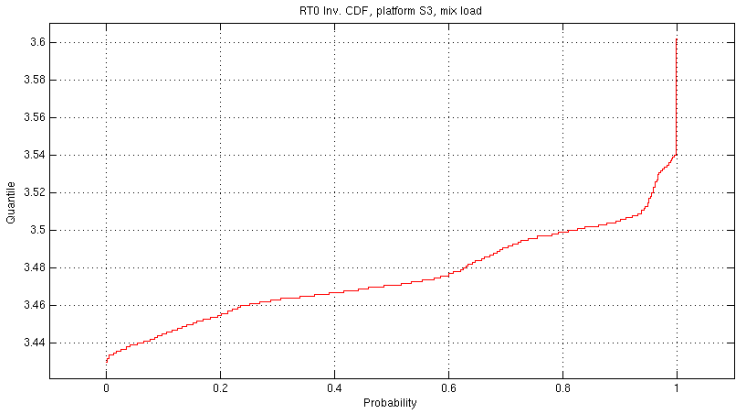

Figure 2.11: Inverse density functions for overhead on system S3, MIX workload, configuration R_w.

Figures 2.11, 2.12, 2.13, 2.14 show inverse Cumulative Densitive Function (CDF) of the probability (x-axis) that a given sample is less than or equal to a threshold execution time (y-axis). For example, in Figure 2.11, the probability that the overhead in the test is less than or equal to 3.5 msec is about 80%. We think this figure clearly explains how the operating system overhead can damage the performance of a real time system. Different operating system activities

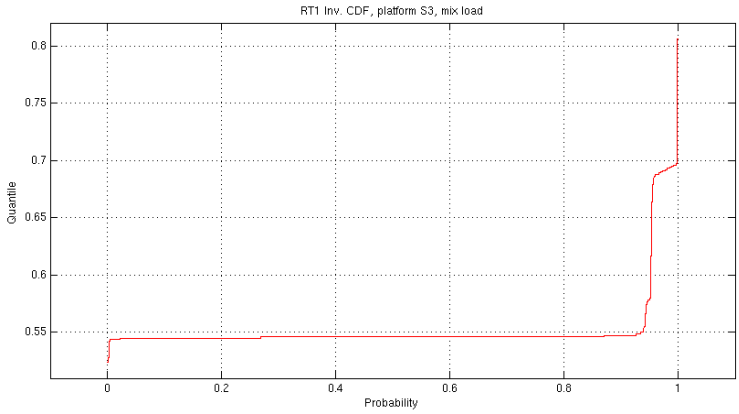

Figure 2.12: Inverse density functions for overhead on system S3, MIX workload, configuration R_b.

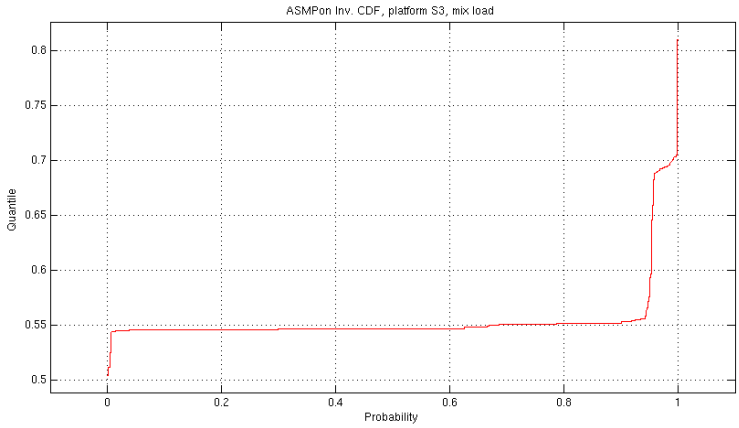

Figure 2.13: Inverse density functions for overhead on system S3, MIX workload, configuration A_{on}.

introduce different amounts of jitter during the execution of the test, resulting in a non-deterministic response time. Moreover, the figure states how the maximum overhead can be significantly higher than

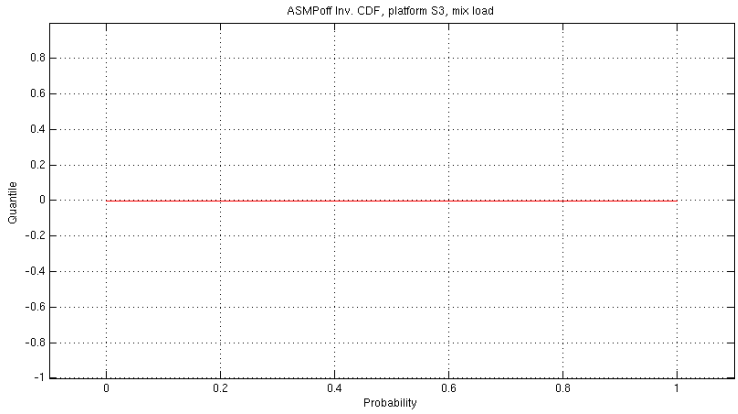

Figure 2.14: Inverse density functions for overhead on system S3, MIX workload, configuration A_{off}.

the average operating system overhead. Once again, the figure shows how ASMP-Linux in the A_{on} version is only suitable for soft real time application, as well as R_b.

2.5.3 Evaluating the operating system latency

The goal of the second test is to evaluate the operating system latency of ASMP-Linux. In order to achieve this, the local timer (see Section 2.4.3) has been programmed in such a way to emulate an external device that raises interrupts to be handled by a real-time application.

In particular, a simple program that sleeps until awakened by the operating system has been implemented in five versions (N, R_w, R_b, A_{on}, and A_{off}). Moreover, a kernel module has been developed in order to simulate an hardware device: it provides a device file that

can be used by a User Mode program to get data as soon as they are available. The kernel module reprograms the local timer in such a way to raise a one-shot interrupt signal after a predefined time interval. The corresponding interrupt handler awakes the process blocked on the device file and returns to it a measure of the time elapsed since the timer interrupts occurred.

The data coming from the experiments yield the time elapsed since the local timer interrupt is raised and the User Mode program starts to execute again. Each test has been repeated 10 000 times; the results are statistically summarized for the MIX workload in Table 2.5.[3]

The delay observed by the real-time application is $\varepsilon_h + \varepsilon_l + \varepsilon_o$. Assuming as in the previous test $\varepsilon_h \approx 0$, the observed delay is essentially due to the operating system overhead and to the operating system latency. Except for the case "N", one can also assume that the operating system overhead is very small because, after being awoken, the real time application does not do anything but issuing another read operation from the device file. This means that the probability of the real-time process being interrupted by any kernel activity in such a small amount of time is very small. In fact, the real-time application is either the only process that can run on the processor (A_{on} and A_{off}), or it has always greater priority than the other processes in the system (R_w and R_b). Thus, once awakened, the real-time task is selected right away by the kernel scheduler and no other process can interrupt it. Therefore, the delays shown in Table 2.5 are essentially

[3]Results for all workloads are reported in Tables A.4, A.5 and A.6 included in appendix A.

Proc	Avg	StDev	Min	Max
N	13923.606	220157.013	6.946	$5.001 \cdot 10^6$
R_w	10.970	8.458	6.405	603.272
R_b	10.027	5.292	6.506	306.497
A_{on}	8.074	1.601	6.683	20.877
A_{off}	8.870	1.750	6.839	23.230

(a) Configuration S1

Proc	Avg	StDev	Min	Max
N	24402.723	331861.500	4.904	$4.997 \cdot 10^6$
R_w	5.996	1.249	4.960	39.982
R_b	5.511	1.231	4.603	109.964
A_{on}	5.120	0.275	4.917	9.370
A_{off}	5.441	0.199	5.207	6.716

(b) Configuration S2

Proc	Avg	StDev	Min	Max
N	182577.713	936480.576	1.554	$9.095 \cdot 10^6$
R_w	1.999	1.619	1.722	66.883
R_b	1.756	0.650	1.548	63.985
A_{on}	1.721	0.034	1.674	3.228
A_{off}	1.639	0.025	1.602	2.466

(c) Configuration S3

Table 2.5: Operating system latencies for the MIX workload (in microseconds)

due to both the interrupt and the scheduler latency of the operating system.

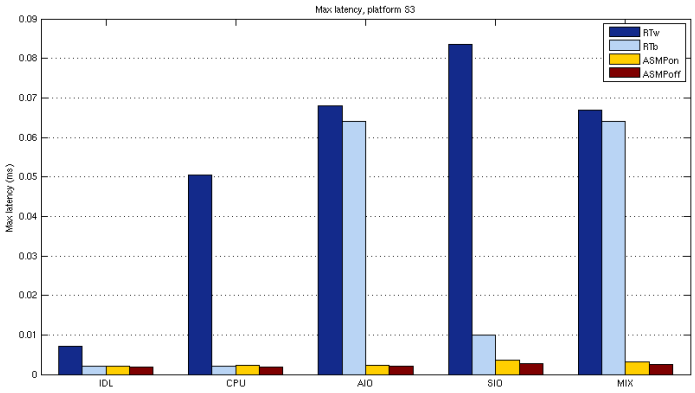

Figure 2.15: OS maximum latency comparison

Figure 2.15 shows how platform S3 performs with the different workloads and test cases R_w, R_b, A_{on}, and A_{off} (we do not show results from the N test case because its times are several orders of magnitude higher than the others). Each box in the figure represents the maximum latency measured in all experiments performed on the specific workload and test cases.

As we said, the probability that some kernel activity interrupts the real time application is very small, yet not null. An Inter–Processor Interrupt (IPI) could still be sent from one processor to the one running the real time application (even for the R_b test) in order to force process load balancing. This is, likely, what happened to R_w and R_b, since they experiment a large, isolated maximum.

As in the previous test, from now on we will restrict ourselves in discussing the MIX workload, which we think is representative of all

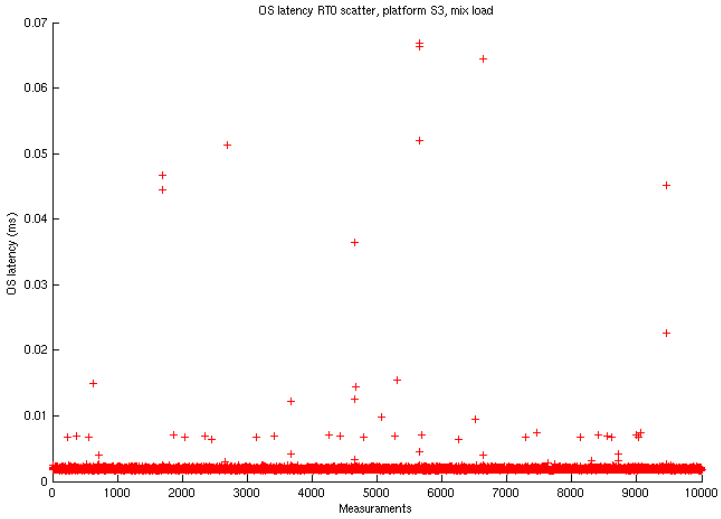

Figure 2.16: Scatter R_w graphics for system S3, MIX workload.

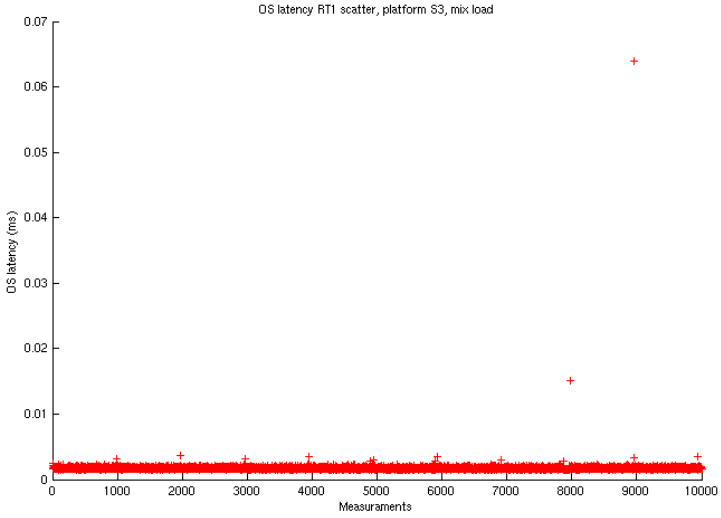

Figure 2.17: Scatter R_b graphics for system S3, MIX workload.

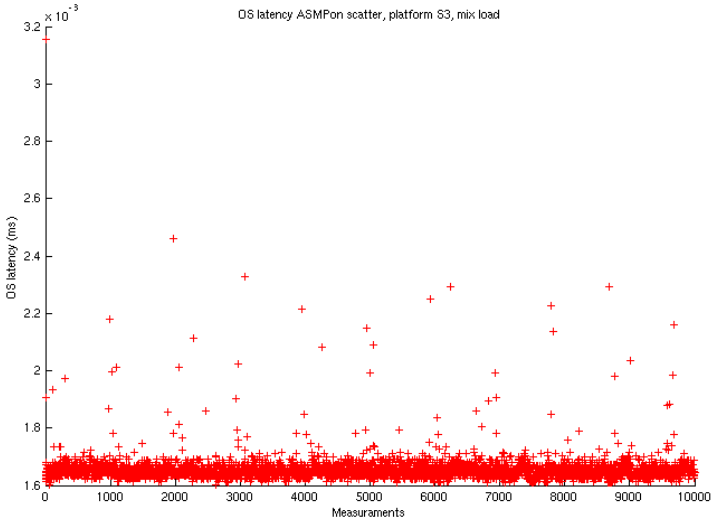

Figure 2.18: Scatter A_{on} graphics for system S3, MIX workload.

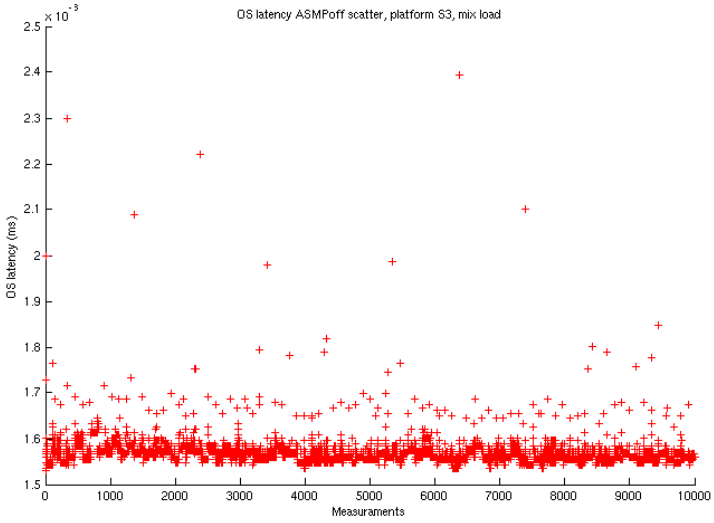

Figure 2.19: Scatter A_{off} graphics for system S3, MIX workload.

the workloads (see Tables A.4, A.5 and A.6 included in appendix A).

Figures 2.16, 2.17, 2.18, and 2.19 show the samples measured on system S3 with MIX workload. Each dot represents a single test; its y-coordinate corresponds to the latency time, as reported in Table 2.5. (The y-axis in the four plots have different scales; thus, for example, the scattered points in Figure 2.19 would appear as a straight horizontal line on Figure 2.17).

Figures 2.20, 2.21, 2.22, and 2.23 show inverse Cumulative Densitive Function (CDF) of the probability (x-axis) that a given sample is less than or equal to a threshold execution time (y-axis). For example, in Figure 2.23, the probability that the latency measured in the test is less than or equal to 1.6 μsec is about 98%. In the A_{off} test case a small jitter is still present; nonetheless, it is so small that it could be arguably tolerated in many real-time scenarios.

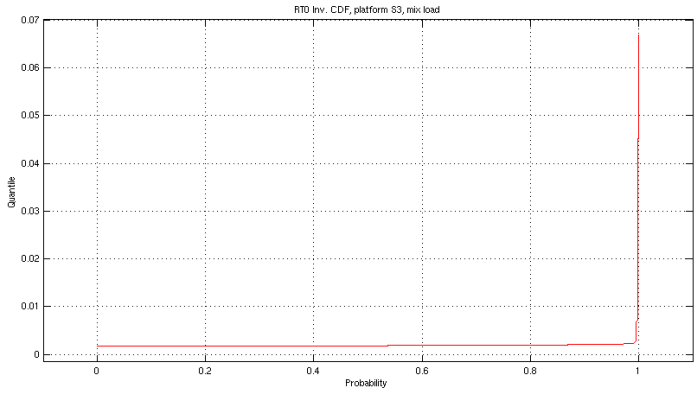

Figure 2.20: Inverse density functions for latency on system S3, MIX workload, configuration R_w.

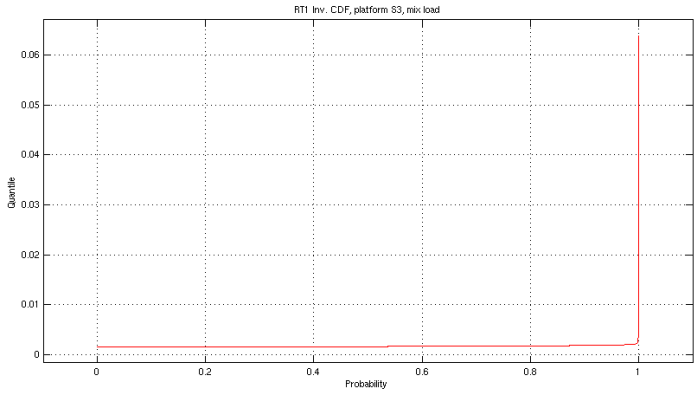

Figure 2.21: Inverse density functions for latency on system S3, MIX workload, configuration R_b.

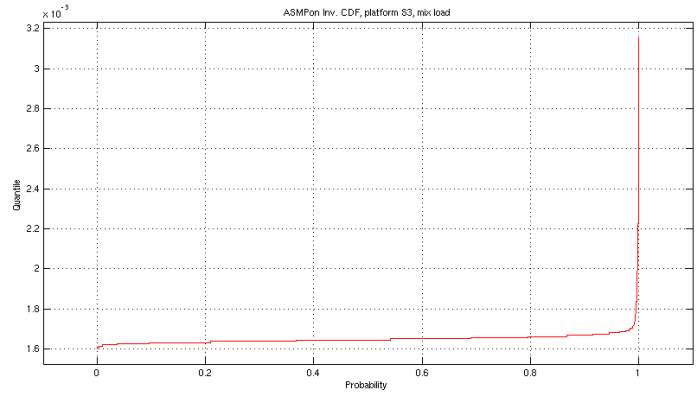

Figure 2.22: Inverse density functions for latency on system S3, MIX workload, configuration A_{on}.

2.5.4 Final consideration

The goal of these tests was to evaluate ASMP-Linux on different platforms. In fact, each platform has benefits and drawbacks: for example, platform S1 is the less power consuming architecture because

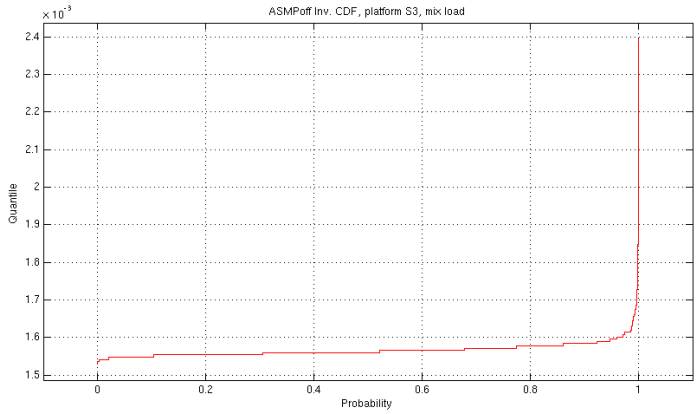

Figure 2.23: Inverse density functions for latency on system S3, MIX workload, configuration A_{off}.

the virtual processors are not full CPUs; however, as we expected, ASMP-Linux does not provide hard real-time performances on this platform. Conversely, ASMP-Linux provides hard real-time performances when running on platform S2; however, this platform is the most expensive in terms of cost, surface, and power consumption, thus we do not think it will fit well with embedded systems' constraints. Platform S3 is a good tradeoff between the previous two platforms: ASMP-Linux still provides hard real-time performance even if the two cores share some resources, resulting in reduced chip surface and power consumption. Moreover, the tested processor has been specifically designed for power-critical system (such as laptops), thus we foreseen it could be largely used in embedded systems, as it happened with its predecessor single-core version.

Chapter 3

Real-time bridge

In chapter 2 we saw how the software jitter, described in section 1.4, can be minimized by using ASMP-Linux. However, working only at the operating system level, it is not possible to reduce the other important source of unpredictability, that is *the hardware jitter*, introduced in section 1.5.

In this chapter we describe a novel real-time I/O management system, designed to address this problem. In particular, we focus on the jitter observed by peripherals when they access main memory.

We know that COTS buses (like PCI, or PCI-E) do not guarantee timeliness, and the system may experience severe timing degradation in the presence of high-bandwidth I/O peripherals (see [45, 44]). The proposed framework introduces two new components in the traditional I/O architecture: a *real-time bridge*, and a *reservation controller*. These are used to transparently put the I/O subsystem of a COTS-based embedded system under the discipline of real-time scheduling. This work has been published in the proceedings of the 30th IEEE Real-Time Systems Symposium [4].

The chapter is organized as follows. First, in section 3.1, we discuss the modern COTS I/O subsystems and the current real-time I/O requirements. In section 3.2, we elaborate on the design of the

novel real-time I/O management system. In section 3.3 the supported real-time scheduling algorithms are discussed. Sections 3.4 and 3.5 give detailed information on the new components we developed. In section 3.6 we focus on the limitations of our architecture. In section 3.7, we demonstrate, on physical hardware, that timing violations can occur without our real-time I/O management system, but with our system we can prevent I/O deadline misses. We finish with related work in section 3.8.

3.1 COTS I/O subsystems and real-time

As we discussed in chapter 1, integrating high-speed COTS peripherals within a real-time system offers several benefits like cost reduction, time-to-market, and overall performance. On the other hand, it can be a daunting task, since COTS components are typically designed paying attention to average performance and not to worst-case timing behaviors.

In this chapter, we focus on I/O subsystems with high bandwidth requirements; a modern real-time system such as a search and rescue helicopter [52] may include several high-bandwidth components such as a Doppler navigation system, a forward look-ahead infrared radar, a night vision system, and several types of communication systems. Modern I/O components such as these can inject significant traffic onto the I/O bus. For example, a single real-time high-definition video may consume an I/O bandwidth of tens to hundreds of Mbps [1].

While priority-based real-time scheduling is a standard practice

for the CPU, it is currently not supported by COTS peripherals and interconnect systems (e.g., PCI bus [42]). Due to the lack of real-time prioritization, data I/O transactions traveling through the COTS bus into or out of main memory can suffer unpredictable delay and cause deadline misses [41]. Unfortunately, end-to-end real-time guarantees can not be achieved unless both tasks and I/O data transactions are properly processed in a prioritized manner. We address this challenge by introducing a real-time I/O management system that supports a wide range of priority-based scheduling policies, retains backward compatibility with existing COTS-based components, and achieves high real-time bus utilization without degrading peripherals' throughput.

The proposed framework acts like a "transparent layer" that does not add any additional burden at the operating system or user level, except for assigning a certain priority to each real-time I/O flow. Such I/O flows may be highly variable depending on content, so that a worst-case I/O interference pattern may not occur during initial testing. Furthermore, overprovisioning for transient, but rare load spikes may raise costs and lead to an underutilized system; on the other hand, ignoring this problem can cause dangerous system instabilities and even deadline misses of safety critical real-time tasks [59]. Predictability within the I/O subsystem is essential to meet the demand of future cyber-physical systems.

3.2 Real-time I/O Management System Design

Before describing our real-time I/O management system for predictable I/O performance on a COTS system, we first describe the way in which a COTS system typically works.

A COTS system may include several commercial peripherals, such as video acquisition boards or network cards, plugged into standard buses, such as PCI or PCI-E, on a commercial motherboard. As figure 3.1 shows, data from these boards would travel through a series of bridges and buses (the specifics depend on the model of the motherboard), until it reaches main memory, where the CPU can read it through the Front Side Bus (FSB). Alternatively, the CPU could write data into main memory and instruct the COTS peripherals to retrieve it. For example, a network card could be instructed to upload packets which are stored in RAM.

Our proposed real-time I/O management system, shown in context in Figure 3.2, adds two types of components to the existing COTS system.

The first type is a *reservation controller*, which implements the system-wide policy for accessing the bus. It can be thought of as a high-level arbiter which instructs the real-time bridges to either communicate on the bus, or yield to other devices.

The other type of component we introduce is a *real-time bridge* which is interposed between each peripheral and the communication bus. Each real-time bridge provides the actuation mechanism to

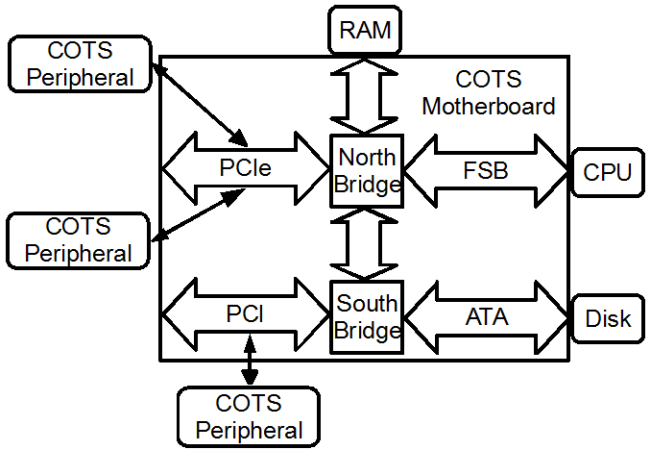

Figure 3.1: A common COTS system architecture.

enforce peripheral bus access.

For rapid development, we implemented both of these components in hardware on field programmable gate arrays (FPGAs), although an industrial application would likely use an Application Specific Integrated Circuit (ASIC). We describe these devices in section 3.4 and section 3.5 and elaborate on details involved in making the prototypes.

3.3 Enforcing real-time scheduling

We now describe the way in which our proposed architecture can be used to provide real-time guarantees for peripheral traffic. Particularly, we are interested in determining the classes of schedulers that can be implemented using multiple real-time bridges connected to a reservation controller. We address the following question: can

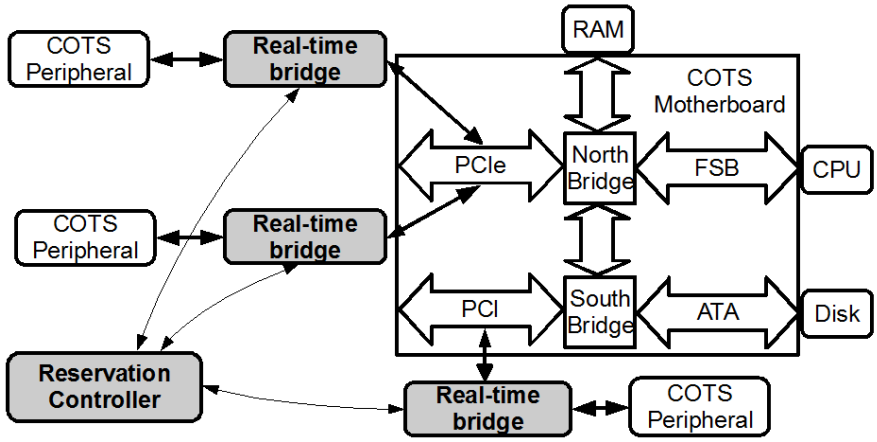

Figure 3.2: The proposed real-time I/O management system adds a reservation controller and real-time bridge to the COTS-based node.

any real-time monoprocessor scheduling algorithm be used with our hardware framework? Although we show that the answer is no, we are still able to employ a large class of monoprocessor schedulers.

Since there are many classifications of scheduling algorithms, we start by showing a definition that provides a more formal framework on which to reason about implementable schedulers.

Definition 4 (Scheduling Servers / Scheduling Algorithms)**.** *Consider a set of tasks requesting access to a single shared resource. Each task (or group of tasks) has an associated* scheduling server *which, based on the task's (tasks') activity, forwards scheduling parameters to the* scheduling algorithm*. At any time instant t, the scheduling algorithm grants the shared resource to at most one scheduling server based on the value of all received scheduling parameters.*

The provided definition is general enough to encompass all common real-time schedulers for a single shared resource. For example, consider scheduling periodic tasks according to *rate monotonic*. The scheduling server for each task forwards a STATIC_PRIORITY parameter equal to the inverse of the task period, and a READY boolean parameter which is true if and only if the task has remaining computation time. The fixed priority scheduling algorithm then selects the task with highest STATIC_PRIORITY and a true READY parameter to execute.

A budget-based server for aperiodic tasks, such as a *sporadic server*, would be more complex, but would still output the same STATIC_PRIORITY and READY parameters. In particular, the server's READY parameter is true if and only if the served task has remaining execution and there is available budget. However, the set of scheduling parameters changes based on the scheduling algorithm. For instance, an *EDF* server would need to output both a READY boolean value and a dynamic ABSOLUTE_DEADLINE parameter.

The logical division between the scheduling servers and the scheduling algorithm corresponds to the physical implementation within the reservation controller. Each scheduling server is implemented by a single VHDL hardware module. The module receives the `data_rdy` signal from the corresponding real-time bridge and computes the scheduling parameters (the boolean READY value and the STATIC_PRIORITY value). Global scheduling logic, such as the logic shown in code block 3.1, then implements the scheduling algorithm, receiving scheduling parameters and producing `block` signals for all scheduling servers. However, in the case of fixed priority scheduling, STATIC_PRIORITY

is a design-time parameter which is absorbed in the implementation of the global scheduling logic as an optimization.

The main drawback of such a design, however, is that each scheduling server must make scheduling decisions based only on the task's `data_rdy` (active) value.

Definition 5 (Active-Dynamic Server)**.** *A scheduling server is an* active-dynamic server *if run-time task behavior can be governed using only one piece of task-related knowledge, whether the task is active (immediately executable) or not.*

Proposition 1. *Our I/O scheduling mechanism can implement any scheduling framework where all scheduling servers are active-dynamic servers.*

Next, we provide some examples of servers that are active-dynamic, and some examples of servers that are not.

Lemma 1. *Under fixed priority scheduling, both a periodic task server and the sporadic server are active-dynamic servers.*

Proof. A scheduling server servicing a periodic task needs only output the STATIC_PRIORITY for the task, and a dynamic READY parameter which is equal to the active value of the task.

A sporadic server is active-dynamic because the rules governing its replenishment time and replenishment amount can be obeyed knowing only the task's active value (and other non-task parameters such as the current time). According to the rules of a sporadic server, the replenishment time is determined as soon as the task becomes active (an aperiodic task requests execution) and there is available

budget. The replenishment time is computed by simply summing the current time with the static server period. The replenishment amount, computed when the server becomes inactive, is equal to the consumed budget. This is computed by measuring the duration of time the task was both active and unblocked (which can be given through feedback from the scheduling algorithm). Determining when the server becomes inactive is done by checking if the task is finished or the budget is exhausted. Since all the server rules can be evaluated based only on the active status of the task, a sporadic server is an active-dynamic server. □

Lemma 2. *Consider a sporadic task feasibly scheduled under EDF with relative deadline greater than its interarrival time. The server for such task is not an active-dynamic server.*

Proof. Consider two successive jobs j and $j+1$ of the sporadic task. Since the interarrival time between jobs is less than the relative deadline, job $j+1$ could arrive in the system before job j is finished. Since the server has no way to know when $j+1$ arrived based on the active status of the task, it can not set the deadline for job $j+1$. □

Also notice that there exist non budget-based servers, such as the total bandwidth server [63], which require arrival-time information about aperiodic job execution time and therefore are not active-dynamic servers.

3.4 Reservation Controller

Multiple peripherals must cooperate to prevent a timeliness reduction caused by mutual interference. The reservation controller centralizes decision making and coordinates multiple real-time bridges by instructing them to either forward or buffer peripheral traffic.

Presently, we consider each peripheral as generating a single real-time I/O flow. In chapter 4, we discuss a real-time bridge extension that supports device virtualization and allows multiple real-time I/O flows per physical device (with potentially different priorities). Furthermore, we only consider I/O flows directed to main memory, and not to other peripherals[1].

As described in section 3.1, and shown later in our evaluation in section 3.7, allowing multiple COTS peripherals to simultaneously access main memory can result in unpredictable bandwidth allocation. As such, we allow only a single real-time flow to transmit at any one time. Therefore, we can consider the time allocated among all real-time bridges by the reservation controller as a shared resource akin to a monoprocessor CPU. In this analogy, each I/O flow is equivalent to a real-time task, and each I/O data chunk in the flow is equivalent to a job; transfer times for I/O data chunks are equated to computation times and can typically be derived by dividing the I/O data amount by the achievable throughput of the real-time bridge.

Coordination between the reservation controller and each real-time bridge is achieved using two physical wires. Each real-time

[1] Our methodology will be extended to cover inter-peripheral communication in our future work.

bridge communicates one boolean value, `data_rdy`, to the reservation controller. This value indicates that data is buffered and ready to be sent on the bus. In turn, the reservation controller sends one boolean value back to the real-time bridge, `block`, which instructs the bridge to either block I/O traffic or permit bus access. Real-time bridges instructed to block do not attempt to gain access to the bus, mandating the bus arbiter grants the unblocked peripherals access to the bus to send their data.

With only these two signals, many kinds of bus scheduling policies can be enforced. Consider, for example, scheduling four real-time bridges according to a static-priority bus scheduling scheme, such as rate monotonic (RM). Let $\texttt{block}_i$ be the block command sent to the ith real-time bridge, and let $\texttt{data\_rdy}_i$ be the indicator of buffered data coming from the ith real-time bridge. Let the bridges be physically connected to the reservation controller in the order of their priorities (in the order of their rates for RM), from the highest priority bridge, $i = 0$, to the lowest priority bridge, $i = 3$. In order to provide static-priority scheduling on the I/O bus, the reservation controller hardware would implement the logical expressions in code block 3.1.

Our framework, however, can also support a large class of mono-processor scheduling algorithms which handle sporadic and aperiodic tasks using real-time servers [11]. In our prototype, for instance, we have implemented support for sporadic servers [3] under fixed-priority scheduling (see section 3.4.1). Notice that the servers are implemented on the reservation controller and not on the associated real-time bridges. This decision has two major advantages.

Code 3.1 These logical expressions, implemented in hardware on the reservation controller, provide a static-priority I/O scheduler for a four-peripheral system.

$$\texttt{block}_0 := \neg(\texttt{data\_rdy}_0)$$

$$\texttt{block}_1 := \neg(\texttt{block}_0 \wedge \texttt{data\_rdy}_1)$$

$$\texttt{block}_2 := \neg(\texttt{block}_0 \wedge \texttt{block}_1 \wedge \texttt{data\_rdy}_2)$$

$$\texttt{block}_3 := \neg(\texttt{block}_0 \wedge \texttt{block}_1 \wedge \texttt{block}_2 \wedge \texttt{data\_rdy}_3)$$

First, it removes the need for precise clock synchronization among real-time bridges and the reservation controller. Since all scheduling servers use the same physical clock, server budgets can be precisely calculated without clock skew.

Second, it simplifies the interface between each real-time bridge and the reservation controller by reducing the number of physical wires to just two, `data_rdy` and `block`. This becomes a concern for algorithms like EDF, where each server must communicate a precise deadline timestamp to the scheduling algorithm, which requires dozens of bits of information. By centralizing all scheduling servers on the reservation controller, the number of physical wires is reduced, simplifying the electrical design.

3.4.1 Prototype Details

In our prototype, the reservation controller is built using the Xilinx ML505 Evaluation Platform [74] which features an XC5VLX50T FPGA. We created VHDL hardware code to implement the rate monotonic scheduling algorithm [36] for strictly periodic tasks, as well the sporadic server algorithm [62] to schedule aperiodic periph-

eral bus traffic.

The implementation itself is split into two types of hardware components:

- servers for each real-time bridge,
- a global scheduling algorithm.

Each server generates a READY signal, and the global scheduling algorithm in this case executes the task with the highest static priority and an asserted READY signal.

Logically speaking, the parameter generating servers output values to the global scheduling algorithm, which then chooses one of them for execution (although in practice some design-time values such as the STATIC_PRIORITY become absorbed in the global scheduling algorithm as an optimization). For example, the parameters generated by both the strictly periodic server and the sporadic server are READY and STATIC_PRIORITY, although the logic necessary to compute these are more complicated in the case of sporadic servers. Since the parameters output are the same, we can mix sporadic servers and strictly periodic servers within the same global scheduling algorithm. The global scheduling algorithm in this case executes the task with the highest STATIC_PRIORITY and an asserted READY signal.

In terms of FPGA area utilization, our implementation is lightweight. We synthesized a two layer reservation controller with four sporadic servers scheduled according to a global RM scheduling policy. The resultant implementation used 531 of the 28,800 available slice registers (1.8%), and 668 of the available 28,800 slice look up tables (LUTs)

(2.3%). The number I/O buffers used, which is associated with the chip's pinout, was also low (9 out of 480, 1.9%). These numbers lead us to conclude that the design will easily scale to concurrently manage dozens of I/O peripherals.

3.5 Real-time bridge

In order to provide real-time guarantees on bus communication, bus access must be controlled according to the policy dictated by the reservation controller. Since off-the-shelf peripherals are unlikely to have such a mechanism built-in, we interpose a device between each peripheral and the bus in order to provide this functionality to our real-time I/O management system. In addition to restricting bus access, *real-time bridges* also provide an important additional service to connected peripherals. Each real-time bridge provides a buffer which is able to store pending traffic while bus access is prohibited. This allows high-bandwidth peripherals to be blocked from the bus for relatively long periods of time without suffering from data loss due to full internal buffers on the COTS peripheral. Combined with a communication guarantee provided by the reservation controller's scheduling policy, this guarantees the I/O system will deliver all communication traffic by its I/O deadline.

3.5.1 Prototype Details

We envision a final general real-time bridge using a setup similar to the ML455 [76] (see figure 3.3), an FPGA development platform

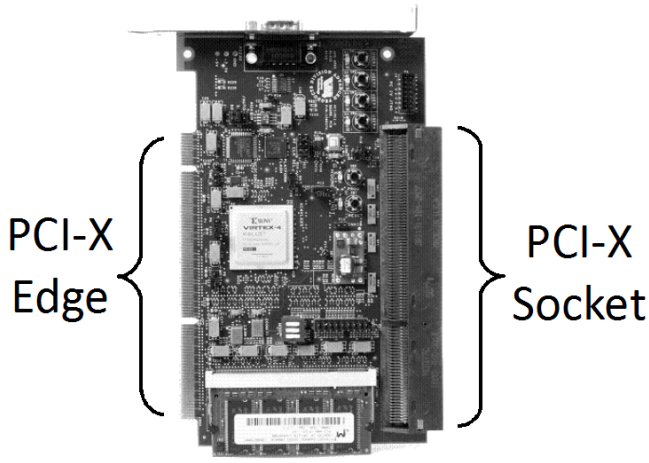

Figure 3.3: Since the ML455 FPGA development platform contains both a PCI-X edge and a PCI-X socket, it can be interposed between a PCI-X peripheral and a PCI-X COTS bus and therefore function as a real-time bridge.

which can be directly interposed between a COTS bus and a COTS peripheral. This device contains both a PCI-X edge connector and a PCI-X socket slot connected to the same Virtex 4 FPGA chip, which would allow various types of peripherals to use the exact same real-time bridge.

However, in order to rapidly develop a complete prototype, we focused our effort on a real-time bridge for one specific peripheral. We targeted a network interface card on the ML505 FPGA Evaluation Platform [74]. This device features both an Ethernet hardware interface as well as a one-lane PCI-E edge connector both connected to the same Virtex 5 FPGA chip.

We first describe the hardware components of the System-on-Chip

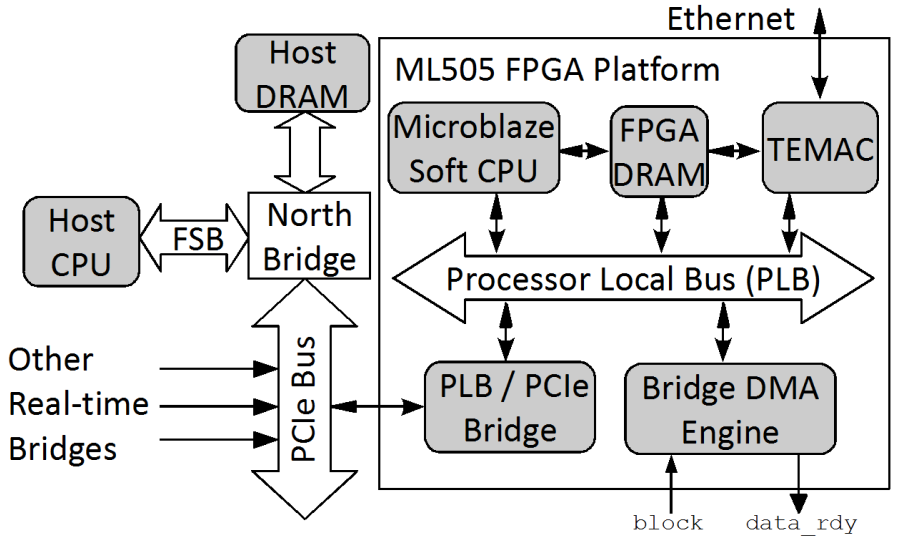

Figure 3.4: Our real-time bridge prototype is a System-on-Chip implemented on the ML505 FPGA Evaluation Platform.

(SoC) design, and then elaborate on the software development effort.

Hardware design

A logical outline of the important hardware components in the network interface card version of the real-time bridge is shown in Figure 3.4. We now describe each of these in the order of the dataflow through the real-time bridge during normal operation. Consider the case where a packet arrives through the Ethernet connection.

First, the physical hardware interacts with the **Tri-state Ethernet MAC (TEMAC)** hardware block. This is a fixed hardware block on the FPGA, and is the COTS peripheral that the real-time bridge is managing. This block maintains a set of memory addresses where to place packets after they are received. After the packet ar-

rives to the TEMAC, it gets stored into the **FPGA DRAM** and an interrupt is raised to the **Microblaze Soft CPU** [75] which provides information on where the packet was stored and the size of the received data.

The Microblaze processor is a soft CPU, meaning that is implemented using the reconfigurable logic on the FPGA. We developed a driver running on the Microblaze that will take the addresses and lengths of the packets and put them into a download queue of data to be sent to the main system. This queue exists in two parts. The potentially long tail of the queue is stored into the **FPGA DRAM**, whereas a bounded number of entries (say 128) of the front of the queue are stored in hardware on the **bridge DMA engine**.

The bridge DMA engine, which is a hardware block we created specifically for the real-time bridge, manages actually moving the data out of FPGA DRAM and transferring it into the host CPU's main memory. Along with the queue of data needing to be transferred, the bridge DMA engine manages the `block` and `data_rdy` signals on the real-time bridge. When `block` is asserted, the bridge DMA engine will not transfer data out of FPGA memory. The `data_rdy` signal is asserted whenever any data is in the hardware queue.

The DMA transfers themselves are abstracted as an address to address copy on the Processor Local Bus (PLB). The **PLB / PCI-E bridge** handles the process of translating a write transaction on the PLB bus to a write transaction on the PCI-E bus. When the bridge DMA engine is unblocked by the reservation controller and performs a DMA operation, the memory containing the packet in FPGA DRAM

is copied into the **Host DRAM**. After the transaction is complete, an interrupt is raised on the **Host CPU**, which then takes the packet data and passes it to the network stack for processing.

Sending packets out from the main CPU works in a similar way, but in the reverse order. The main CPU stores the packet data in Host DRAM and then writes the addresses to an upload queue which has the tail in Host DRAM and the front part on the bridge DMA engine. When the bridge DMA engine is unblocked, it transfers the packets from Host RAM into FPGA DRAM (the PCI / PLB Bridge will again do address translation) and raises an interrupt to the Microblaze Soft Processor. Our driver on the Microblaze then sends the TEMAC hardware block the addresses and lengths of the packet data in FPGA DRAM. Finally, the TEMAC hardware sends the data over the physical Ethernet medium.

Software Design

The software architecture for the real-time bridge is paramount to our real-time I/O management system's applicability. Although the hardware design may be generalized for a particular bus interface and therefore reused, each unique COTS peripheral requires some software effort to be transparently controlled by a real-time bridge. In particular, each peripheral requires two drivers, a *host driver* to run on the main CPU, and an *FPGA driver* to run on the real-time bridge bridge's Microblaze Soft CPU. A logical layout of their interactions is show in Figure 3.5.

One advantage of using COTS peripherals is that it is common for

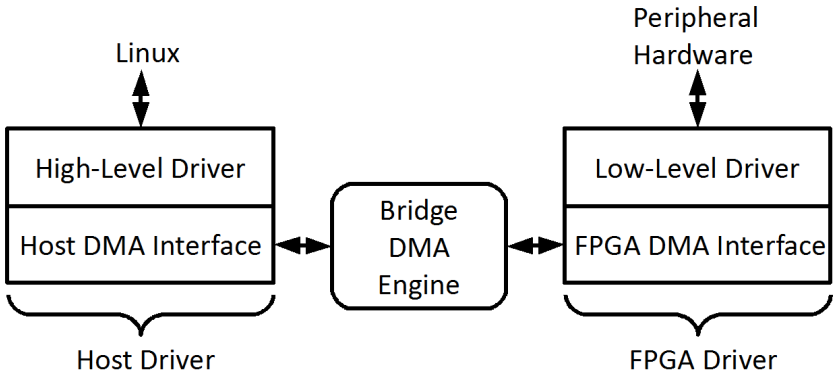

Figure 3.5: The real-time bridge software architecture consists of a host driver and an FPGA driver.

stable, open-source Linux drivers to already exist for peripherals that we are interested in controlling, which simplifies the software effort. Thus, in order to maximize code reuse, we run the Linux kernel on both the real-time bridge and the host CPU.

The real-time bridge's FPGA runs PetaLinux version 0.30rc1 [50], a Linux port for the Microblaze Soft CPU based on version 2.6.20 of the Linux kernel. The host system runs a Linux kernel, version 2.6.29.

Importantly, the driver on the main system is designed to make the real-time bridge completely transparent to the end user and end-user applications. In Figure 3.5, the Host DMA Interface portion of the host driver and the FPGA DMA Interface portion of the FPGA driver can be reused in all real-time bridges. The high-level and low-level driver portions can be extracted from an open-source Linux driver for the targeted COTS peripheral. We now elaborate on each of the drivers.

The **host driver**, which is a kernel module running on the main CPU, creates a network card interface for the real-time bridge. From the user's perspective, there is no difference between using a network card directly and using a network card through a real-time bridge.

To handle incoming traffic from the network, the host driver allocates memory for incoming packets and maintains the *available addresses hardware queue* on the bridge DMA engine. When a packet arrives, the bridge DMA engine uses these addresses to store packet data and then may raise an interrupt. To reduce overhead, the bridge DMA engine does not raise an interrupt for every packet transferred. Instead an interrupt is raised when any of these conditions becomes true:

- one or more packets have been transferred, and the bridge DMA engine has no more packets ready to be transferred,
- one or more packets have been transferred, and the `block` line becomes asserted,
- the number of transferred packets since the last interrupt reaches a design-time parameter.

Every time an interrupt is raised, the driver, without making a copy, delivers the packet addresses and lengths to the Linux network layer. The Linux network layer then processes and frees the packets. Finally, the driver refills the bridge DMA engine's *available addresses hardware queue* with new addresses to replace those that were consumed.

For outgoing traffic, the driver receives packets from the Linux network layer and puts their lengths and addresses into the *outgoing packet hardware queue* on the bridge DMA engine. If the *outgoing packet hardware queue* fills up, the driver blocks the network layer until some packets have been transferred. The network layer, in turn, stores the backlogged packet information in a software queue. After the packet data is transferred to FPGA DRAM, an interrupt is raised by the bridge DMA engine using the same rules as incoming packets, to inform the driver that the memory associated with the transferred packets can be freed.

The only dependence of the host driver on the COTS peripheral is the type of interface that is exported to Linux. This means that, if we were to design a real-time bridge for a different type of network interface card, the host driver would remain the same. To adapt the driver for a different class of peripherals, only the Linux interface would need be changed[2]. In terms of driver reuse, the Linux interface is contained in the high-level driver. Even if a Linux driver does not exist for the specific COTS peripheral we are using, all drivers for the same type of peripheral will contain the same Linux interface and therefore the same high-level driver. Thus, for developing the host driver, any existing driver for the same type of peripheral that we are using will reduce software effort.

The **FPGA Driver** runs on the PetaLinux kernel on the real-time bridge. The driver consists of an FPGA DMA Interface, and the low-level driver for the TEMAC hardware block. The driver deals only

[2]The Linux interface determines the class of the device (character device, block device, or network interface), and the relevant device file operations.

with lengths and addresses of packets, and does not make copies or perform any processing in order to minimize the performance impact of the slow 125MHz Microblaze Processor.

Incoming and outgoing packets work as expected, with a few intricacies. When incoming packets become queued, perhaps because the `block` signal is asserted to the real-time bridge, the *incoming packet hardware queue* may become full. In this case, the driver maintains a software queue to store incoming packets until space becomes available in the *incoming packet hardware queue*. Therefore, packet drops and retransmissions are avoided resulting in better performance of network protocols like TCP. Additionally, the FPGA driver shares information with the Host Driver in order to propagate network parameters such as the MAC address and the Maximum Transmission Unit (MTU).

Even though the FPGA driver has direct interaction with the COTS hardware, most of the code can be reused from either the already-developed FPGA DMA interface, or the low-level driver portion of an existing Linux driver. For example, in our network card real-time bridge prototype, we use the existing TEMAC driver until the packets are ready to be given to the Linux network stack and then instead instruct the bridge DMA engine to send them to host memory. For outgoing packets, the data received from the bridge DMA engine is sent to the TEMAC hardware block using the same interface that the PetaLinux network layer uses.

In the worst case, if no driver source code is available for the COTS peripheral we want to control, the development effort required to develop the entire driver is still strictly less than the non-COTS

approach: building application specific hardware in addition to the entire driver.

3.6 Architecture's limitations

Two additional architectural attributes need to be considered when developing the proposed real-time I/O management system. First, the I/O transfer time is only valid if the device receives the achievable throughput on the front side bus (FSB). Since the CPU main memory access is not regulated by the reservation controller, it may concurrently access main memory. Second, for performance reasons, typical COTS buses do not allow individual bus transactions to be preempted and so they must run always to completion. This means that although we may activate a real-time bridge's `block` signal, the bridge will still need to complete its current transaction before relinquishing control of the bus, which may affect the schedulability. We address these concerns in order.

The first concern is that there may still be contention on main memory even if all other peripherals do not transmit on the bus. The CPU may concurrently attempt to access main memory. From a general framework perspective, we would need to pessimistically account for any increase in bus transfer time due to uncontrolled CPU interference through a technique similar to that used to analyze the reverse problem, CPU main memory delay because of peripheral interference [45]. However, there is a key difference which makes the analysis easier, in that PCI-E and PCI transactions are typically buffered within the interconnect. This means that unless the main

memory bandwidth is exceeded, the peripheral will not notice any increase in transfer time because of the CPU accessing main memory (the reverse, however, is not true because the interconnect does not typically buffer CPU main memory access). In a typical COTS architecture, the cumulative bandwidth consumed by the CPU and each individual high-speed peripheral does not exceed the bandwidth of main memory, so calculating the I/O transfer time is straightforward.

The next concern is that, since individual bus transactions must be allowed to run to completion because of the COTS bus, the reservation controller's `block` command is not acted upon immediately. In order to evaluate the impact of this delay, we computed its maximum value. A typical single lane PCI-E device has an achievable bus throughput of 250MB/sec where each bus transaction is 4KB. This results in single bus transaction time of 16 microseconds. Since this is three orders of magnitude less than our I/O period (a high resolution video stream at 50 frames per second has an I/O period of 20ms), we consider its effect negligible for the schedulers we evaluate. However, if the reservation controller uses a scheduler that switches between devices with the same granularity as the single bus transaction time (for example a least slack first scheduler), then this effect would need to be taken into account (for example by adding an `executing` output from each the real-time bridge to measure exact bus access time).

3.7 Evaluation

The goal of our evaluation is twofold. First, we demonstrate that there is a problem using COTS interconnect for a real-time system. We present an I/O task set which results in I/O deadline misses when running on a standard COTS bus. Next, we run the same I/O task set within our scheduling framework, and show that all deadlines are met. This demonstrates the non-real-time nature of COTS interconnect, and validates the correctness of our solution.

Performing direct measurements on a high performance COTS I/O system such as PCI-E is difficult: the PCI-E protocol implements point-to-point connections between each peripheral and the rest of the system running at the very high speed of 2.5 GHz, making it hard to directly observe. In order to make the most accurate measurements, we used dedicated hardware on the reservation controller. Our trace acquisition hardware module polls the state of the `data_rdy` and `block` signals with a one microsecond resolution. Any changes in these signals, along with an associated timestamp, are output over the reservation controller ML505's serial port where they can be received by an external computer for processing.

The most notable feature of our hardware setup is the Asus P5W DH Motherboard, which features four PCI-E slots (two PCIE1 and two PCIE16), an Intel 975X system controller (northbridge). For the real-time I/O system, we use a Xilinx ML505 Evaluation Platform [74] for the Reservation Controller, and connect it to three other ML505 traffic generators with PCIE1 connections (real-time bridges), and to one Xilinx ML555 PCI Express Development Board [78] with

a PCIE8 bus interface (programmed to act as PCI-E traffic generator). Figure 3.6 shows a picture of our evaluation testbed.

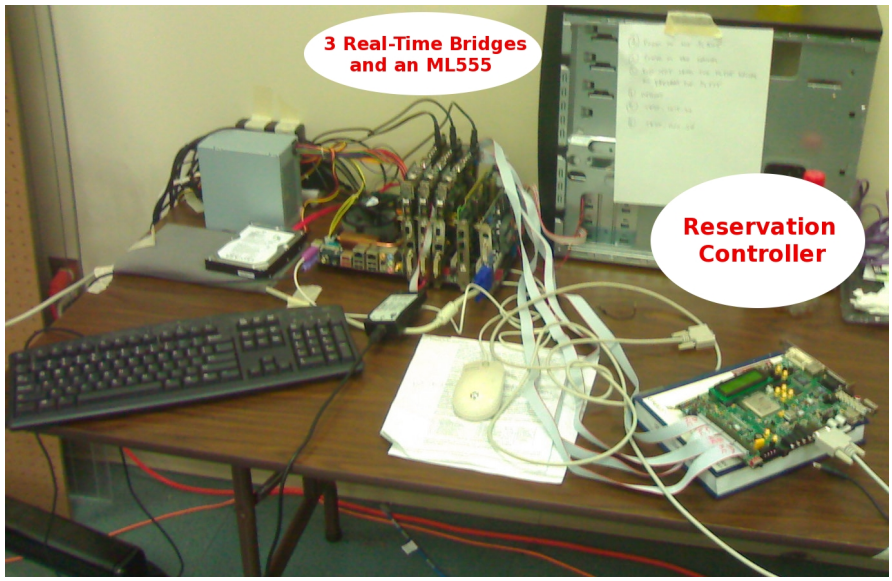

Figure 3.6: Real-time bridge evaluation testbed.

Using a PC platform permits easy access to all PCI slots, however, to derive meaningful measurements, we changed the FSB clock frequency obtaining a theoretical memory bandwidth of 2.4 Gbyte/s, which is in line with typical values for embedded platforms. At this purpose we slowed down the Front Side Bus to 400MHz, and use a single RAM chip[3].

To make our experiments more easily repeatable, we instructed the real-time bridge prototype to generate synthetic traffic instead of using traffic received by the TEMAC over the network. Our periodic task generating drivers run on the main CPU, and since our

[3]In this way, *Double-Data-Rate* signaling is disabled [68].

Board	Data Size	Transfer Time	Budget	Period
ML555	4.0 MB	4.4 ms	5 ms	8 ms
ML505	1.1 MB	7.5 ms	9 ms	72 ms
ML505	1.1 MB	7.5 ms	9 ms	72 ms
ML505	1.1 MB	7.5 ms	9 ms	72 ms

Table 3.1: Our experiments using four flows.

I/O schedule uses periods on the order of milliseconds, it is difficult to exactly synchronize all synthetic tasks. For this reason, we ran the tests for many hyperperiods, and show here the traces from the most closely aligned arrival times, which correspond to the *near-critical instants*. The arrival times of the presented traces are never separated by more than 0.8 milliseconds. As we said, additionally we implemented a traffic generator using an ML555 PCI Express Development Board [78] with a faster 8 lane PCI-E connection. The synthetic traffic generator is programmed to send a constant amount of data to main memory every period and obeys the I/O scheduling commands from the reservation controller.

The task set used in our experiments consists of four real-time flows competing for main memory. The task parameters (data size, transfer time, period) are shown in Table 3.1. The tasks' periods are harmonic, and the total utilization does not exceed 100%, so the task set is schedulable under RM.

In the first experiment, the COTS bus is used without the reservation controller. Traffic gets sent on the bus as soon as it arrives, increasing the execution time of the ML555's periodic task from 4.4 ms to over 8 ms (an increase of 82%) when the tasks start at a near-

critical instant. This causes a deadline miss (see Figure 3.7).

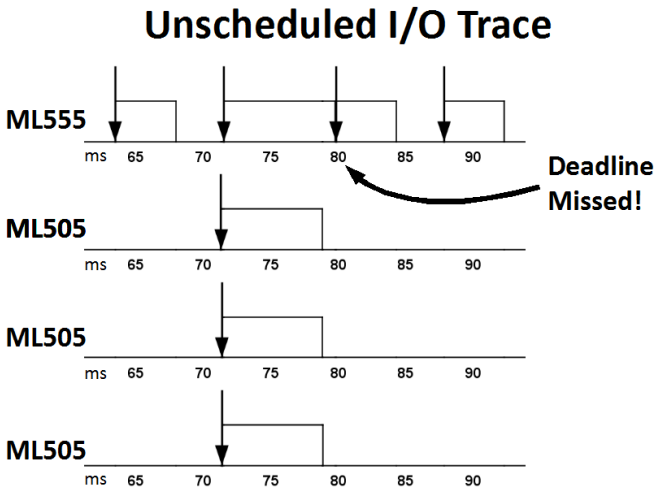

Figure 3.7: The trace of a standard COTS I/O system reveals a deadline miss if the tasks are released at a near-critical instant.

In the second experiment, each peripheral is handled by a sporadic server (whose corresponding budget and period are shown in Table 3.1) and all the servers are scheduled according to Rate Monotonic with total utilization $\frac{5}{8} + \frac{9}{72} + \frac{9}{72} + \frac{9}{72} = 1$. By using the proposed real-time I/O management system, the task set is now successfully scheduled without missing deadlines because the traffic is prioritized. A trace of one hyperperiod starting at a near-critical instant is shown in Figure 3.8.

3.7.1 Network performance and overhead test

An important detail of this implementation is that the (comparatively) slow Microblaze processor does minimal processing (working

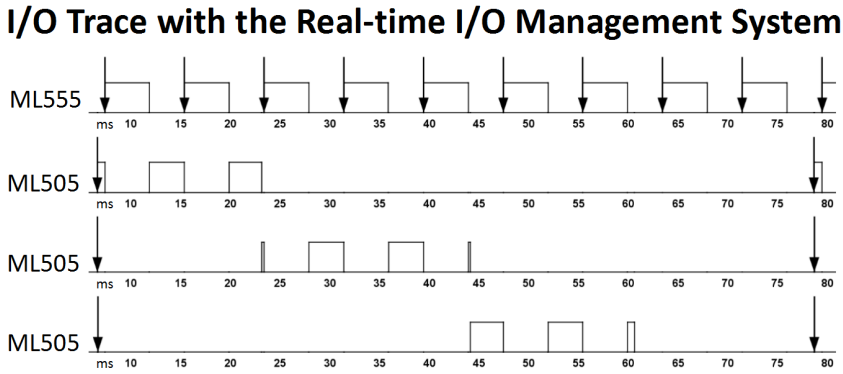

Figure 3.8: The trace of the task set running with the real-time I/O management shows the system preventing deadline misses by prioritizing traffic.

with only the addresses and lengths) and no copying of the potentially high-bandwidth packet data. This allows our prototype implementation to achieve a network throughput of about 100 Mbps for upload and 80 Mbps for download, which coincides exactly with Xilinx's TEMAC performance benchmarks for our setup (ML505, 125MHz Microblaze, 1500 Maximum Transmission Unit (MTU)) [27]. Performance can be further improved by using a larger MTU. Additionally, without the Microblaze bottleneck, the bridge DMA engine was able to send data at 207 MBps, which approaches the theoretical limit of a PCI-E single lane connection (250 MBps).

It is worth noticing that the proposed real-time I/O management system introduces additional latency compared with a COTS peripheral communicating directly to the PCI-E bus. This is one tradeoff that is made in order to provide control of peripheral bus access.

We ran an experiment to get an idea of the effect of this addi-

tional latency by sending ping packets to the main CPU through our real-time bridge I/O management system (with the real-time bridge scheduled as the highest-priority sporadic server) and sending ping packets directly to the FGPA's PetaLinux OS. Surprisingly, the packet round-trip times through our bus scheduling prototype (2.40ms) were actually *lower* than the round-trip times for the packets processed immediately on the FPGA (2.62ms). Hence, it is faster to place the packet data in the bridge DMA engine, assert `data_rdy` to the reservation controller, wait for the `block` signal to be deasserted, receive access to the PCI-E bus, transmit the data into Host DRAM, process the ping on the host's 2.66 GHz CPU, and reverse the entire process for the ping response, than to handle the ping packet directly on the slower 125 MHz Microblaze processor. This experiment demonstrates the efficiency of our implementation.

3.8 Related work

In earlier work, Pellizzoni *et al.* proposed a coscheduling framework between CPU and I/O peripherals to guarantee main memory latency for tasks running on the CPU [45, 44]. The authors proposed to use *passive Peripheral gate (P-gate)* devices to block and unblock peripherals (possibly reducing I/O throughput and causing internal peripheral buffers to overflow) and to synchronize I/O activity with tasks executing on the CPU.

Although this technique was effective for predicting each task's worst-case execution time (WCET) in spite of I/O traffic spikes, it did not guarantee the timeliness of I/O traffic. To the best of our

knowledge, our real-time I/O management system provides significant advancement over the current state of the art. In fact, while it remains compatible with the cited timing analysis to guarantee main memory latency for tasks running on the CPU, the real-time I/O management system provides the following novel features:

1. it enforces predictable bandwidth reservations for I/O COTS peripherals,

2. it does not require synchronization between the CPU scheduler and the I/O subsystem

3. it facilitates lossless reshaping (under given assumptions) of bursty traffic from a network of distributed real-time nodes.

Finally, it is worth noticing that the proposed real-time I/O management system is completely transparent to main CPU applications: in fact, we were able to use a network card through our real-time I/O management system prototype without any user-end application modifications.

Apart from the work done by Pellizoni *et al.* [45, 44], several papers address the problem of interference at the main memory level. Empirical approaches can estimate the impact of PCI-bus load on task computation time based on experimental measurements of reference tasks [58]. Alternately, analytical approaches exist to bound I/O interference [32]. However, the analysis is restricted to a single DMA controller using predictable cycle-stealing arbitration, and can not be applied to a COTS system. There is also analysis to estimate the impact of mutual interference among processing cores. For

example, static analysis can compute cache access delay in a multiprocessor system [55]. However, these results focus on deriving the increase in task execution time while neglecting the effect of delay on communication flows.

Modeling complex COTS interconnections and estimating delay and buffer requirements for peripheral flows can be done in an AADL-based environment [41]. An event-based model may be used to estimate delay for both computation and communication activities in a multicore system-on-chip [56]. However, lack of precise knowledge of COTS behavior implies that these analyses must make pessimistic assumptions, which can lead to high delay and buffer sizes. Our real-time I/O management system removes such unpredictability by forcing an implicit bus schedule.

Chapter 4

Real-time management of shared COTS peripherals

The hardware framework described in chapter 3 supports a single real-time flow through each real-time bridge. Therefore, if the controlled peripheral is shared among two or more tasks with different real-time priorities it is not possible to distinguish the different data flows and, as a consequence, schedule them properly.

In this chapter we describe a similar architecture based on a **multi-flow real-time bridge** [5], which is an improved real-time bridge capable to export to the host system several virtual devices mapped on the same physical peripheral. With this new feature, multiple real-time flows can be scheduled within the same real-time bridge, giving the whole architecture the following main improvements:

- possibility of sharing a peripheral also among tasks with different priorities,
- increased peripheral utilization,
- increased system scalability (reducing the number of peripherals and bridges needed).

In the following sections, we first give an overview of the new features and benefits of the multi-flow real-time bridge (section 4.1). Then we describe design and implementation details for our prototype (section 4.2). Finally, in section 4.3, the new experiments' results are discussed.

4.1 Multi-flow real-time bridge overview

Using the real-time bridge, described in chapter 3, it is possible to enforce a real-time scheduling for I/O transactions on COTS buses like PCI-E. Each real-time bridge can control one peripheral. Moreover, all the data traffic that goes through the bridge is considered as a single real-time flow and scheduled with the same priority.

With such architecture we obtained excellent results, avoiding interferences among different peripherals (see section 3.7). However, we could not address interference within I/O peripherals. As an evolution of the real-time bridge, we designed and implemented a new component, called **multi-flow real-time bridge**, that allows multiple real-time flows through a single bridge. This new bridge has been defined *multi-flow* because it is able to distinguish different data flows, which vary in real-time importance, and send to the *reservation controller* (see section 3.4) the needed information to schedule each flow according to its own priority.

The main contribute of this new feature is the possibility of sharing a peripheral among tasks with different priorities, being able to enforce a real-time scheduling for the different flows. In a single-flow real-time bridge, if two or more tasks are accessing the same periph-

eral (which means also the same bridge), they may delay each other transaction, since the policy to schedule transactions within a real-time bridge is just FIFO (First-In First-Out). When the competing tasks have the same priority this might be acceptable, but it needs to be managed with a proper design of the real-time system. Instead, when the tasks have different criticality, a lower priority task might delay a higher priority one, and potentially make the critical task misses its deadline. A multi-flow real-time bridge is capable to avoid this kind of interferences within an I/O peripheral, scheduling one flow at the time, according to a predeterminated real-time scheduling policy. Hence, peripheral sharing is now safe, and it is not needed anymore to have a peripheral and a bridge for each priority level. In section 4.3, we show some experimental results on our prototype that demonstrate the interference within the peripheral and how the multi-flow real-time bridge can solve this problem.

Being able of sharing a device has some direct, positive consequences: the devices' utilization and the system scalability are increased, while the cost of the system is reduced. For example, we could imagine a hard real-time system where there are four tasks with different priorities:

- `task1` has the highest priority and requires 10% of the device bandwidth;
- `task2` has a lower priority than `task1` and requires 50% of the device bandwidth;
- `task3` has a lower priority than `task2` and requires 70% of the device bandwidth.

- `task4` has a lower priority than `task3` and requires 40% of the device bandwidth.

Using a single-flow real-time bridges we would need four devices and four bridges in order to assure a proper real-time scheduling. With multi-flow real-time bridges we could use only two devices and two bridges, having `task1` and `task3` sharing the same bridge, and `task2` and `task4` using the other one. As a consequence, the device utilization is definitely higher (two device with 80% and 90%, instead of four with 10%, 50%, 70%, and 40%). Moreover, the whole system will be less expensive and more scalable (since there will be space for more devices, hence more tasks).

As usual, a trade-off must be paid, and, compared to a single-flow real-time bridge, a multi-flow real-time bridge requires:

- more memory for buffers,
- more complex drivers,
- a hardware or software logic to distinguish different flows,
- in case of a software logic a faster CPU is needed.

In the next section, discussing our prototype implementation, we will see how these new requirements can be easily satisfied. For example, for a network card, the amount of memory used for each flow is less than one Megabyte. Increasing the number of flows, the amount of memory needed increases linearly; i.e. for 8 flows we need about 8 Megabytes, which is still reasonable. The new prototype runs a software logic to distinguish the different flows using an IBM

Power PC 400MHz, which is a quite common processor for embedded systems, and more than enough to accomplish the multi-flow real-time bridge requirements. Finally, the drivers are more complex, but just in that parts which are independent from the controlled device, and have been already developed for our prototype. As a consequence, changing the controlled device would require just small changes, like we already discussed in section 3.5. Moreover, increasing the number of supported flows is a completly dynamic process, so it does not require any change in the software.

4.2 Multi-flow real-time bridge design and implementation

As we said for the single-flow real-time bridge in section 3.5, we envision a final implementation using a setup similar to the Xilinx ML455 [76] (see figure 3.3 in page 81), where a COTS peripheral can be connected to a real-time bridge, which is directly connected to a COTS bus.

However, also in this case, in order to rapidly develop a complete prototype, we decided to use a Xilinx ML507 Evaluation Platform [77], which features an XC5VFX70T FPGA, and a 400 MHz IBM Power PC processor[1]. We also targeted the same network card interface, a Tri-state Ethernet MAC (TEMAC), which will act as the controlled COTS peripheral.

[1]For the single-flow real-time bridge we used a Xilinx ML505 [74] with a Microblaze soft CPU [75].

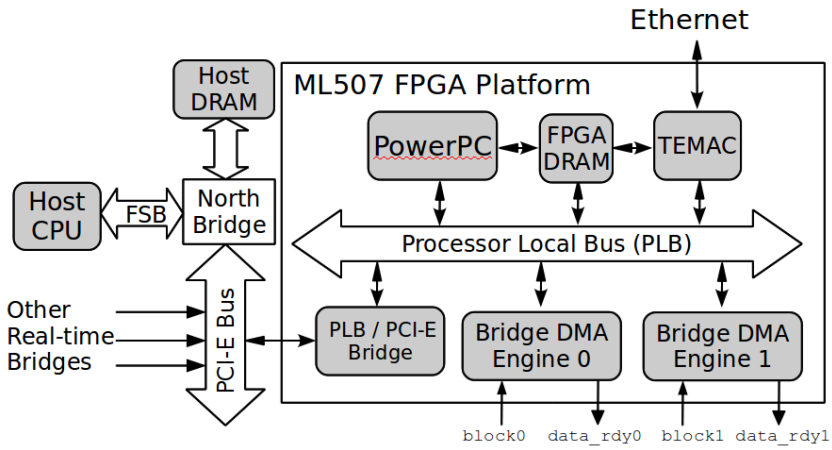

Figure 4.1: Our multi-flow real-time bridge prototype is a System-on-Chip implemented on the ML507 FPGA Evaluation Platform. In this case only two real-time flows are supported, but more bridge DMA engines could be easily added.

To allow multiple real-time flows through a single real-time bridge we needed to replicate the bridge DMA engine for each flow we would like to support. Figure 4.1 shows an overview of the hardware components of a 2-flow real-time bridge prototype; it is possible to notice that, in order to support two real-time flows, a second bridge DMA engine has been added. Each bridge DMA engine has its own `data_rdy` and `block` pin, thus each flow can be separately scheduled. Moreover, since the scheduling wires interact only with the bridge DMA engine, the reservation controller does not need to be modified. In fact, in this design there is no difference between scheduling bridge DMA engines resident in the same bridge or in different ones.

It is also worth noticing that an IBM Power PC processor is now

replacing the Microblaze Soft CPU used in the single-flow real-time bridge. The Power PC runs at 400 MHz, which is much faster than the Microblaze, running at only 125 MHz; this allowed us to implement a complete software logic to distinguish different flows, without introducing any significant delay. Anyway, as a direct consequence, the original real-time bridge drivers needed to be widely modified to support *device virtualization*, and *incoming data routing*: these concepts will be explained in detail in the following section.

4.2.1 Multi-flow real-time bridge software details

In the hardware design the main difference between a single-flow and a multi-flow real-time bridge is that the second one has multiple bridge DMA engines available. This allows us to schedule different flows within the same peripheral according to the priority assigned to the bridge DMA engine. However a logic to split the incoming and out-coming data is still needed. Every time a block of data needs to be transferred from main memory to the device (write operation) we need to commit this transfer to one of the available bridge DMA engines, according with the priority of the task that initiates the transaction. Equally, when a transfer is required from the the device to main memory (read operation) the right bridge DMA engine must be selected.

To easily distinguish these logic flows we developed a driver for the host system that supports *device virtualization*. In other words, the multi-flow real-time bridge exposes a separate memory I/O block for each bridge DMA engine and the host driver, instead of creating a

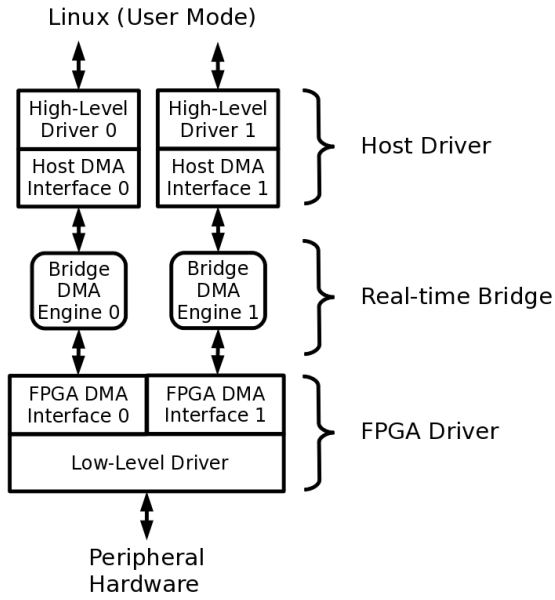

Figure 4.2: Software architecture overview of a 2-flow real-time bridge.

single device interface, creates as many *virtual devices* as the amount of bridge DMA engines probed. From the user point of view this approach is completely transparent: multiple device interfaces are exported in User Mode and each of them can be accessed like it is the only one using the physical device.

Figure 4.2 shows an overview of the software architecture when two bridge DMA engines are present. Comparing this scheme with the one in figure 3.5 (page 85) we can notice how the low level driver, running in the FPGA, is still implemented like a common layer. Conversely, all the other components run a different instance for each bridge DMA engine. In particular, each instance of the high level driver creates a distinct software interface for the device: for exam-

ple, it could be a character device or a network interface, depending on the type of controlled peripheral.

The device virtualization approach in now used also in commercial devices, like the Intel 82598 10 Gigabit Ethernet Controller [18]. In this case, a hardware virtualization is realized to optimize the device sharing among several virtual machines: the main purpose is offloading data sorting and data copying from the virtual machine monitor (VMM) software layer to the hardware. In term of average performance, our multi-flow real-time bridge prototype is slower compared with the Intel 82598, but this is just a limitation of the actual implementation. Instead, from the design point of view, we are able to offer a similar device virtualization feature, but also a global real-time I/O bus scheduling within the peripheral and among different peripherals, which is completely innovative and very important in hard real-time systems.

For a better understanding of the multi-flow real-time bridge design, let the two cases, read and write operations, be treated separately and explained in the context of our implementation: a multi-flow real-time bridge controlling a Gigabit Ethernet network card. Having in mind a specific device will make the general design just easier to understand, but all the basic concepts could still be applied to other kinds of device. For a network card, we can think of a write operation (memory to device) like an out-coming packet (upload), and of a read operation (device to memory) like an incoming one (download): this will be the terminology used for the rest of this section.

Upload

One of the goals we want to achieve with the multi-flow real-time bridges is that the priority assigned to each task is reflected also when the task is using a peripheral and, as a consequence, accessing the bus. In order to do this, in our design each bridge DMA engine has a different priority (assigned by the reservation controller), and for each one of them a virtual network device (`eth0`, `eth1`, etc...) is exported in User Mode. Each network device can be configured with its own IP address. Hence, tasks can be bound with the right virtual network device, according with their priorities. For example, the highest priority task could use `eth0` (let it be the highest priority bridge DMA engine) and lower priority tasks could be bound to other virtual devices. Writing data to a certain virtual device will activate only the corresponding bridge DMA engine, so that the packet will be consequently prioritized.

It is evident how this approach is completely transparent to the application, which does not need any modification. All is needed is to configure the real-time application to use a specific (virtual) device, which is needed in any case; for example, with a network card this means only assigning the right IP address to the virtual device and having the task communicating in the related network.

Download

Every time a packet arrives it is delivered in the FPGA main memory, and the FPGA driver is activated by an interrupt. At this point, in order to forward the packet to the host system, a bridge DMA engine

needs to be selected, according to the priority of the packet itself. Since we are scheduling data flows, the packet priority is directly related to the destination task, which needs to be guessed reading the packet.

A hook for a *filter function* is provided by the FPGA driver and, knowing the network protocols used, a special custom function could be easily written. Anyway, for a common application we can imagine that the IP protocol is used, and the proper filter function has been implemented. In this case, assuming that each task is using a virtual device with its own IP address, reading the *destination IP* field in the IP header is enough to select the right bridge DMA engine. It follows that the packet will get access to the bus according to its priority.

A special policy needs to be applied for broadcast packets. In our implementation the default filter function can be configured to send all the broadcast traffic to a certain bridge DMA engine, or to equally distribute them among all the available bridge DMA engine. Also in this case a custom filter function can be written. However we assume that in a hard real-time system, the traffic pattern and the used protocols are well know, and by reading the packet it should be easy to identify the final destination.

Hardware implementation

As we will see in the section 4.3, this software implementation does not introduce any significant delay. However, as a future work or for an ASIC implementation, the hardware support for device virtualization could be improved, for example using multiple PCI functions

and MSI-X vectors. Moreover, the filter function for incoming traffic could be implemented as a hardware block.

4.3 Evaluation

To show the advantages of our I/O management system, and evaluate the implementation of our prototype, we performed three kinds of experiments, measuring:

1. interference among I/O peripherals,
2. interference within I/O peripherals,
3. network performance of our prototype.

For the first experiment, we obtained the same results already discussed in section 3.7 (page 91) and performed with real-time bridge. This is reasonable since, dealing with interference among different peripherals, nothing changed between real-time bridges and the new multi-flow real-time bridge implementation.

The other two experiments are discussed in the following sections.

4.3.1 Interference within I/O peripherals

The goal of the second experiment is to demonstrate that, although giving each device guaranteed main memory access is a significant improvement to a COTS system (as shown in the first experiment), more control may still be necessary. Particularly, if real-time tasks with different priorities are using the same peripheral, it is desirable

Source	Budget	Period
bridge DMA engine #0	4 ms	6 ms
ML555	4 ms	21 ms
bridge DMA engine #1	5 ms	42 ms

Table 4.1: Our second experiment has three flows on two peripherals (both bridge DMA engines are on the same peripheral).

to create prioritized virtual channels through the peripheral, where traffic destined for the high priority task is given its own reservation and is scheduled differently than traffic destined for the lower priority task.

The task set used in our experiments consists of three real-time flows on two devices. The multi-flow real-time bridge device contains two bridge DMA engines with traffic of different priorities. The task parameters (budget, period) are shown in Table 4.1. The transfer time is slightly less than the allocated sporadic server budget. The tasks' periods are harmonic, and the total utilization does not exceed 100%, so the task set is schedulable ($\frac{4}{6} + \frac{4}{21} + \frac{5}{42} \approx 0.98 < 1$).

In the first run, the devices transmit data without the reservation controller. Since both bridge DMA engine are co-located on the same multi-flow real-time bridge, their traffic is interleaved and the transfer time of the task corresponding to bridge DMA engine #0 increases from less than 4 ms to over 6 ms, missing its I/O deadline (Figure 4.3). The bottleneck here is not main memory bandwidth, but rather peripheral bus bandwidth. Such a case may arise if, for example, two real-time tasks of different priorities attempt to send data at the same

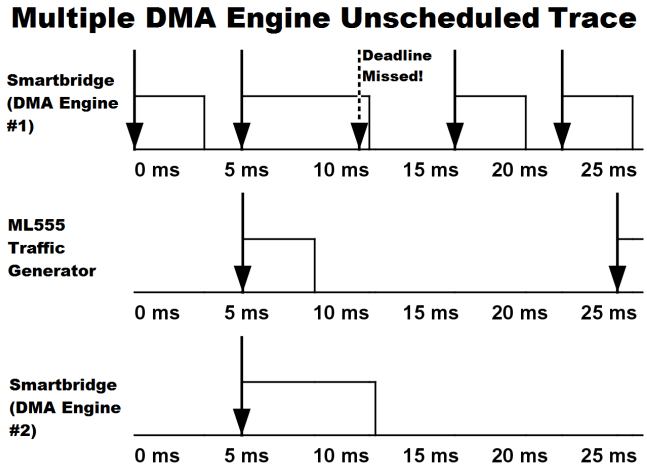

Figure 4.3: The bus-access trace of a system with two DMA engines on the same multi-flow real-time bridge reveals the low priority flow interfering with the high priority flow, causing a deadline miss.

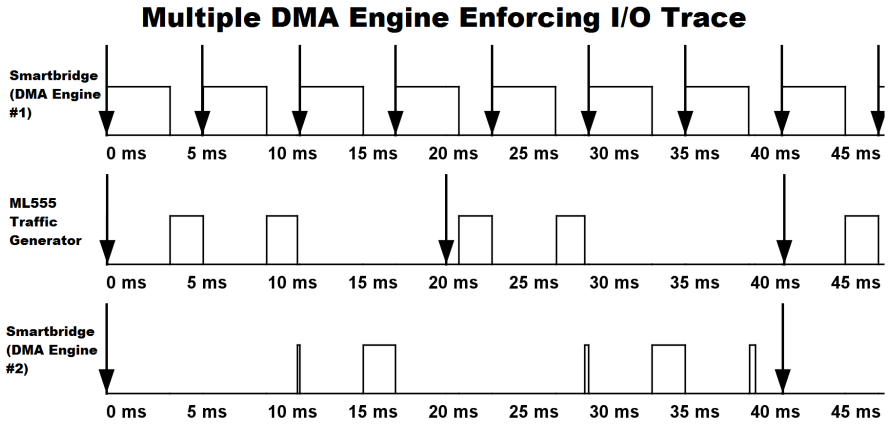

Figure 4.4: The bus-access trace of the same task set shows the real-time I/O management system prioritizing bus access among real-time flows, preventing a deadline miss.

time out over the same network card.

In the second run, each virtual flow is schedule by a sporadic server according to the parameters in Table 4.1. By using the proposed real-time I/O management system, the task set is now successfully scheduled without missing deadlines because the traffic is prioritized. An execution trace is shown in Figure 4.4. Notice that because there is a medium-priority real-time flow from the ML555, the lower priority traffic from bridge DMA engine #1 is not transferred immediately after bridge DMA engine #0's traffic completes. Such system-aware I/O scheduling is not possible on any hardware virtualized network card (since they are aware only of flows going through the one device), which reveals another advantage of using our I/O management system.

	TCP RX	UDP RX	TCP TX	UDP TX
MontaVista Linux	209	307	**393**	**492**
Real-time bridge	**543**	**750**	331	345

Table 4.2: TEMAC bandwidth in Mbit/sec mesuared with NetPerf. MTU was set to 1500 bytes.

4.3.2 Network performance

In order to measure the network bandwidth that our prototype can achieve, we used NetPerf [33], a standard benchmark suite. We also compared our results with the one discussed in the Xilinx Application Note 1127 [30]. It is worth to notice that, in Xilinx's test, MontaVista Linux 4.0 is installed on the FPGA and NetPerf runs directly on the Power PC. On the contrary, in our test NetPerf runs on the host system and the Linux installation on the real-time bridge runs only the FPGA driver, as discussed in section 4.2.1. In this way all the the operations needed to handle the TCP/IP protocol stack are executed on the main system, which is much faster than the FPGA. On the other hand, running NetPerf on the main system, we need to transfer the packets between main memory and the FPGA DRAM and, for incoming traffic, we also run on the FPGA a software filter to select the right bridge DMA engine for each packet.

Table 4.2 shows compared results between MontaVista Linux and our multi-flow real-time bridge. We can see how the real-time bridge is much faster in download (145%–160% increment), but it is slower in upload (16%–30% loss). The speed up in download is mainly due

to the fact that we offload TCP/IP computation from the FPGA to the main system. We instead measured a loss in upload bandwidth due to the implementation of the PCI-E bridge that we are using on the real-time bridge, where read operations performe much slower than write ones. Anyway, this is just a limitation of the current implementation and not of the general design.

Another interesting result is that, performing tests using 1 or 2 bridge DMA engines we measured the same bandwidth. With just one bridge DMA engine the filter function for incoming traffic and the device virtualization are both disabled. It follows that, enabling them does not introduce any evident overhead, even though they are completely software implemented.

Chapter 5

PREM: PRedictable Execution Model

With ASMP-Linux and multi-flow real-time bridges we are able to mitigate, and in some cases to remove, software jitter and internal- and inter-peripheral interference. However, other two important sources of indeterminism have not been addressed yet:

- in ASMP-Linux, if the real-time task running on a shielded processor needs to use a peripheral, the interference between the driver that controls the peripheral and the task cannot be managed;
- using only our novel I/O management system, based on multi-flow real-time bridges and a reservation controller, we are not able to prevent interferences between a peripheral and the CPU, when they access memory at the same time.

Solving also these two issues would open the possibility to build a fully predictable system, running on COTS components.

In this chapter, we introduce a novel system execution model, the PRedictable Execution Model (PREM[48]), which, in contrast to the standard COTS execution model, coschedules at a high level all

active COTS components in the system, such as CPU cores and I/O peripherals, and the interrupts of critical device drivers.

The chapter is organized as follows. Section 5.1 introduce the general concepts of PREM. Section 5.2 discusses related work. In Section 5.3 we describe PREM's main contribution: a co-scheduling mechanism that schedules I/O interrupt handlers, task memory accesses and I/O peripheral data transfers in such a way that access to shared COTS resources is serialized achieving zero or negligible contention during memory accesses. Then, in sections 5.4 and 5.5 we discuss the challenges in term of hardware architecture and code organization that must be met to predictably compile real-time tasks. Finally, in section 5.6 we detail our prototype testbed, including our compiler implementation based on the LLVM Compiler Infrastructure [34], and provide an experimental evaluation.

5.1 Introduction

To exploit the high average performance of COTS components without experiencing the long delays occasionally suffered by real-time tasks, we need to control the operating point of each COTS shared resource and maintain it below saturation limits. This is necessary because the low-level arbiters of the shared resources are not typically designed to provide real-time guarantees. We believe that this is indeed possible by carefully rethinking the execution model of real-time tasks and by enforcing a high-level coscheduling mechanism among all active COTS components and the drivers in the system. Briefly, the key idea is to coschedule active components so that contention

for accessing COTS shared resources (caches, memories, buses) is implicitly resolved by the high-level coscheduler without relying on low-level, non-real-time arbiters. Several challenges had to be overcome to realize the PRedictable Execution Model:

- Task execution times suffer high variance due to internal CPU architecture features (caches, pipelines, etc.) and unknown cache miss patterns. This source of temporal unpredictability forces the designer to make very pessimistic assumptions when performing schedulability analysis. To address this problem, PREM uses a novel program execution model with three main features:

  1. jobs are divided into a sequence of non-preemptive scheduling intervals;
  2. some of these scheduling intervals (named **predictable intervals**) are executed *predictably* and *without cache-misses* by prefetching all required data at the beginning of the interval itself;
  3. the execution time of **predictable intervals** is kept constant by monitoring CPU time counters at run-time.

- as we already discussed, I/O peripherals with DMA master capabilities contend for physically shared resources, including memory and buses, in an unpredictable manner. To address this problem, we use real-time bridges, described in chapters 3 and 4, and introduce a new *peripheral scheduler*, controlled by

the main CPU, that puts the COTS I/O subsystem under the discipline of real-time scheduling.

- combining our I/O management system with a new compiler pass, we enforce a coscheduling mechanism that serializes arbitration requests of active components (CPU cores, I/O peripherals, and interrupts). During the execution of a task's predictable interval, a scheduled peripheral can access the bus and memory without experiencing delays due to cache misses caused by the task's execution.

Our PRedictable Execution Model can be used with a high level programming language like C by setting some programming guidelines and by using a modified compiler to generate predictable executables. The programmer provides some information, like beginning and end of each predictable execution interval, and the compiler generates programs which perform cache prefetching and enforce a constant execution time in each predictable interval. In light of the above discussion, we argue that real-time embedded applications should be compiled according to a new set of rules dictated by PREM. At the price of minor additional work by the programmer, the generated executable becomes far more predictable than state-of-the-art compiled code, and when run with the rest of the PREM system, shows significantly reduced worst-case execution time.

5.2 Related work

Several solutions have been proposed in prior real-time research to address different sources of unpredictability in COTS components, including real-time handling of peripheral drivers, real-time compilation, and analysis of contention for memory and buses. For peripheral drivers, Facchinetti et al. [23] proposed using a non-preemptive interrupt server to better support the reusing of legacy drivers. Additionally, analysis can be done to model worst-case temporal interference caused by device drivers [35]. For real-time compilation, a tight coupling between compiler and worst-case execution time (WCET) analyzer can optimize a program's WCET [24]. Alternatively, a compiler-based approach can provide predictable paging [53]. For analysis of contention for memory and buses, existing techniques can analyze the maximum delay caused by contention for a shared memory or bus under various access models [46, 56]. All these works attempt to analyze or control a single resource, and obtain safe bounds that are often highly pessimistic. Instead, PREM is based on a global coschedule of all relevant system resources.

Instead of using COTS components, other researchers have discussed new architectural solutions that can greatly increase system predictability by removing significant sources of interference. Instead of a standard cache-based architecture, a real-time scratchpad architecture can be used to provide predictable access time to main memory [73]. The Precision Time (PRET) machine [21] promises to simultaneously deliver high computational performance together with cycle-accurate estimation of program execution time. While our

PRedictable Execution Model borrows some ideas from these works, it exhibits one key difference: our model can be applied to existing COTS-based systems, without requiring significant architectural redesign. This approach allows PREM to leverage the advantage of the economy of scale of COTS systems, and support the progressive migration of legacy systems.

5.3 System model

We consider a typical COTS-based real-time embedded system comprising of a CPU, main memory and multiple DMA peripherals. While in this first design of PREM we restrict our discussion to single-core systems with no hardware multithreading, we believe that our predictable execution model is also applicable to multicore systems.[1]

The CPU can implement one or more cache levels. We focus on the last cache level, which typically employs a write-back policy. Whenever a task suffers a cache miss in the last level, the cache controller must access main memory to fetch the newly referenced cache line and possibly write-back a replaced cache line. Peripherals are connected to the system through COTS interconnect such as PCI or PCI-E [42].

DMA peripherals can autonomously initiate data transfers on the interconnect. We assume that all data transfers target main memory, that is, data is always transferred between the peripheral's internal buffers and main memory. Therefore, we can treat main memory as a

[1] We will present a predictable execution model for multicore systems as part of our planned future work.

single resource shared by all peripherals and by the cache controller.

The CPU executes a set $\Gamma = \{\tau_1, \ldots, \tau_N\}$ of N real-time periodic tasks. Each task can use one or more peripherals to transfer input or output data to or from main memory. We model all peripheral activities as a set of M periodic I/O flows $\Gamma^{\mathrm{I/O}} = \{\tau_1^{\mathrm{I/O}}, \ldots, \tau_M^{\mathrm{I/O}}\}$ with assigned timing reservations, and we want to schedule them in such a way that only one flow is transferred at a time. At this purpose, we use the real-time bridges described in chapters 3 and 4. However, the *reservation controller* discussed in 3.4 has been replaced with a slightly different device, called *peripheral scheduler*.

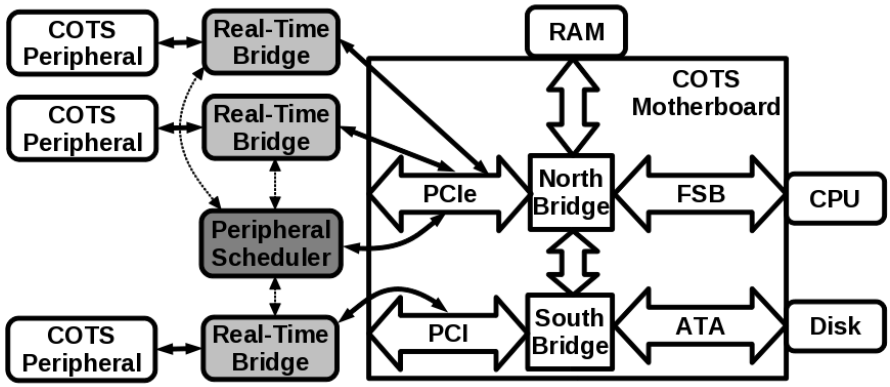

Figure 5.1: Real-Time I/O Management System.

As we can see in figure 5.1, the peripheral scheduler is directly connected to the system through the PCI-E bus. This device, receiving information from the tasks compiled according to PREM's rules, will control the real-time bridges, scheduling the I/O peripherals traffic and the interrupts, in order to avoid any kind of interference in the bus. For simplicity, in the rest of this chapter we assume that each

peripheral services a single task. However, this assumption can be easily lifted by supporting peripheral virtualization with multi-flow real-time bridges.

The peripheral scheduler, in order to allow an I/O transaction on the bus, must be sure that the CPU will not access main memory within a certain interval of time. Unfortunately, when a typical real-time task is executed on a COTS CPU, cache misses are unpredictable, making it difficult to avoid low-level contention for access to main memory. To overcome this issue, we propose a set of compiler and OS techniques that enable us to predictably schedule all cache misses during a given portion of a task execution. The code for each task τ_i is divided into a set of N_i scheduling intervals $\{s_{i,1}, \ldots, s_{i,N_i}\}$, which are executed sequentially at run-time. The timing requirements of τ_i can be expressed by a tuple $\{\{e_{i,1}, \ldots, e_{i,N_i}\}, p_i, D_i\}$, where p_i, D_i are the period and relative deadline of the task, with $D_i \leq p_i$, and $e_{i,j}$ is the maximum execution time of $s_{i,j}$, assuming that the interval runs in isolation with no memory interference. A job can only be preempted by a higher priority job at the end of a scheduling interval. This ensures that the cache content can not be altered by the preempting job during the execution of an interval. We classify the scheduling intervals into *compatible intervals* and *predictable intervals*.

Compatible intervals are compiled and executed without any special provisions (they are backwards compatible). Cache misses can happen at any time during these intervals. The task code is allowed to perform OS system calls, but blocking calls must have bounded blocking time. Furthermore, the task can be preempted by interrupt

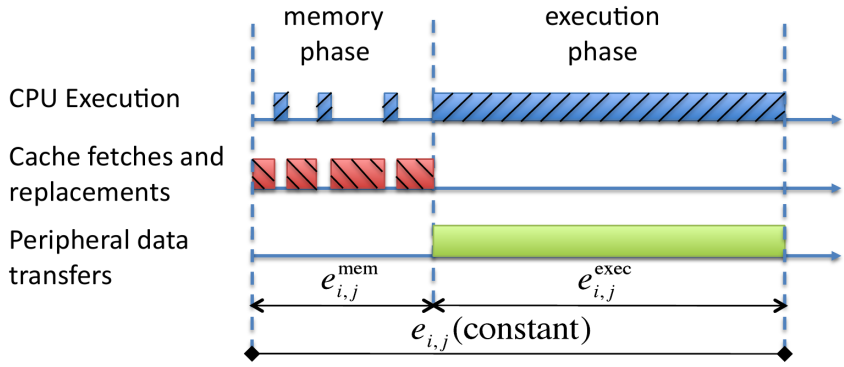

Figure 5.2: Predictable Interval with constant execution time.

handlers of associated peripherals. We assume that the maximum execution time $e_{i,j}$ for a compatible interval can be computed based on traditional static analysis techniques. However, to reduce the pessimism in the analysis, we prohibit peripheral traffic from being transmitted during a compatible interval. Ideally, there should be a small number of compatible intervals which are kept as short as possible.

Predictable intervals are specially compiled to execute according to the PREM model shown in figure 5.2, and exhibit three main properties.

First, each predictable interval is divided into two different phases. During the initial *memory phase*, the CPU accesses main memory to perform a set of cache line fetches and replacements. At the end of the memory phase, all cache lines required during the predictable interval are available in last level cache.

Second, the second phase is know as the *execution phase*. During

this phase, the task performs useful computation without suffering any last level cache misses. Predictable intervals do not contain any system calls and can not be preempted by interrupt handlers. Hence, the CPU does not perform any external main memory access during the execution phase. Due to this property, peripheral traffic can be scheduled during the execution phase of a predictable interval without causing any contention for access to main memory.

Third, at run-time, we force the execution time of a predictable interval to be always equal to $e_{i,j}$. Let $e_{i,j}^{\text{mem}}$ be the maximum time required to complete the memory phase and $e_{i,j}^{\text{exec}}$ to complete the execution phase. Then offline we set $e_{i,j} = e_{i,j}^{\text{mem}} + e_{i,j}^{\text{exec}}$ and at run-time, even if the memory phase lasts for less than $e_{i,j}^{\text{mem}}$ time units, the overall interval still completes in exactly $e_{i,j}$. This property greatly increases task predictability without affecting CPU worst-case guarantees.

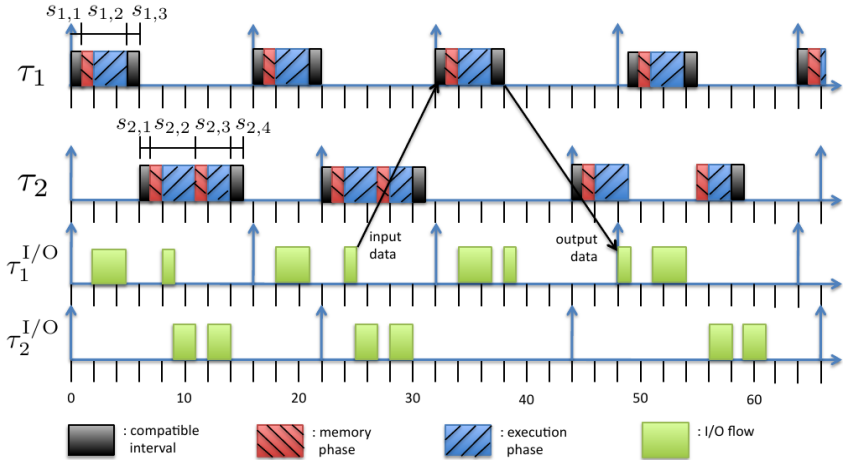

Figure 5.3: Example System-Level Predictable Schedule

Figure 5.3 shows a concrete example of a system-level predictable schedule for a task set comprising two tasks τ_1, τ_2 together with two I/O flows $\tau_1^{\mathrm{I/O}}, \tau_2^{\mathrm{I/O}}$ which service τ_1 and τ_2 respectively. Both tasks and I/O flows are scheduled according to fixed priority, with τ_1 having higher priority than τ_2 and $\tau_1^{\mathrm{I/O}}$ higher priority than $\tau_2^{\mathrm{I/O}}$. We set $D_i = p_i$ and assign to each I/O flow the same period and deadline as its serviced task and a transmission time equal to 4 time units. As shown in Figure 5.3 for task τ_1, this means that the input data for a given job is transmitted in the period before the job is executed, and the output data is transmitted in the period after. Task τ_1 has a single predictable interval of length $e_{1,2} = 4$ while τ_2 has two predictable intervals of lengths $e_{2,2} = 4$ and $e_{2,3} = 3$. The first and last interval of both τ_1 and τ_2 are special compatible intervals. These intervals are needed to execute the associated peripheral driver (including interrupt handlers) and set up the reception and transmission buffers in main memory (i.e. read and write system calls). More details are provided in section 5.6. I/O flows can be scheduled both during execution phases and while the CPU is idle. The described scheme can be modeled as a hierarchical scheduling system [61], where the CPU schedule of predictable intervals supplies available transmission time to I/O flows.

5.4 Architectural constraints and solutions

Predictable intervals are executed in a radically different way compared to the speculative execution model that COTS components are typically designed to support. In this section, we detail the challenges

and solutions to implement the PRedictable Execution Model on top of a COTS architecture.

5.4.1 Caching and prefetch

Our general strategy to implement the memory phase consists of two steps:

1. we determine the complete set of memory regions that are accessed during the interval. Each region is a continuous area in virtual memory. In general, its start address can only be determined at run-time, but its size A is known at compile time.

2. During the memory phase, we prefetch all cache lines that contain instructions and data for required regions; most COTS instruction sets include a prefetch instruction that can be used to load specific cache lines in last level cache.

Step (1) will be detailed in section 5.5. Step (2) can be successful only if there is no cache *self-eviction*, that is, prefetching a cache line never evicts another line loaded during the same memory phase. In the remainder of this subsection, we describe self-eviction prevention.

Most COTS CPUs implement the last-level cache as an N-way set associative cache. Let B be the total size of the cache and L be the size of each cache line in bytes. Then the byte size of each of the N cache ways is $W = B/N$. An associative set is the set of all cache lines, one for each way, which have the same index in cache; there are W/L associative sets. Last level cache is typically physically tagged and physically indexed, meaning that cache lines are accessed based

on physical memory addresses only. We also assume that last level cache is not exclusive, that is, when a cache line is copied to a higher cache level it is not removed from the last level.

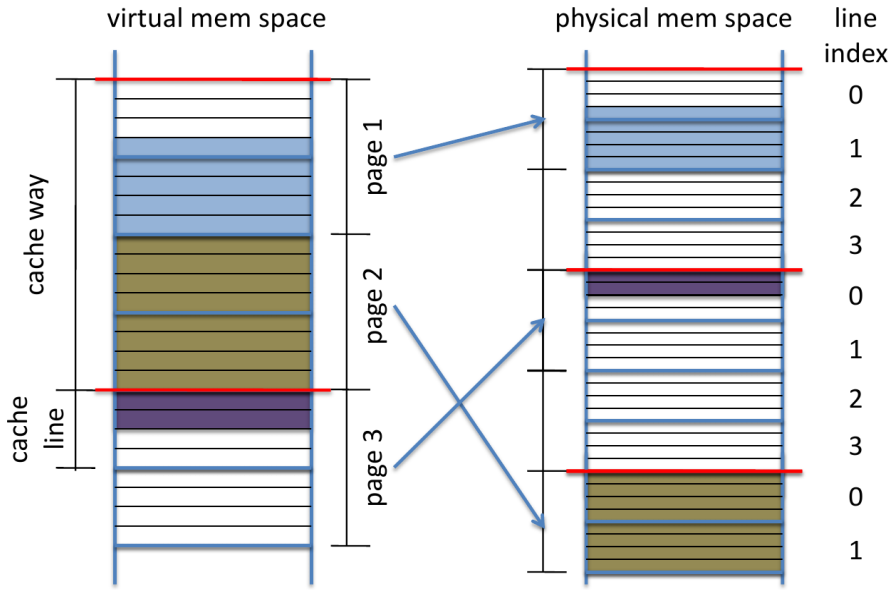

Figure 5.4: Cache organization with one memory region

Figure 5.4 shows an example where $L = 4, W = 16$ (parameters are chosen to simplify the discussion and are not representative of typical systems). The main idea behind our conflict analysis is as follows: we compute the maximum amount of entries in each associative set that are required to hold the cache lines prefetched for all memory regions in a scheduling interval. Based on the cache replacement policy, we then derive a safe lower bound on the amount of entries that can be prefetched in an associative set without causing any self-eviction.

Consider a memory region of size A. The region will occupy at most $K = \lceil \frac{A-1}{L} \rceil + 1$ cache lines. As shown in Figure 5.4 for a region with $A = 15, K = 5$, the worst case is produced when the region uses a single byte in its first cache line. Assume now that virtual memory addresses coincide with physical addresses. Then since the region is contiguous and there are W/L cache lines in each way, the maximum number of entries used in any associative set by the region is $\lceil \frac{K}{W/L} \rceil$. For example, the region in Figure 5.4 requires two entries in the set with index 0. We then derive the maximum number of entries for the entire interval by summing the entries required for each memory region. Unfortunately, this is not generally true if the system employs paged virtual memory. If the page size P is smaller than the size W of each way, the index of each cache line inside the cache way is different for virtual and physical addresses. In the example of Figure 5.4 with $P = 8$, the number of entries for the memory region is increased from 2 to 3. We consider two solutions: 1) if the system supports it, we can select a page size multiple of W just for our specific process. This solution, which we employed in our implementation, solves the problem because the index in cache for virtual and physical addresses is the same no matter the page allocation. 2) We use a modified page allocation algorithm in the OS. Note that a suitable allocation algorithm could decrease the required number of associative entries by controlling the allocation in physical memory of multiple regions. We plan to pursue this solution in our future work.

Due to space constraints, a thorough discussion of cache replacement policies is provided in [43]. Let Q be the maximum number of entries in any associative set required by the scheduling interval. Fur-

thermore, let Q' be the number of such entries relative to cache lines that are accessed multiple times during the memory phase; in our implementation, this only includes the cache lines that contain the small amount of instructions of the memory phase itself, so $Q' = 1$. In [43] we prove the following:

Theorem 3. *A memory phase will not suffer any cache self-eviction if Q is at most equal to:*

- N*: for FIFO or LRU replacement policy;*
- $N/(2^{Q'}) + Q'$*: for pseudo-LRU replacement policy;*
- 1*: for random replacement policy.*

Finally, for systems implementing paged virtual memory, we employ the following three assumptions:

- the CPU supports hardware Translation Lookaside Buffer (TLB) management;
- all pages used by predictable intervals are locked in main memory;
- the TLB is large enough to contain all page entries for a predictable interval without suffering any conflict.

Under such assumptions, each page used in a predictable interval can cause at most one TLB miss during the memory phase, which requires a number of fetches in main memory equal to at most the level of the page table.

5.4.2 Interval length enforcement

As described in section 5.3, each predictable interval is required to execute for exactly $e_{i,j}$ time units. If $e_{i,j}$ is set to be at least $e_{i,j}^{\text{mem}} + e_{i,j}^{\text{exec}}$, the computation in the execution phase is guaranteed to terminate at or before the end of the interval. The interval can then enter active wait until $e_{i,j}$ time units have elapsed since the beginning of its memory phase. In our implementation, elapsed time is computed using a CPU performance counter that is directly accessible by the task; therefore, neither OS nor memory interaction are required to enforce interval length.

5.4.3 Scheduling synchronization

In our model, peripherals are only allowed to transmit during a predictable interval's execution phase or while the CPU is idle. To compute the peripheral schedule, the peripheral scheduler must thus know the status of the CPU schedule. Synchronization can be achieved by connecting the peripheral scheduler to a peripheral interconnection as shown in figure 5.1. Scheduling messages can then be sent by either a task or the OS to the peripheral scheduler. In particular, at the end of each memory phase the task sends to the peripheral scheduler the remaining amount of time until the end of the current predictable interval. Note that propagating a message through the interconnection takes non-zero time. Since this time is typically negligible compared to the size of a scheduling interval[2], we will ignore

[2]In our implementation we measured an upper bound to the message propagation time of 1us, while we envision scheduling intervals with a length of 100-

it in the rest of our discussion.

Finally, to avoid executing interrupt handlers during predictable intervals, a peripheral should only raise interrupts to the CPU during compatible intervals of its serviced task. As we describe in section 5.6, in our I/O management scheme peripherals raise interrupts through their assigned real-time bridge. Since the peripheral scheduler communicates with each real-time bridge, it is used to block interrupt propagation outside the desired compatible intervals. Note that blocking real-time bridge interrupts to the CPU will not cause any loss of input data because the real-time bridge is capable of independently acknowledging the peripheral and storing all incoming data in the bridge local buffer.

5.5 Programming model

Our system supports COTS applications written in standard high-level languages such as C. Unmodified code can be executed within one or more compatible intervals. To create predictable intervals, programmers add source code annotations as C preprocessor macros. The PREM real-time compiler creates code for predictable intervals so that it does not incur cache misses during the execution phase and the interval itself has a constant execution time.

In order to create a predictable interval, the programmer should first perform the cache and architecture analysis as presented in previous section, then based on the results and task information, partition

1000us.

the task into intervals. Compatible intervals can be handled in the conventional way while each predictable interval is encapsulated into a single function. All code within the function, and any functions transitively called by this function is executed as a single predictable interval. Due to current limitations of our static code analysis, we impose several constraints upon the code within a predictable interval:

1. Only scalar and array-based memory accesses should occur within a predictable interval; there should be no use of pointer-based data structures.

2. The code can use data structures, in particular arrays, that are not local to functions in the predictable intervals, e.g. they are allocated either in global memory or in the heap[3]. However, the programmer should specify the first and last address that is accessed within the predictable interval for each non-local data structure. In general, it is difficult for the compiler to determine the first and last access to an array within a piece of code. The compiler needs this information to be able to load the relevant portion of each array into the last-level cache.

3. The functions within a predictable interval should not be called recursively. As described below, the compiler will inline callees into callers to make all the code within an interval contiguous in virtual memory. Furthermore, no system calls should be made by code within a predictable interval.

[3]Note that data structures in the heap must have been previously allocated during a compatible interval.

4. Code within a predictable interval may only have direct calls. This alleviates the needs for pointer-analysis to determine the targets of indirect function calls; such analysis is usually imprecise and would bloat the code within the interval.

5. No stack allocations should occur within loops. Since all variables must be loaded into the cache at function entry, it must be possible for the compiler to safely hoist the allocations to the beginning of the function which initiates the predictable interval.

While these constraints may seem restrictive, some of these features are rarely used in real-time C code e.g., indirect function calls, and the others are met by many types of functions. We believe that the benefit of faster, more predictable behavior for program hotspots outweighs the restrictions imposed by our programming model. Furthermore, existing code that is too complex to be compiled into predictable intervals can still be executed inside compatible intervals. Therefore, our model permits a smooth transition for legacy systems.

Notice that, the compiler can be used to verify that all of the aforementioned restrictions are met. Simple static analysis can determine whether there is any irregular data structure usage, indirect function calls, or system calls. During compilation, the compiler employs several transforms to ensure that code marked as being within a predictable interval does not cause a cache miss.

First, the compiler inlines all functions called (either directly or transitively) during the interval into the top-level function defining the interval. This ensures that all program analysis is intra-

procedural and that all the code for the interval is contiguous within virtual memory.

Second, the compiler can transform the program so that all cache misses occur during the memory phase, which is located at the beginning of the predictable scheduling interval. To be specific, it inserts code after the function prologue to prefetch the code and data needed to execute the interval. Based on the described constraints, this includes three types of contiguous memory regions: (1) the code for the function; (2) the actual parameters passed to the function and the stack frame (which contains local variables and register spill slots); and (3) the data structures marked by the programmer as being accessed by the interval.

Third, the compiler inserts code to send scheduling messages to the peripheral scheduler as will be described in section 5.6. Finally, the compiler emits code at the end of predictable interval to enforce its constant length. In particular, the compiler identifies all return instructions within the function and adds the required code before them.

5.6 Evaluation

In order to verify the validity and practicality of PREM, we implemented the key components of the system. In this section, we describe our evaluation, first introducing the new *peripheral scheduler*, and some new features introduced on the real-time bridges, followed by the corresponding software driver and OS calibration effort. We then discuss our compiler implementation and analyse its effective-

ness on a DES benchmark. Finally, using synthetic tasks we measure the effectiveness of the PREM system as a function of cache stall time, and show traces of PREM when running on COTS hardware.

5.6.1 PREM hardware components

As discussed in section 5.3, in order to enforce its real-time I/O scheduling PREM uses slightly modified real-time bridges and a new *peripheral scheduler*, based on the *reservation controller* (see section 3.4). Here, we briefly describe the additions to these components which we made to provide the mechanism for PREM execution. First we discuss the real-time bridge component, then we describe the peripheral scheduler component.

Our **real-time bridge** prototype is still wired to peripheral scheduler, which allows direct communication between these components. In this last version, three wires are used to transmit information between each real-time bridge and the peripheral scheduler: `data_ready`, `data_block`, and `interrupt_block`. The `data_ready` wire is an output signal sent to the peripheral scheduler which is asserted whenever the COTS peripheral has buffered data in the real-time bridge. The peripheral scheduler sends two signals, `data_block` and `interrupt_block`, to the real-time bridge. The `data_block` signal is asserted to block the real-time bridge from transferring data in DMA mode over the PCI-E bus from its local buffer to main memory or viceversa. The `interrupt_block` signal is a new signal in the real-time bridge implementation, added to support PREM, and instructs the real-time bridge to not further raise interrupts.

The **peripheral scheduler** is a unique component in the system, connected directly to each real-time bridge. Unlike our previous *reservation controller* where the peripherals were scheduled asynchronously with the CPU, PREM requires the CPU and peripherals to coordinate their access to main memory. While the OS real-time scheduler runs its scheduling model for CPU tasks, the peripheral scheduler provides a hardware implementation of a child scheduling model used to schedule peripherals, with each peripheral given a sporadic server to schedule its traffic. The peripheral scheduler is connected to the PCI-E bus, and exposes a set of registers accessible from the main CPU. In the `configuration` register, constant parameters such as the maximum cache write-back time are stored. Writing a value to the `yield` register indicates that the CPU will not access main memory for the given amount of time, and I/O peripherals should be allowed to read to and write from RAM. The value written to the `yield` register contains a 14 bit unsigned integer indicating the number of microseconds to permit peripheral traffic with main memory, as described in section 5.4. The CPU can also use the `yield` register to allow interrupts to be raised by peripherals. Another 14 bit unsigned integer indicates the number of microseconds to allow peripheral interrupts, and a 3 bit interrupt mask selects which peripherals should be allowed to raise the interrupts. In this way, different CPU tasks can predictably service interrupts from different I/O peripherals.

5.6.2 Software evaluation

The software effort required to implement our PREM prototype involves two aspects: (1) creating a new driver for the peripheral scheduler, and (2) calibrating the OS to eliminate undesired execution interference. We now discuss these, in order.

The two custom hardware components, the real-time bridge and the peripheral scheduler, each require a software driver to be controller from the main CPU. Additionally, each peripheral requires a driver running on the real-time bridge's CPU to control the COTS peripheral. The real-time bridge drivers have been already discussed in sections 3.5 and 4.2. The driver for the peripheral scheduler is straightforward, mapping the bus addresses corresponding to the exposed registers to user space where a PREM-compiled process can access them.

For our experiments, we use a Intel Q6700 CPU with a 975X system controller; we set the CPU frequency to 1Ghz obtaining a measured memory bandwidth of 1.8Ghz/s to configure the system in line with typical values for embedded systems. We also disable the speculative CPU HW prefetcher since it negatively impacts the predictability of any real-time task. The Q6700 has four CPU cores and each pair of cores shares a common level 2 (last level) cache. Each cache is 16-associative with a total size of $B = 4$ Mbytes and a line size of $L = 64$ bytes.

As we discussed in section 1.4, since we use a PC platform running a COTS Linux operating system, there are many potential sources of timing noise, such as interrupts, kernel threads, and other processes,

which must be removed for our measurements to be meaningful. For this reason, in order to emulate at our best a typical uni-processor embedded real-time platform, we divided the 4 cores in two partitions. The *system partition*, running on the first pair of cores, receives all interrupts for non-critical devices (ex: the keyboard) and runs all the system activities and non real-time processes (ex: the shell we use to run the experiments). The *real-time partition* runs on the second pair of cores. One core in the real-time partition runs our real-time tasks together with the drivers for real-time bridges and the peripheral scheduler; the other core is not used. As you can notice, this approach is similar to the one used in ASMP-Linux. In this case we needed just to emulate a uni-processor system, but, in our future work, extending the PREM model to multicore systems, we also plan to integrate PREM and ASMP-Linux.

Note that the cores of the system partition can still produce a small amount of unscheduled bus and main memory accesses, or raise rare inter-processor interrupts (IPI) that can not be easily prevented. However, in our experiments we found these sources of noise to be negligible. Finally, to solve the paging issue detailed in section 5.4, we used a large, 4MB page size, just for the real-time tasks, using the HugeTLB feature of the Linux kernel for large page support.

5.6.3 Compiler evaluation

We built the PREM real-time compiler prototype using the LLVM Compiler Infrastructure [34], targeting the compilation of C code. LLVM was extended by writing self-contained analysis and transfor-

mation passes, which were then loaded into the compiler.

In the current PREM real-time compiler prototype, we rely on the programmer to partition the task into predictable and compatible intervals . The partitioning is done by putting each predictable interval into its own function. The beginning and end of the scheduling interval correspond to the entry and exit of the function, and the start of execution phase (the end of memory access phase) is manually marked by the programmer. We assume that non-local data accessed during a predictable interval exists in continuous memory spaces, which can be prefetched by a set of `PREFETCH_DATA(start_address, size)` macros that must be placed by the programmer during the memory phase. The implementation of this macro does the actual prefetching of the data into level 2 cache by prefetching every cache line in the given range with the i386 `prefetcht2` instruction. After the memory phase, the programmer adds a `STARTEXECUTION(wcet)` macro to indicate the beginning of the execution phase. This macro measures the amount of time remaining in the predictable interval using the CPU performance counter, and writes the time remaining to the `yield` register in the peripheral scheduler.

All remaining operations needed to transform the interval are performed by a new LLVM function pass. The pass iterates over all the functions in a compilation unit. When a function representing a predictable interval is found, the pass performs code transformation. First, our transform inlines called functions using preexisting LLVM inlining functions. This ensures that there are only a single stack frame and segment of code that need to be prefetched into the cache. Second, our transform inserts code to read the CPU perfor-

mance counter at the beginning of the interval and save the current time. Third, it inserts code to prefetch the stack frame and function arguments. Bringing the stack frame into the cache is done by inserting instructions into the program to fetch the stack pointer and frame pointer. Code is then inserted to prefetch the memory between the stack pointer and slightly beyond the frame pointer (to include function arguments) using the `prefetcht2` instruction. Fourth, the transform prefetches the code of the function. This is done by transforming the program so that the function is placed within a unique ELF section. We then use a linker script to define variables pointing to the beginning and end of this unique ELF section. The compiler then adds code that prefetches the memory inside the ELF section. Finally, the pass identifies all return instructions inside the predictable interval function and adds a special function epilog before them. The epilog performs interval length enforcement by looping until the performance counter reaches the worst-case cycle count based on the time value saved at the beginning of the interval. It may also enable peripheral interrupts by writing the worst-case interrupt processing time to the peripheral scheduler's `yield` register.

To verify the correctness of the PREM real-time compiler prototype and to test its applicability, we used LLVM to compile a DES cypher benchmark. The DES benchmark was selected because it represents a typical real-time data flow application. The benchmark comprises one scheduling interval which encrypts a variable amount of data. We compiled it as both a predictable and a compatible interval (e.g. with and without prefetching), and measured number of cache misses with a performance counter. Adapting the interval

Data size	4K	8K	32K	128K	512K	1M
Compatible	138	254	954	3780	15k	31k
Predictable	2	2	4	2	1	81

Table 5.1: DES benchmark.

required no modification to any cypher functions and a total of 11 `PREFETCH_DATA` macros.

Results are shown in Table 5.1 in terms of the number of cache misses suffered in the execution phase of the predictable interval (after prefetching), and in the entire compatible interval. Data size is in bytes. The compatible interval suffers an excessive number of cache misses, which increases roughly proportionally with the amount of processed data. Conversely, the execution phase of the predictable interval has almost zero cache misses, only suffering a small increase when large amounts of data are being processed. The reason the number of cache misses is not zero is that the Q6700 CPU core used in our experiments uses a random cache replacement policy, meaning that with more than one contiguous memory region the probability of self-eviction is non-zero. In all the following experiments, we observed that the number of self-evictions is typically so small that it can be considered negligible.

5.6.4 WCET experiments with synthetic tasks

In this section, we evaluate the effects of PREM on the execution time of a task. To quickly explore different execution parameters, we developed two synthetic applications. In our `linear_access` application,

each scheduling interval operates on a 256-kilobyte global data structure. Data is accessed sequentially, and we vary the amount of computation performed between memory references. The `random_access` application is similar, except that references inside the data structure are nonsequential. For each application, we measured the execution time after compiling the program in two ways: into predictable intervals which prefetch the accessed memory, and into standard, compatible intervals. For each type of compilation, we ran the experiment in two ways, with and without I/O traffic transmitted by an 8-lane PCI-E peripheral with a measured throughput of 1.2Gbytes/s. In the case of compatible intervals, we transmitted traffic during the entire interval to mirror the worst case according to the traditional execution model.

Figures 5.5 and 5.6 show the observed worst case execution time for any scheduling interval as a function of the cache stall time of the application, averaged over 10 runs. The cache stall time represents the percentage of time required to fetch cache lines out of an entire compatible interval, assuming a fixed (best-case) fetch time based on the maximum measured main-memory throughput. Only a single line is shown for predictable intervals because experiments confirmed that injecting traffic during the execution phase does not increase execution time. In all cases, the computation time decreases with an increase in stall time. This is because stall time is controlled by varying the amount of computation between memory references. Furthermore, execution times should not be compared between the two figures because the two applications execute different code.

In the `random_access` case, predictable intervals outperform com-

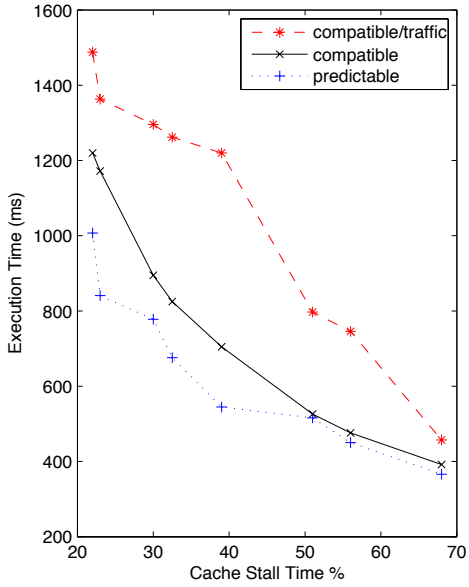

Figure 5.5: `random_access`

patible intervals (without peripheral traffic) by up to 28%, depending on the cache stall time. We believe this effect is primarily due to the behavior of DRAM main memory. Specifically, accesses to adjacent addresses can be served quicker in burst mode than accesses to random addresses. Thus, we can decrease the execution time by loading all the accessed memory into cache, in order, at the beginning of each predictable interval. Furthermore, note that transmitting peripheral traffic during a compatible interval can increase execution time by more than 60% in the worst case. In figure 5.6, predictable intervals perform worse than compatible intervals (without peripheral traffic). We believe this is mainly due to out-of-order execution in the Q6700 core. In compatible intervals, while the core performs a cache fetch,

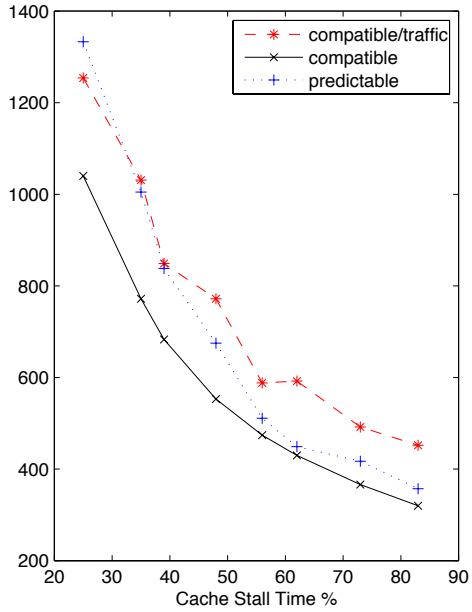

Figure 5.6: `linear_access`

instructions in the pipeline that do not depend on the fetched data can continue to execute. When performing linear accesses, fetches require less time and this effect is magnified. Furthermore, the gain in execution time for the case with peripheral traffic is decreased: this occurs because bursting data on the memory bus reduces the amount of blocking time suffered by a task due to peripheral interference (this effect has been previously analyzed in detail [46]). In practice, we expect the effect of PREM on an application's execution time to be between the two figures, based on the specific memory access pattern.

5.6.5 System-wide coscheduling traces

We now present execution traces of the implemented system which demonstrate the advantage of the PREM coscheduling approach. The traces are obtained by using the peripheral scheduler as a logic analyzer for the various signals which are being sent to or from the real-time bridges, `data_block`, `data_ready`, and `interrupt_block`. Additionally, the peripheral scheduler has a `trace` register which allows timestamped trace information to be recorded with a one microsecond resolution when instructed by the main CPU, such as at the start and end of an execution interval. An execution trace is shown for a task running the traditional COTS execution model in figure 5.7, and the same task running within the PREM model is shown in figure 5.8[4].

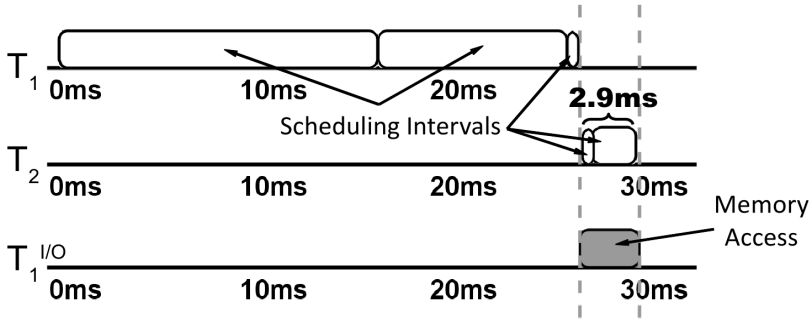

Figure 5.7: An unscheduled bus-access trace (without PREM)

In the first trace (figure 5.7), although the execution is divided

[4]The (compatible) intervals at the end of T_1 and at the start of T_2 were measured as 0.107ms and 0.009ms, respectively, and have been exaggerated in the figures to be visible.

into unpreemptable intervals, there is no memory phase prefetch or constant execution time guarantees. When the scheduling intervals of task T_1 finish executing, an I/O peripheral begins to access main memory, which may happen if T_1 had written output data into RAM. Task T_2 then executes, suffering cache misses that compete for main memory bandwidth with the I/O peripheral. Due to the cold cache and peripheral interference, the execution time of T_2 grows from 0.5 ms (the execution time with warm cache and no peripheral I/O), to 2.9ms as shown in the figure, an increase of about 600%.

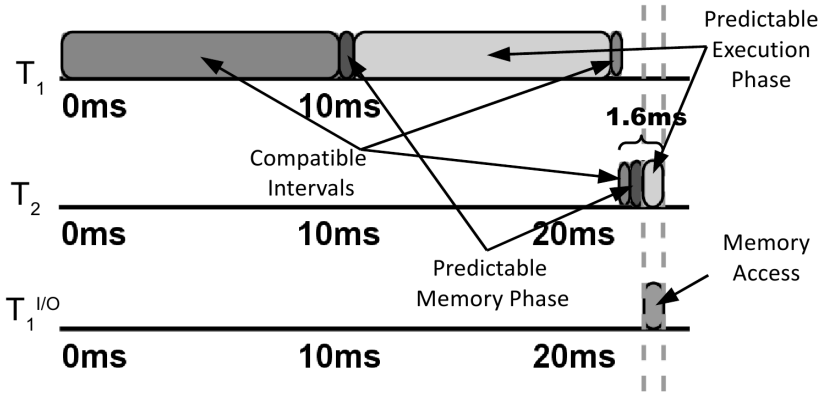

Figure 5.8: A scheduled trace using PREM

In the second trace (figure 5.8), the system executes according to the PREM execution model, where peripherals only access the bus when permitted by the peripheral scheduler. The predictable interval is divided into a memory phase and an execution phase. Instead of competing for main memory access, task T_2 gets contentionless access to main memory during the memory phase. After all to-be-accessed

data is loaded into the cache, the execution phase begins which incurs no cache misses, and the peripheral is allowed to access the data in main memory. The constant execution time for the predictable interval in the PREM execution model is 1.6ms, which is significantly lower than the worst-case observed for the unscheduled trace (and is about the same as the execution time of the scheduling interval of T_2 in the unscheduled trace with a cold cache and no peripheral traffic).

Conclusions

In this thesis we focused on satisfying hard real-time constraints on COTS components. In particular we addressed the sources of unpredictability introduced by the operating system and I/O subsystem.

In order to mitigate operating system overhead and latency, ASMP-Linux, a variant of the Linux operating system, has been developed. The test results show how ASMP-Linux is capable of minimizing both operating system overhead and latency, thus providing deterministic results for the tested applications.

We have also presented a framework for providing real-time control of the I/O peripherals in a COTS-based embedded system. This framework involves interposing real-time bridges between COTS peripherals and the COTS interconnect, all of which communicate with a central reservation controller. In this way, we are able to schedule bus transactions such that all I/O deadlines are met, as well as prevent data loss by buffering traffic of high-bandwidth peripherals when bus access is prohibited. We have shown through experiments that an unmodified COTS I/O system can cause excessive I/O delay leading to deadline misses, while our prototype real-time COTS-based I/O framework provides deterministic delays and meets all I/O deadlines. We demonstrated the way in which classical uni-processor scheduling theory can be applied within our framework, and created hardware logic to implement the Rate Monotonic and Sporadic Server scheduling policies on COTS peripheral bus traffic. We provided analysis to determine maximum buffer size and delay based on the arrival and

service curves of I/O traffic.

Multi-flow real-time bridges have been also developed. They extend real-time bridges capabilities, being also able to schedule transactions within the peripheral. In this way, a peripheral can be shared among tasks with different criticalities, improving the device utilization, and the system costs and scalability. We also developed a prototype and performed tests to show how, without our architecture, a critical task using a shared device can miss its deadline, while, using multi-flow real-time bridge the tasks always meet their deadlines.

We also discussed the concept and implementation of a novel task execution model, PRedictable Execution Model (PREM), that aims to cope with two last sources of jitter: interference between peripherals and CPU, and interference between a critical task and drivers. Our evaluation shows that by enforcing a high-level coschedule among CPU tasks and peripherals, PREM can greatly reduce or outright eliminate low-level contention for shared resource access. We plan to further develop our solution in two main directions. First, we will study extensions to our compiler infrastructure to lift some of the more restrictive code assumptions and compile and test a larger set of benchmarks. Second, it is important to recall that contention for shared resources becomes more severe as the number of active components increases. In particular, worst-case execution time can greatly degrade in multicore systems [47]. Since PREM can make the system contentionless, we predict that the benefits of our approach will become even more significant when applied to multiple-processor systems.

Finally, we will integrate ASMP-Linux with multiprocessor PREM,

achieving a fully-predictable and high-speed COTS-based system.

Appendix A

Full experiments data for ASMP-Linux

The following tables show the results of the experiments performed with all workloads (idle, CPU, AIO, SIO, and MIX) on the three hardware configurations $S1$, $S2$, and $S3$.

The results of all tests are coherent on all platforms. However, a short note is due to explain why results of the SIO workload in all platforms show that the R_w test case is worse than the N test case.

The reason is that in the SIO workload the system is saturated by the interrupts from the disk. In the R_w test case these interrupts are always handled by the same CPU that executes the test program; conversely, in the N test case, the test program can run on any CPU, thus it is often not affected by the interrupts storm.

Proc	Avg	StDev	Min	Max
		IDL		
N	2.072	0.034	2.031	2.376
R_w	4.176	0.062	4.117	4.757
R_b	1.784	0.077	1.747	2.493
A_{on}	1.764	0.073	1.723	2.429
A_{off}	0.286	0.008	0.267	0.319
		CPU		
N	7199.851	606.625	6010.521	7610.221
R_w	12.769	1.203	9.790	18.527
R_b	9.872	1.403	6.782	14.023
A_{on}	10.472	1.014	7.022	13.905
A_{off}	8.849	0.957	5.670	11.992
		AIO		
N	6264.578	776.284	4793.751	9047.211
R_w	40.347	4.088	25.538	47.532
R_b	1.889	0.135	1.768	2.703
A_{on}	1.685	0.096	1.602	2.485
A_{off}	0.286	0.004	0.276	0.315
		SIO		
N	3.664	1.393	2.108	7.161
R_w	8.244	0.666	6.755	13.752
R_b	1.872	0.147	1.603	2.332
A_{on}	1.647	0.074	1.535	2.019
A_{off}	0.318	0.010	0.295	0.363
		MIX		
N	20275.784	6072.575	12.796	34696.051
R_w	28.459	12.945	10.721	48.837
R_b	27.461	9.661	3.907	42.213
A_{on}	30.262	8.306	8.063	41.099
A_{off}	27.847	7.985	6.427	38.207

Table A.1: Operating system overhead on configuration S1 (in milliseconds).

Proc	Avg	StDev	Min	Max
		IDL		
N	1.481	0.190	1.447	7.277
R_w	3.707	0.188	3.643	9.581
R_b	1.360	0.025	1.326	1.516
A_{on}	1.420	0.021	1.392	1.558
A_{off}	0.000	0.000	0.000	0.000
		CPU		
N	7486.825	746.094	6399.201	7998.691
R_w	3.638	0.191	3.566	9.546
R_b	1.402	0.019	1.380	1.492
A_{on}	1.372	0.180	1.342	7.032
A_{off}	0.000	0.000	0.000	0.000
		AIO		
N	5967.168	1549.961	3001.521	16609.651
R_w	4.325	0.192	4.272	10.266
R_b	1.415	0.024	1.386	1.525
A_{on}	1.518	0.184	1.476	7.282
A_{off}	0.000	0.000	0.000	0.000
		SIO		
N	1.702	0.557	1.462	8.596
R_w	6.688	0.561	5.784	12.252
R_b	1.405	0.023	1.369	1.529
A_{on}	1.382	0.181	1.353	7.085
A_{off}	0.000	0.000	0.000	0.000
		MIX		
N	18513.615	5996.971	1.479	33993.351
R_w	4.215	0.226	3.913	10.146
R_b	1.420	0.029	1.393	1.554
A_{on}	1.490	0.044	1.362	1.624
A_{off}	0.000	0.000	0.000	0.000

Table A.2: Operating system overhead on configuration S2 (in milliseconds).

Proc	Avg	StDev	Min	Max
		IDL		
N	3.552	0.048	3.519	4.325
R_w	3.534	0.042	3.514	4.066
R_b	0.561	0.071	0.547	1.663
A_{on}	0.577	0.068	0.548	1.541
A_{off}	0.001	0.000	0.001	0.001
		CPU		
N	8773.632	796.256	8001.631	9601.331
R_w	3.465	0.018	3.438	3.772
R_b	0.552	0.029	0.544	0.796
A_{on}	0.554	0.029	0.545	0.803
A_{off}	0.000	0.000	0.000	0.000
		AIO		
N	6953.769	1638.890	4430.931	17497.731
R_w	3.628	0.447	3.444	5.099
R_b	0.553	0.032	0.543	0.806
A_{on}	0.554	0.032	0.541	0.815
A_{off}	0.000	0.000	0.000	0.000
		SIO		
N	249.870	83.592	81.021	315.688
R_w	894.175	3.718	883.581	904.691
R_b	0.517	0.041	0.474	0.778
A_{on}	0.508	0.039	0.472	0.842
A_{off}	0.000	0.000	0.000	0.000
		MIX		
N	20065.194	6095.807	0.606	32472.931
R_w	3.477	0.024	3.431	3.603
R_b	0.554	0.031	0.525	0.807
A_{on}	0.556	0.032	0.505	0.811
A_{off}	0.000	0.000	0.000	0.000

Table A.3: Operating system overhead on configuration S3 (in milliseconds).

Proc	Avg	StDev	Min	Max
		IDL		
N	6.399	1.014	5.632	75.477
R_w	6.019	43.468	5.213	4352.040
R_b	5.811	0.450	5.341	12.826
A_{on}	6.277	0.415	5.723	12.869
A_{off}	6.424	0.236	5.962	15.802
		CPU		
N	3845.041	221531.315	6.712	$12.792 \cdot 10^6$
R_w	7.113	1.041	6.576	21.674
R_b	6.866	0.918	6.301	32.162
A_{on}	6.851	0.391	6.379	13.003
A_{off}	6.956	0.209	6.509	9.429
		AIO		
N	2474.152	34619.008	6.869	$1.504 \cdot 10^6$
R_w	13.172	9.050	7.120	841.115
R_b	10.649	10.469	7.029	1015.144
A_{on}	12.542	2.490	7.256	26.999
A_{off}	10.529	1.652	7.400	23.151
		SIO		
N	68.286	650.755	5.811	10882.761
R_w	8.998	40.020	5.433	1208.948
R_b	6.292	4.598	5.347	216.600
A_{on}	6.629	1.805	5.758	30.018
A_{off}	6.661	2.103	5.944	37.285
		MIX		
N	13923.606	220157.013	6.946	$5.001 \cdot 10^6$
R_w	10.970	8.458	6.405	603.272
R_b	10.027	5.292	6.506	306.497
A_{on}	8.074	1.601	6.683	20.877
A_{off}	8.870	1.750	6.839	23.230

Table A.4: Operating system latency on configuration S1 (in microseconds).

Proc	Avg	StDev	Min	Max
		IDL		
N	5.346	0.592	4.879	15.655
R_w	5.206	0.666	4.898	15.151
R_b	5.284	0.297	5.015	9.948
A_{on}	5.290	0.267	5.056	9.785
A_{off}	5.289	0.056	5.172	6.459
		CPU		
N	680.058	95398.550	4.920	$13.491 \cdot 10^6$
R_w	5.331	0.577	5.061	15.428
R_b	5.313	0.321	4.869	9.312
A_{on}	5.020	0.233	4.845	8.841
A_{off}	5.102	0.094	4.966	5.863
		AIO		
N	13012.278	252843.297	4.768	$9.993 \cdot 10^6$
R_w	5.919	3.812	4.920	371.279
R_b	5.487	2.049	4.573	200.549
A_{on}	4.956	0.219	4.767	8.797
A_{off}	5.298	0.105	5.143	6.304
		SIO		
N	34.399	276.160	4.909	6161.732
R_w	6.865	20.038	5.033	758.550
R_b	6.065	14.296	5.040	732.543
A_{on}	5.479	0.279	5.145	10.354
A_{off}	5.361	0.113	5.141	7.134
		MIX		
N	24402.723	331861.500	4.904	$4.997 \cdot 10^6$
R_w	5.996	1.249	4.960	39.982
R_b	5.511	1.231	4.603	109.964
A_{on}	5.120	0.275	4.917	9.370
A_{off}	5.441	0.199	5.207	6.716

Table A.5: Operating system latency on configuration S2 (in microseconds).

Proc	Avg	StDev	Min	Max
		IDL		
N	1.753	0.344	1.626	44.707
R_w	1.728	0.172	1.656	7.044
R_b	1.642	0.022	1.548	2.088
A_{on}	1.630	0.018	1.590	2.076
A_{off}	1.593	0.015	1.566	1.890
		CPU		
N	296.696	41677.046	1.614	$5.894 \cdot 10^6$
R_w	1.765	0.571	1.656	50.377
R_b	1.719	0.055	1.602	2.124
A_{on}	1.664	0.019	1.626	2.238
A_{off}	1.581	0.016	1.542	1.818
		AIO		
N	1.874	2.664	1.590	355.958
R_w	1.898	1.928	1.608	68.029
R_b	1.764	0.700	1.578	63.973
A_{on}	1.703	0.023	1.638	2.226
A_{off}	1.557	0.019	1.530	2.004
		SIO		
N	91.204	392.681	1.680	3974.279
R_w	28.172	23.220	1.788	83.521
R_b	1.837	0.231	1.578	9.966
A_{on}	1.779	0.088	1.626	3.648
A_{off}	1.613	0.059	1.554	2.628
		MIX		
N	182577.713	936480.576	1.554	$9.095 \cdot 10^6$
R_w	1.999	1.619	1.722	66.883
R_b	1.756	0.650	1.548	63.985
A_{on}	1.721	0.034	1.674	3.228
A_{off}	1.639	0.025	1.602	2.466

Table A.6: Operating system latency on configuration S3 (in microseconds).

Bibliography

[1] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A performance characterization of high definition digital video decoding using h.264/avc. In *Proc. of the IEEE International Workload Characterization Symposium*, Oct 2005.

[2] Internet FAQ Archives. Real-time computing FAQ. http://www.faqs.org/faqs/realtime-computing/faq/.

[3] L. Sha B. Sprunt, J.P. Lehoczky. Scheduling sporadic and aperiodic events in a hard real-time system. Technical report, CMU, 1989.

[4] Stanley Bak, Emiliano Betti, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Real-time control of i/o cots peripherals for embedded systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, Washington, D.C., USA, December 2009.

[5] Emiliano Betti, Stanley Bak, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Real-time management of shared peripherals for cots-based embedded systems. *IEEE Transactions on Computers*, 2010. *[to be submitted].*

[6] Emiliano Betti, Daniel Pierre Bovet, Marco Cesati, and Roberto Gioiosa. ASMP-Linux. www.sprg.uniroma2.it/asmplinux.

[7] Emiliano Betti, Daniel Pierre Bovet, Marco Cesati, and Roberto Gioiosa. Hard real-time performances in multiprocessor-

embedded systems using ASMP-Linux. *Eurasip Journal on Embedded Systems, Hindawi Publishing Corporation*, 2008(82648), 2008.

[8] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, 2000.

[9] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Really, 3rd edition, 2005.

[10] S. Brosky and S. Rotolo. Shielded processors: Guaranteeing sub-millisecond response in standard Linux. In *Fourth Real-Time Linux Workshop*, Boston, December 2002.

[11] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Second Edition*. Kluwer Academic Publishers, Boston, 2004.

[12] IBM Corp. The Cell project at IBM Research. Available from http://www.research.ibm.com/cell/home.html.

[13] Intel Corp. Hyper-Threading technology. Available from http://www.intel.com/technology/hyperthread/index.htm.

[14] Intel Corp. Intel xeon processor x7460. http://download.intel.com/design/xeon/datashts/32033501.pdf.

[15] Intel Corp. Intel Xeon processors. Available from http://www.intel.com/products/processor/xeon/index.htm#top.

[16] Intel Corp. Intel Core2 Duo Mobile Processor datasheet, 2006. Available from http://download.intel.com/design/mobile/datashts/31407801.pdf.

[17] Intel Corp. Intel Core2 Extreme Processor X6800 and Intel Core2 Duo Desktop Processor E6000 sequence datasheet, 2006. Available from http://download.intel.com/design/processor/designex/31368501.pdf.

[18] Intel Corporation. Intel 82598 10 gigabit ethernet controller. www.intel.com/assets/pdf/prodbrief/317796.pdf.

[19] Advanced Micro Devices. AMD Opteron™ Processor Product Data Sheet, 2006. Available from http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf.

[20] L. Dozio and P. Mantegazza. Real time distributed control systems using RTAI. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, page 11, Washington, DC, USA, 2003. IEEE Computer Society.

[21] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *DAC '07: Proc. of the 44th annual Design Automation Conference*, 2007.

[22] L.D.J. Eggermont, editor. *Embedded Systems Roadmap 2002*. STW Technology Foundation, March 2002. Available from http://www.stw.nl/Programmas/Progress/ESroadmap.htm.

[23] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *ECRTS '05: Proc. of the 17th Euromicro Conf. on Real-Time Systems*, pages 98–105, 2005.

[24] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a wcet-aware c compiler. In *ESTMED '06: Proc. of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 121–126, 2006.

[25] Free Software Foundation, Inc. GNU General Public License, version 2, June 1991. Available from http://www.gnu.org/licenses/gpl2.html.

[26] A. Gambier. real-time control system:a tutorial. In *Proceedings of the 5th Asian Control Conference*, Melbourne, Australia, July 2004.

[27] Doug Gibbs. Measuring treck tcp/ip performance using the xps locallink temac in an embedded processor system. www.xilinx.com/support/documentation/application_notes/xapp1043.pdf, 2008.

[28] R. Gioiosa. Asymmetric kernels for multiprocessor systems (in Italian), October 2002. Master thesis, University of Rome "Tor Vergata".

[29] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *The 4th IEEE International Symposium on Signal Processing and*

Information Technology (ISSPIT 2004), Rome, Italy, December 2004. Available from http://bravo.ce.uniroma2.it/home/gioiosa/pub/isspit04.pdf.

[30] Brian Hill. Xps ll tri-mode ethernet mac performance with monta vista linux. www.xilinx.com/support/documentation/application_notes/xapp1127.pdf, 2008.

[31] K. Hoyme and K. Driscoll. Safebus(tm). *IEEE Aerospace Electronics and Systems Magazine*, pages 34–39, Mar 1993.

[32] Tay-Yi Huang, Jane W. S. Liu, and Jen-Yao Chung. Allowing cycle-stealing direct memory access i/o concurrent with hard-real-time programs. In *Int. Conf. on Parallel and Distributed Systems*, Tokyo, 1996.

[33] Rick Jones. Netperf homepage. http://www.netperf.org.

[34] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the International Symposium of Code Generation and Optimization*, San Jose, CA, USA, Mar 2004.

[35] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability: Study of a network driver. In *Proc. of the* 13^{th} *IEEE Real Time Application Symposium*, Apr 2007.

[36] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[37] P. Marquet, E. Piel, J. Soula, and J.-L. Dekeyser. Implementation of ARTiS, an asymmetric real-time extension of SMP Linux. In *Sixth Real-Time Linux Workshop*, Singapore, November 2004.

[38] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel. Characterization of simultaneous multithreading (SMT) efficiency in POWER5. *IBM J. Res. Dev.*, 49(4/5):555–564, 2005.

[39] P. McKenney. SMP and embedded real-time. *Linux Journal*, 153, January 2007. Available from http://www.linuxjournal.com/article/9361.

[40] M. Momtchev and P. Marquet. An open operating system for intensive signal processing. Technical Report 2001-08, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille (France), 2001.

[41] M. Y. Nam, R. Pellizzoni, R. M. Bradford, and L. Sha. ASIIST: Application specific I/O integration support tool for real-time bus architecture designs. In *Proceedings of the IEEE ICECCS*, Potsdam, Germany, 2009.

[42] PCISIG. Conventional pci 3.0, pci-x 2.0 and pci-e 2.0 specifications. www.pcisig.com, 2009.

[43] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, and M. Caccamo. PRedictable Execution Model: Concept and implementation. Technical report, University of Illinois at Urbana-

Champaign, May 2010. https://netfiles.uiuc.edu/rpelliz2/www/index_files/techreps/PREMtechrep.pdf.

[44] R. Pellizzoni, B.D. Bui, M. Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Real-Time Systems Symposium, 2008*, pages 221–231, 30 2008-Dec. 3 2008.

[45] R. Pellizzoni and M. Caccamo. Toward the predictable integration of real-time cots based systems. In *Proc. of the 28th IEEE International Real-Time Systems Symposium*, pages 73–82, Washington, DC, USA, 2007.

[46] R. Pellizzoni and M. Caccamo. Impact of peripheral-processor interference on wcet analysis of real-time embedded systems. *IEEE Trans. on Computers*, 59(3):400–415, Mar 2010.

[47] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of Design, Automation and Test in Europe (DATE)*, Dresden, Germany, Mar 2010.

[48] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, and Marco Caccamo. "prem: The predictable execution model for cots-based real-time embedded systems". In *Proceedings of the 31th IEEE Real-Time Systems Symposium*, San Diego, CA, USA, December 2010. *[submitted]*.

[49] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for mem-

ory interference in multicore systems. In *Design, Automation and Test in Europe (DATE)*, Dresden, Germany, March 2010.

[50] PetaLogix. Petalinux. http://developer.petalogix.com/, 2008.

[51] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, November 10–16, 2003. Available from http://hpc.pnl.gov/people/fabrizio/papers/sc03_noise.pdf.

[52] John Pike. Hh-60g pave hawk. www.globalsecurity.org/military/systems/aircraft/hh-60g.htm, 2009.

[53] I. Puaut and D. Hardy. Predictable paging in real-time systems: A compiler approach. In *ECRTS '07: Proc. of the 19th Euromicro Conf. on Real-Time Systems*, pages 169–178, 2007.

[54] Wind River. VxWorks programmer guide, 2003. Available from http://www.windriver.com.

[55] J. Rosen, P. Eles A. Andrei, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of the* 28^{th} *IEEE Real-Time System Symposium*, December 2007.

[56] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th CODES+ISSS*, Oct 2008.

[57] S. Schoenberg. Impact of PCI-bus load on applications in a PC architecture. In *24th IEEE Real-Time Systems Symposium*, pages 430–439, Dec. 3–5 2003.

[58] S. Schönberg. Impact of pci-bus load on applications in a pc architecture. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, Dec 2003.

[59] Sebastian Schönberg. Impact of pci-bus load on applications in a pc architecture, 2003.

[60] L. Sha, T. Abdelzaher, K. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok. Real-time scheduling theory: A hystorical perspective. *Real-Time Systems*, 28(82648):101–155, 2004.

[61] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of Proceedings of the 23th IEEE Real-Time Systems Symposium*, Cancun, Mexico, Dec 2003.

[62] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1, July 1989.

[63] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1994.

[64] S.N. Storino, R.J. Eickemeyer, R.N. Kalla, and S.R. Kunkel. A commercial multithreaded RISC processor. *Digest of Papers,*

International Solid-state Circuits Conference, pages 236–237, 1998.

[65] J. M. Tendler, J. Steve Dodson, J. S. Fields Jr., Hung Le, and Balaram Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[66] Wikipedia the free encyclopedia. Commercial off-the-shelf. http://en.wikipedia.org/wiki/Commercial_off-the-shelf.

[67] Wikipedia the free encyclopedia. Cyber-Physical System. http://en.wikipedia.org/wiki/Cyber-physical_system.

[68] Wikipedia the free encyclopedia. Double data rate. http://en.wikipedia.org/wiki/Double_Data_Rate.

[69] Wikipedia the free encyclopedia. Intel Core 2. http://en.wikipedia.org/wiki/Intel_Core_2.

[70] Wikipedia the free encyclopedia. Multi-core processor. http://en.wikipedia.org/wiki/Multi-core_processor.

[71] Wikipedia the free encyclopedia. Xeon. http://en.wikipedia.org/wiki/Xeon.

[72] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM Press.

[73] J. Whitham and N. Audsley. Implementing time-predictable load and store operations. In *Proc. of the Intl. Conf. on Embedded Systems (EMSOFT)*, Grenoble, France, Oct 2009.

[74] Xilinx. Virtex-5 LXT FPGA ML505 Evaluation Platform. www.xilinx.com/products/devkits/HW-V5-ML505-UNI-G.htm, 2008.

[75] Xilinx. Microblaze soft processor core. www.xilinx.com/tools/microblaze.htm, 2009.

[76] Xilinx. Ml455. www.xilinx.com/support/documentation/ml455.htm, 2009.

[77] Xilinx. Ml507. www.xilinx.com/products/boards/ml507/docs.htm, 2009.

[78] Xilinx. Virtex-5 LXT ML555 FPGA Development Kit for PCI Express, PCI-X, and PCI Interfaces. www.xilinx.com/products/devkits/HW-V5-ML555-G.htm, 2009.

[79] K. Yaghmour. Adaptive domain environment for operating systems, 2001. Available from http://www.opersys.com/ftp/pub/Adeos/adeos.pdf.