A UNIFIED HARDWARE/SOFTWARE PRIORITY SCHEDULING MODEL
FOR GENERAL PURPOSE SYSTEMS

BY

Keith Alan Preston

Submitted to the graduate degree program in Electrical Engineering and Computer Science
and the Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Co-Chairperson: Dr. Arvin Agah

Co-Chairperson: Dr. David Andrews

Dr. Douglas Niehaus

Dr. Andrew Gill

Dr. Sara Wilson

Date defended: Monday, October 17th, 2011

The Dissertation Committee for Keith Alan Preston certifies
that this is the approved version of the following dissertation:

A UNIFIED HARDWARE SOFTWARE PRIORITY SCHEDULING MODEL
FOR GENERAL PURPOSE SYSTEMS

Committee:

Co-Chairperson: Dr. Arvin Agah

Co-Chairperson: Dr. David Andrews

Dr. Douglas Niehaus

Dr. Andrew Gill

Dr. Sara Wilson

Date approved:

Abstract

Migrating functionality from software to hardware has historically held the promise of enhancing performance through exploiting the inherent parallel nature of hardware. Many early exploratory efforts in repartitioning traditional software based services into hardware were hampered by expensive ASIC development costs. Recent advancements in FPGA technology have made it more economically feasible to explore migrating functionality across the hardware/software boundary. The flexibility of the FPGA fabric and availability of configurable soft IP components has opened the potential to rapidly and economically investigate different hardware/software partitions. Within the real time operating systems community, there has been continued interest in applying hardware/software co-design approaches to address scheduling issues such as latency and jitter. Many hardware based approaches have been reported to reduce the latency of computing the scheduling decision function itself. However continued adherence to classic scheduler invocation mechanisms can still allow variable latencies to creep into the time taken to make the scheduling decision, and ultimately into application timelines. This dissertation explores how hardware/software co-design can be applied past the scheduling decision itself to also reduce the non-predictable delays associated with interrupts and timers. By expanding the window of hardware/software co-design to these invocation mechanisms, we seek to understand if the jitter introduced by classical hardware/software partitionings can be removed from the timelines of critical real time user processes. This dissertation makes a case for resetting the classic boundaries of software thread level scheduling, software timers, hardware timers and interrupts. We show that reworking the boundaries of the scheduling invocation mechanisms helps to rectify the current imbalance of traditional hardware invocation mechanisms (timers and interrupts) and software scheduling policy (operating system scheduler). We re-factor these mechanisms into a unified hardware software priority scheduling model to facilitate

improvements in performance, timeliness and determinism in all domains of computing. This dissertation demonstrates and prototypes the creation of a new framework that effects this basic policy change. The advantage of this approach lies within its ability to unify, simplify and allow for more control within the operating systems scheduling policy.

Contents

List of Figures

List of Tables

Terms

AB Apache Benchmark

AP Application Process

API Application Programming Interface

ASIC Application Specific Integrated Circuit

BRAM Block Random Access Memory

BSP Board Support Package

CBIS CPU-Bypass Interrupt Scheduler

CFS Completely Fair Scheduler

CPU Central Processing Unit

DDoS Distributed Denial of Service

DoS Denial of Service

DSP Digital Signal Processor

FPGA Field Programmable Gate Array

FSM Finite State Machine

GB Gigabyte

GPOS General Purpose Operating System

GPU Graphics Processing Unit

HW Hardware

HWTI Hardware Thread Interface

IBM International Business Machines Corporation

ID Identifier

I/O Input and Output

IPIC Intellectual Property Interconnect

IRQ Interrupt Request

ISR Interrupt Service Routine

IST Interrupt Service Thread

KB Kilobyte

KURT Kansas University Real Time Linux

LIFO Last In First Out

LITMUSRT Linux Testbed for Multiprocessor Scheduling in Real-Time systems

MB Megabyte

MHZ Megahertz

MIPS Millions of Instructions Per Second

MP3 Moving Picture Experts Group Layer 3 Codec

MPSOC MultiProcessor Systems On Chip

NAPI New API

OPB On-Chip Peripheral Bus

OS Operating System

O(1) Of Order 1

O(N) Of Order N

O(log N) Of Order Logarithm of N

PC Personal Computer

PCT Processor Control Threads

PIC Programmable Interrupt Controller

POSIX Portable Operating System Interface for Unix

SW Software

SRAM Static Random Access Memory

TCP Transmission Control Protocol

RT Real-time

RTLinux Real-time Linux

RTM Real-time Task Manager

RTOS Real-time Operating System

USB Universal Serial Bus

VGA Video Graphics Accelerator

VHDL Very High-level Design Language

Chapter 1

Introduction

1.1 Operating Systems: A New Level of Portability

Operating systems are arguably the most fundamental system software component within modern computing systems. A fundamental role of the operating system is to abstract and manage the underlying hardware resources of the system [3]. Operating systems abstract the underlying system hardware resources through the formation of a virtual machine. Client processes interact with the virtual machine through a set of Application Programming Interfaces (APIs). These APIs allow access to the hardware resources.

The role of the operating system as virtual machine traces its lineage back to the pioneering work of the IBM OS/360 [4][5]. The OS/360 was a batch processing operating system made for IBM's System/360 Mainframe computer announced in 1964. IBM's system software designers sought to enable programmers to easily port both legacy and new application code across an emerging family of new machine architectures. This early work freed programmers from having to rework source code to match each different machine's low level platform specific hardware organization. Current operating systems such as Linux and Windows have evolved from this early work to bring the same benefits of portability to all users. The development of operating systems

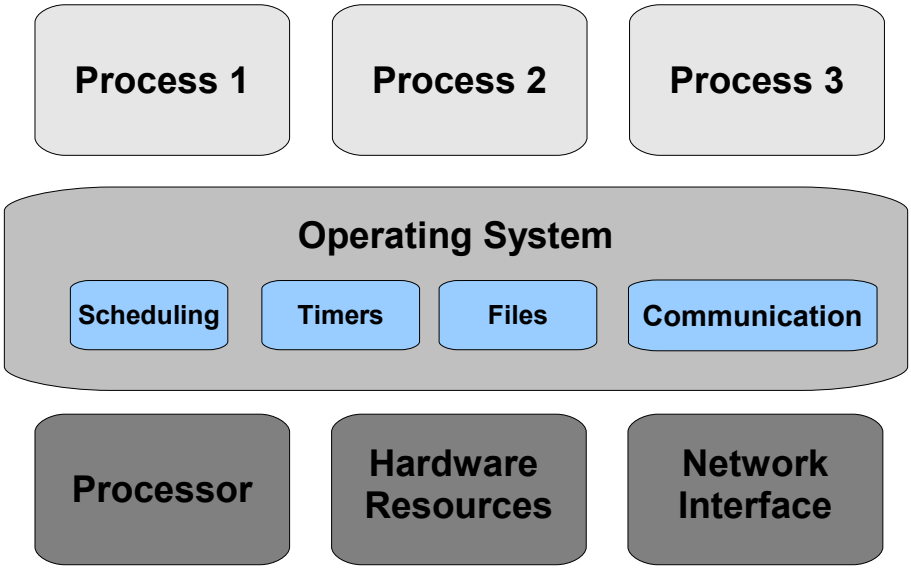

Figure 1.1: Operating System Abstractions

was important because operating systems brought portability, higher level functionality and decreased design time to all types of systems.

1.2 Overhead: The Cost of Portability

As a general rule, the benefits of portability and abstractions come at a tradeoff of decreased performance[6]. Operating systems are not immune from this tradeoff. As executable programs, they require processing time that takes away from cycles available for executing the application program. Thus operating system execution time is generally referred to as overhead. The actual overhead incurred by any operating system can vary based on changing system loads, and is also dependent on a machines specific hardware organization.

For a large family of general purpose computing applications, the overhead and variance introduced by the operating system are an acceptable tradeoff when compared to the benefits provided through abstraction and reuse [7]. However, for certain classes of applications, such as

those found in real time systems, the overhead and variance negatively affect the overall quality of service that can be provided by the system. Operating systems researchers of embedded and real time systems have historically sought to deliver operating systems that merge the benefits of portability and abstraction associated with general purpose operating systems, but with reduced latency and minimal jitter necessary for meeting real time requirements [8].

One of the primary resources that operating systems manage is the CPU itself. In today's complex systems, there can often be hundreds of processes competing for time on the CPU. To manage time sharing of processes on the CPU, the operating system runs a scheduling algorithm. Significant research over the last 30 years has been devoted to developing fair and appropriate scheduling algorithms. The scheduler as a program requires CPU cycles to run its scheduling algorithm. Thus, the act of running the scheduler itself can result in performance degradations for the application programs. To minimize this effect user-level thread scheduling frameworks [9] have been developed to allow groups of threads to schedule themselves. These frameworks allow the use of special scheduling algorithms that are optimized for known thread requirements and interactions of the program rather than relying on a one size fits all approach to scheduling.

Others have worked on making the operating system scheduler as fast as possible. Ingo Molnar created a new scheduler in Linux[10], termed the O(1) scheduler, to make scheduling decisions in constant time. The name O(1) can be misleading as the scheduler still has O(n) timings for certain actions such as interrupt processing. However, the most frequent decision of the scheduler, what thread should run next, is constant time, or O(1), with respect to the number of threads in the system. This capability is important for real time systems that require very precise timing of the scheduled applications. Any jitter introduced is essentially unwanted variation in timing for a periodic real time thread [11]. To further lessen the impact of the scheduler, later versions of Linux replaced the O(1) scheduler with the CFS, Completely Fair Scheduler[12]. This scheduler uses a more complex data structure, a red-black tree, to shorten more scheduling operations from O(N) to O(log n).

1.3 Invocation Mechanisms: A Lasting Hardware Legacy

Even though the O(1) scheduler reduces the time taken to run the software scheduling algorithm, additional overhead and variance can still be introduced when the scheduler is invoked using historical invocation mechanisms. As an example, countdown timers are a familiar trigger for invoking a scheduling decision in Linux. In a classic configuration, the timer computes a periodic interrupt that represents a jiffy. During each jiffy interrupt, timers are checked and expired timers are executed[13]. This introduction of time keeping as a service of the operating system causes an additional overhead and variance because of a periodic interrupt.

In addition, interrupts are by definition asynchronous invocation mechanisms, and can have fairly high processing overhead in modern computers[14]. A fundamental problem arises when using interrupts as invocation mechanisms when very precise scheduling is required. The interrupts stop the CPU before determining if a scheduling decision is required. Since some real time threads could be a higher priority than interrupts, stopping the CPU to make a decision causes jitter in the system.

There have been various software approaches used to explore how to lessen the overhead of servicing interrupts. RTLinux[15] attempts to reduce the overhead of interrupts by creating an additional register that allows the system software to quickly determine if the interrupt is attempting to invoke real time or non-real time processing. The PREEMPT_RT patch[16] for the Linux kernel takes another software approach. This approach queues the interrupt in thread context quickly, returning to a new scheduling decision, and possibly the previous task. These software approaches attempt to minimize the overhead and jitter of checking the interrupt, but they cannot fully eliminate it.

Scheduling policy is actually a combination of two scheduling mechanisms. This is true for most types of computing policies. The policies are first dictated by the design of the hardware they run on, and second, by the design of the software built to run on top of the hardware[17]. In

scheduling policy, design is first dictated by the age old concept of hardware interrupts, and secondly, by the software-based operating system scheduler. The design of the hardware can often limit what is possible with software.

1.4 Hardware Software Co-design: An Answer?

This overhead and jitter caused by interrupts and the operating system itself take away processing time and add timing uncertainty into the applications[18]. To eliminate this type of overhead and jitter altogether cannot be achieved through software methods along, and requires modifications to the underlying hardware micro-architecture. While researchers have long been aware of the effects of inefficient invocation mechanisms, and overhead of running the scheduler, the cost of refactoring these functions into special purpose hardware have been prohibitive. Chip manufacturers are understandably reluctant to change baseline hardware designs, as these changes would have an unacceptably large economic impact on system and application software. Creating additional application-specific integrated circuit (ASIC) accelerators can also address the problem, but they generally have non-recurring engineering costs in the millions of dollars range [19] [20].

Recently, modern FPGAs have emerged that contain significant gate densities, diffused components such as processors, and distributed SRAM based memories to form complete multiprocessor systems on chip (MPSoPC) architectures[21]. Many embedded systems designers are turning towards FPGAs to replace more expensive ASICS. FPGAs also represent convenient testbeds for exploring migration of functionality across the software/hardware boundary[22]. Thus FPGA's represent convenient sandboxes that can be used to explore different hardware software partitionings in a quick and cost effective way. The growing adoption of FPGA's as hardware infrastructure within embedded systems then makes any exploratory work performed on an FPGA available for use within these fielded systems. The Hybridthread system[23] is a great

example of this exploration in embedded systems. Through the use of FPGAs the Hybridthreads project has reimagined what an operating system looks like by transitioning many functions into hardware, such as Scheduling, Mutexes and Condition Variables. This dissertation uses an FPGA and the basis of Hybridthreads to explore and propose a new hardware/software repartitioning of the scheduling algorithm, and the invocation mechanisms to enhance the traditional scheduling goals of efficiency, timeliness, and predictability.

We perform our explorations under the constraint of not degrading the benefits associated with portability and software reuse. These benefits have been at historical odds with the timing constraints under which application designers typically must operate within the fields of embedded and real time systems. We believe a new framework can be designed that achieves both attributes by working within the framework of Linux. Our approach is to not alter or change the policies or API interfaces of Linux, but rather the mechanism used to invoke them. This allows our platform to enable desirable real-time capabilities into the general purpose operating system.

To validate the findings in this work, a wide range of tests have been run to evaluate different performance characteristics of the new scheduling framework. The benchmarks have been selected to show the effects on scheduling overhead, multi threaded performance, timing accuracy, interrupt latency, interrupt control, and real time performance. These tests include standard Linux benchmarks and programs such as Hackbench[24] and Apache[25] web server.

1.5 Dissertation Contributions

The contributions of this dissertation are:

- Integrating interrupts into the HybridThreads[26] [27] operating system scheduler
- Porting the HybridThreads operating system kernel to the Linux Platform
- Redesigning the interrupt and scheduler from the HybridThreads operation system into a

unified scheduler to maximize results for lowest hardware cost

- Demonstrating and evaluating the effects of the unified model on common domain specific use cases to previous approaches

Chapter 2

Background

Operating systems continue to be a important and popular area of research due to their impact on system performance. Recently the switch from scalar to manycore systems has reinvigorated research on how the operating system should be restructured to better support scalable numbers of heterogeneous processor resources. Central to these efforts are how the new requirements of heterogeneous manycores will redefine the role and structure of the scheduler. The historical objective of scheduling threads and processes on a single shared CPU was to increase the overall utilization of the CPU. The switch to heterogeneous manycores is now causing researchers to modify this objective function to minimize application latencies and not simply maximize the utilization of a single shared resource. This is moving the scheduler towards the roll of a dispatcher and bringing in scheduling latency issues that have been a driving scheduling research within the discipline of embedded and real time systems . While traditionally limited to only specialized embedded systems, real time functionality is now been sought in all domains of computing.

There are many different ways to define a real-time system. A popular definition is “A real-time system is one in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are generated”[28]. Generally

there are two separate categories of real time, hard real time and soft real time. Hard real time is when a missed deadline results in catastrophic failure, like the loss of life. Soft real time is when missed deadlines generally result in bad user experience, such as skipping frames in multimedia video. Operating systems are striving to be better equipped to handle real time tasks by improving real time operating system performance metrics that include task switch time, interrupt latency, preemption time, system call processing time, and overall run-time efficiency[29].

Another reoccurring theme in operating system research is the need for flexibility in design and implementation of operating systems. A research operating system, K42 [30], was made to show this need. This research operating system is designed for flexibility and monitoring. Its subsystems are replaceable, and the core system has the monitoring infrastructure to rapidly evaluate new research ideas. It has shown the fault of today's operating system in their utilization of nonflexible techniques like global data structures and global policies. K42 is only one of many other research operating systems, like Exokernel [31], Spin [32], Vino [33] which illustrate similar weaknesses in current operating system approaches.

Flexible component based operating systems generally start with a very small, fast and efficient core, and expand on this through additional components. The SPACE project [34] takes this approach in a new operating system abstraction, and is identified as a sub-micro kernel. It takes scheduling out of the base kernel, aside from a very simple priority mapping of processes groups, and lets an application decide its interaction inside the process group. This can be beneficial from an application stand point as it can define the interaction between its threads. Priority, and other scheduling models, may not extract maximum performance from these applications.

A lot of the need for flexibility in an operating system comes from the additional overhead incurred by every additional component. This has become apparent in super computer designs. Recent studies[35] have even shown that operating system overhead on supercomputing clusters affect performance so much that if a system is designed to dedicate and isolate operating system

overhead, they can often achieve higher performance. This is done by designating one processor of a multiprocessor node to not perform normal computation and only handle operating system tasks, rather then trying to distribute the task to node running the application.

Another group, lead by Jean-Charles Tournier[36], has discussed the flexibility of an a-la-carte operating system, or an operating system that can be dynamically built with the services needed by the system. This way, systems such as supercomputers, the example in the paper, have a more flexible model to eliminate costly scheduling algorithms, software timers, time slicing or other unneeded components. They discuss Puma[37], a lightweight operating system for massively parallel systems developed at Sandia National Laboratories. Puma is a small operating system consisting of the quintessential kernel (Q-Kernel), the process control thread (PCT) and the application processes(AP). This design allows for adaptation and elimination of the PCT, or scheduling thread, if the system calls for it.

In a newer approach to supercomputing, the BEE2 project[38] has looked at a massively temporal and spatial parallel system consisting of FPGAs. They have discussed many problems from their concept of stream-based computing. The most notable is that the high bandwidth nature of data through FPGAs causes hard real-time requirements into operating systems and applications. Without real-time processing, the CPU backs ups requests and must pause the FPGA from processing to catch up. Most of these real time requirements come from message passing. The predictability of operating systems highly affect message passing performance, as the slowest node of a computational cluster will cause others to idle to its speed during synchronization. To avoid problems they have even chosen to only run the Linux kernel on one node per board of five FPGAS. The other FPGAs run a micro kernel that is a slave to the Linux FPGA to help better keep real-time requirements.

Another example of the effect of scheduling overhead is the HPC-Colony project[39], a collaboration between Lawrence Livermore National Laboratory, the University of Illinois and IBM which is funded by the Department of Energys Office of Science under the FastOS

program[40]. This project is focused on services and interfaces for systems with 100,000+ processors. The project clearly defines problems in supercomputing domains with respect to operating systems. They point out that Operating System processes and scheduling are much too coarse-grained to allow accurate load balancing. In fact, the time wasted by the scheduler in deciding the load balance often negates the benefit of load balancing.

In the same manner that super computing application exposes scheduling issues, multi threading has many issues with respect to the scheduler[41]. Unfortunately, these are exceedingly hard to quantize because of non-determinism of multi threaded systems. Projects, such as LITMUSRT (Linux Testbed for multiprocessor Scheduling in Real-Time systems)[42], have developed entire systems just to try and predictably measure this non-determinism. They establish that the four sources of overhead of relevance are task preemption, migration costs, context switching and scheduling overhead. They specifically leave out interrupts, going to lengths to try and avoid them and other background activities in their system. They do this by booting into single user mode to start only a minimal set of tasks. They establish in multiprocessor systems that migration costs between processors have the highest impact in terms of performance. Preemption and scheduling overhead are identified as sources for easy improvement because they cause performance issues when scaling algorithms from tens to thousands of cores.

With respect to real-time systems, most traditionalists have ignored multi threading and multiprocessor systems due to the lack of predictability inherent in the asynchronous models. Some use multiprocessor systems, but limit the real-time applications to run on an isolated CPU[43]. Redhawk Linux[44] is another example that uses this shielded processor approach to achieve sub 1 millisecond response times on Linux. This helps to deal with the scalability issues on scheduling algorithms on significant number of processor[45]. Recent work on this area[41] suggests that solutions to this problem involve a move to deterministic switching and scheduling, allowing multiple real-time tasks to complete deterministically, if deadlines have been proved feasible.

Most of this research points toward the idea of a low latency scheduler. Most future direction of scheduling based research is summed up well by a group at Princeton University[46]. They reiterate the idea of a new hardware/software contract. Software is to expose as much parallelism through threads, and hardware is suppose to provide facilities to provide low latency scheduling so as to not hinder multi threaded software. It is expressed as a chicken and egg problem. Due to the overhead of multi threading, software developers often do not develop multi threaded software because it can often be slower than single threaded software. This is generally true only when the number of threads is greater than the number of cores available. In the same way, hardware designers are reluctant to design high numbers of cores into processors because software cannot yet use them efficiently.

Most experts agree that to properly make a low latency scheduler, hardware must be utilized[47]. Migrating operating system functionality into hardware is not a new approach used in the world of computing. Virtual Memory is a great example of another subsystem that has a hardware component. This is generally due to the high cost of a software implementation. Virtual memory in software requires twice the amount of memory accesses. One memory access is to look up the translation from virtual to physical, and the additional access is to read the physical memory. However, with a processor memory unit, this cost is greatly reduced when a cache hit is made. Performance is not the only benefit of hardware. Power issues, a key in the embedded market, has given support for operating system hardware development. As embedded devices strive for power efficiency, they look to the obvious advantage of hardware over software in this realm. Elais T. Silva Jr et al. [48] show many reasons for the case of middleware services in hardware. They have developed a hardware core for Task Scheduling and Communication on top of the Java Real Time Specification and the FemtoJava Processor. They find that a hardware scheduler gives a more fine grain control to throttling processors, and can significantly increase energy savings and battery life.

2.1 Timers

Research in scheduling affects other areas, especially the area of timers. This is because the design of most general purpose operating systems use periodic timers to regain control of the system and measure the passage of time. This is quickly being determined as inefficient by many recent efforts because this is a use of timers to do synchronous polling, which is known to be inefficient as compared to asynchronous interrupts.

Many describe this as the failure of the timing tick[49]. The tick fails mostly due to the resolution of the timer. For example Unix 6 in 1976 used a tick rate of 60 hz[50]. While this has increased slightly, it is common for operating systems to have a 100 hz timer frequency. This has been nearly unchanged for decades. This leads to failure in multiple arenas[51]. In the embedded environment, power consumption has been tied to periodic timers. Every tick in the system must wake up and waste power while taking care of operating system components dependent on the tick. The solutions for the failure of the timer tick all point to eliminating this polling at an operating system level and pushing it down to the hardware. This is done with one-shot timers, although some operating system components need to be redesigned to work with them. These timers along with other operating system overhead eliminate help to reduce System Noise[52] to a minimum.

2.2 Interrupts

Interrupts, since their invention, have been both a blessing and a curse to real-time designers. Although originally designed to facilitate real-time functionally by reducing latency through their asynchronous nature, they have inherent drawbacks from a software point of view[53]. They are generally non-portable across compilers and hardware platforms. Interrupts tend to expose race conditions in software due to unpredictability. There is a need to design interrupts and interrupt software in a safe and structured way. As more real time tasks move into software, even more care

is needed to interact with interrupts. One manifestation of interrupt problems is interrupt overload[54]. Newer hardware interfaces, such as the Universal Serial Bus(USB)and Gigabit Ethernet, have high requirements on maximum interrupt frequency. Respectively, they can generate on the order of 100,000 and 1,000,000 interrupts a second, and can pose significant challenges for real-time systems[55]. This can cause many system problems, including the highly publicized denial of service attacks. These attacks take form when enough persistent network traffic causes millions of interrupts, exposing faults in how the operating system deals with scheduling these interrupts, and processing network data. Some work has been done to mitigate these effects in software. The Linux kernel has added the NAPI ("New API")[56] as a device packet processing framework that can switch drivers into polling mode during periods of high traffic.

Care must, also, be taken in the testing of interrupts. Tests designed to test interrupts should be random in nature[57]. They should be done with random uncorrelated events and stress the system to help establish failure events. Current interrupt semantics challenge predictability and timeliness by allowing asynchronous interrupts to temporarily stop the execution of the application program. This, in return, introduces additional delays in terms of costly exception processing and context switching. The performance degradation due to context switches will continue to grow as next generation CPUs, with ever deeper pipelines and hierarchical caches, introduce more states that must be saved during a context switch. Additionally, the interrupt requests are serviced outside the system scheduler, and are not deterministically considered with threads and processes within the ready-to-run scheduler queue.

There are three approaches to minimizing this effect. An operating system can process the whole interrupt every time it is stopped. A simple approach, but causing lots of jitter to threads at the advantage of short interrupt latency.

Most research in real-time operating systems (RTOSs) attempt to minimize the overhead of exception processing and reduce the nondeterministic jitter by only executing a small portion of

the interrupt service, routing the ”top half” of the routine. This top half typically saves data to a buffer and schedules the ”bottom half” to process the data. This approach is a balance of thread latency versus interrupt latency.

Lastly, the PREEMPT_RT patch for Linux and other efforts [58] [59] simply marks an interrupt as an enqueue thread operation and continues. Unfortunately, this approach must allow the asynchronous request to occur and suffer the overhead and jitter of marking the interrupt as pending. Due to the adherence of the current interrupt semantics, research in pure software-based techniques has reached a plateau. This approach sacrifices interrupt latency significantly in efforts to lower thread latency. The latency of ISR can become 3-10 times higher even if parts of the ISR processing are outsourced to an external co-processor[60].

Research efforts like the SLOTH kernel[61] [62] take the opposite approach. Rather than moving interrupts into thread context, they have moved threads into interrupt context. This avoids interrupt latency, the main penalty of moving interrupts into thread. In fact, it shrink overall latency considerably, considering the system scheduler becomes a multiple priority level interrupt controller. It does have a few major drawbacks in that it is limited to the number of threads by the interrupt levels in a system, and all threads must have a run to completion semantics without blocking.

2.3 Linux

This plateau of software schedulers is now commonly found in general purpose schedulers. A great example of a general purpose scheduler can be seen in the Linux operating system. Linux was originally created by Linus Torvalds for use on his personal computer. It has since become a community open source effort, and today reflects a lot of the widely accepted ideas about what an operating system should be. It, currently, has made much progress in both the server and embedded market, along with the original desktop market for which it was written. Its scheduler

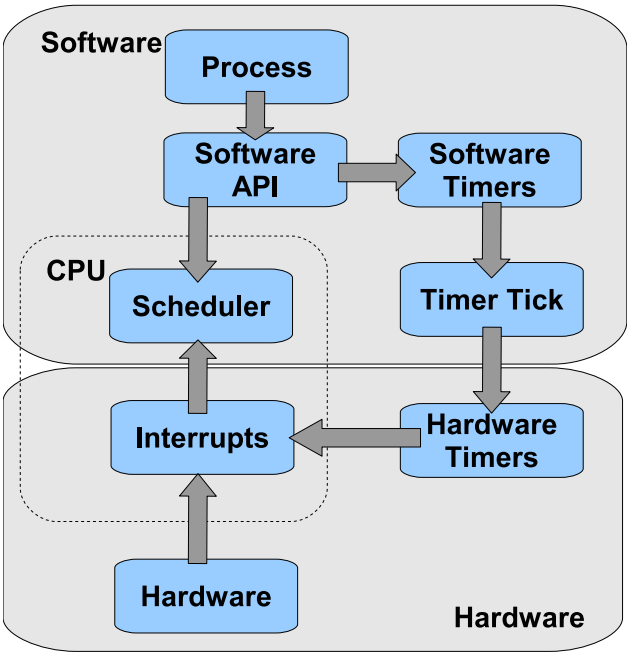

Figure 2.1: Traditional Scheduler Model

has many artifacts of the original hardware interrupt semantics, most notably the introduction of softirqs and tasklets. Softirqs and tasklets are typically used in the traditional top and bottom half interrupt paradigm. While the initial processing of an IRQ is in a dedicated response handler, the majority of the work is typically performed at a later time in a softirq and tasklet. This has the advantage of reenabling interrupts and even scaling softirqs or tasklets across multiple CPUs for the majority of the work. However, this has shown how software has been used to redefine legacy scheduling policy.

With speed and efficiency increases in computer technology, computers are doing even more work. However this makes scheduling the work an even more difficult task. This is shown by the ever evolving nature of the Linux scheduler. There has been much improvement recently, lead by Ingo Molnar. Ingo Molnar [10] started by identifying six sources of long latencies in the Linux kernel. He has established patches that have been incorporated into the mainstream kernel to fix problems. The problems Ingo identified include calls to the disk buffer cache, memory page management, /proc file system, VGA subsystem, large processes and the keyboard driver. His

work has been adopted and methods extended through other patches relating to timing and scheduling, such as the UTIME patch, developed at the University of Kansas[63]. UTIME adds higher resolution timers into Linux. This is supplemented by many other projects, such as Rapid Reaction Linux[64], which further improves upon these results through precalculating delays in processing. Ingo Molnar makes a permanent mark on the Linux kernel history with his O(1) scheduler introduced in the 2.6 version of the kernel. This scheduler accomplishes constant time operations through a trade off of memory for speed. This change from v2.4 kernel has increased the schedulers ability to handle a greater number of processes without performance degradation.

Much of this timing improvement has evolved into real time distributions of Linux like KU Real Time Linux (KURT), RTLinux [65] and Timesys Linux [66]. These projects have accomplished progress in speeding up real time[15] performance. To give better control to application developers, an extension known as the Real Time Application Interface(RTAI)[67], was made as a real-time extension to the linux kernel. These new APIs allow real-time applications to run with strict timing constraints. These APIs have been utilized in many real time systems involving motion control [68].

The KURT project has had further expansion into hardware through a hardware scheduler module as seen through a recent thesis[69]. Feasibility of a hardware scheduler with the Linux kernel is shown. Performance is not significantly improved in its design, although a distinct improvement in worst-case scenario time is shown with the hardware scheduler.

Efforts in Linux have continued into new ideas and these continue to evolve projects like HRTimers[70]. HRTimers further develop multiple timer systems that respond to the accuracy needs of different applications. Not all timers are created equal. Some are just timeouts for networking, and device drivers that rarely occur and do not need high resolution. Others are used to schedule more important events, and need more accurate precision. By creating multiple systems with different resolutions, overhead versus accuracy can be balanced.

More recently, the PREEMPT_RT patch[71] for the Linux Kernel has been the widest used

kernel for Linux real time systems. The PREEMPT_RT is a combination of many previous ideas of lower latency and preemptable kernel patches[72]. The biggest idea behind the PREEMPT_RT patch is to minimize the amount of kernel code that is non-preemptable. This results in priority inheritance for many in-kernel spinlocks and semaphores to allow preemption. Many other critical sections are also shortened, deferred or made preemptable. In addition, interrupts become preemptable and can run in process context, allowing scheduling on the same priority level as threads. The PREEMPT_RT patch is one of the best approximations in comparison to some hardware schedulers. It generally represents a plateau of what is possible in minimizing interrupt jitter[73]. While it does not completely eliminate interrupt jitter because hardware interrupts still cause the CPU to stop, it does the bare minimum to schedule the thread that services the interrupt. After this, it goes back to the highest priority process.

Despite these improvements, the Linux scheduler still adds overhead to the system. This project focuses on moving key aspects of the scheduler into hardware, primarily the run queue and the decision of what should be scheduled next. This improvement allows the CPU time consumed by the operating system scheduler to be reduced and made available to other processes.

2.4 Hardware Schedulers

Hardware scheduling has also been a popular research topic in the areas of real time and embedded systems. With well co-designed hardware cores, the minimum level of jitter and overhead can be significantly reduced [74][75][76]. However, despite this effort, most of the work has not made an impact on mainstream computing. The questions of why and what can be improved must be asked of these projects.

There are many different parts of scheduling to try to improve with hardware. Invocation methods, scheduling algorithms and context switching make up the majority of time when doing scheduling. While most historical hardware schedulers have focused on moving the scheduling

algorithm to hardware, this research takes a different approach in that the focus is on all of the invocation mechanisms. There has been little research into context switching, because it generally needs a major architectural change in processors. Most of the original hardware scheduling units involve complex hardware sorters to compare thread priority. Originally, work is done to minimize sorting size in hardware but to keep the speed in insertion and extraction from the hardware sorting mechanism[77]. A few examples of the first hardware scheduling systems follow.

The Spring Kernel and Scheduling Co-Processor[78] is a project from the University of Massachusetts seeking to enhance real time scheduling. The Spring Kernel is designed for meeting hard real-time deadlines. The process sends scheduling requests to the kernel which can only be accepted for execution if the deadline constraints are possible. From this work, they design a scheduling co-processor showing significant improvements to scheduling. They even recommend incorporation of this scheduling co-processor into general purpose processors, calling it analogous to special purpose hardware for branch prediction[79]. This project starts a base line for future endeavors. As hardware design techniques and technologies improve it gives way to easier and larger hardware schedulers.

Sergio Saez of the Universidad Politecnice de Valencia proposes a circuit solution of a sophisticated interrupt controller used as a scheduler[80]. It combines systolic arrays to achieve constant time decisions. They propose a few advancements to real time deadlines. One advancement is keeping deadline information in the hardware and looking for slack between deadlines. This slack can be used to run lower priority processes with early deadlines. His designs show the advantages and disadvantages of hardware schedulers with lots of process state information. Although deadline information allows better hard real-time performance, the size and complexity trade off makes it unsuitable for general purpose systems.

A group from Mlardalens University has created their ASIC hardware scheduler[81]. It can handle up to 64 tasks at eight priority levels that can be mapped on up to three CPUs. Most of the

communication between the processor and scheduler are done through interrupts and 4-6 bus accesses to registers. It introduces new ideas on multiple CPU systems. However, its register system has issues with atomic operations and extra care has to be given to this design. This project shows how a hardware scheduler can support a tickless kernel. It determines a time model for comparison of real-time systems and explores new hardware accelerated operating system functions like semaphores.

Paul Kohout with the University of Maryland[47] has also designed and simulated a Real time Task Manager (RTM) that takes the most common RTOS operations, like task scheduling, time management and event management. It implements static priority scheduling for up to 64 threads. His design differs from existing approaches, by taking a more minimalistic design. He has a status and a timer register for each priority level. No other information is kept about priority levels. In simulation, he shows results of 80-90% speed up in common scheduling tasks, even though the hardware design is also very small. However, the system is lacking in that it is not designed to accommodate interrupts. Although never actually implemented, this design characterizes a base for this dissertation. The design is being expanded to a larger and more general system while being implemented.

Early implementation of hardware schedulers focuses on a limited scope system that is as fast as possible [75]. It usually includes a limited amount of threads and utilizes a great number of hardware resources to achieve this goal. Generally these implementations require additional hardware resources for each additional thread in the system because of the use of systolic arrays[82]. With significant hardware resources needed, much planning is needed to implement this type of system. This leans away from the idea of a general purpose system. This thesis seeks to minimize hardware resources, only using hardware when there is a specific advantage to it. After these initial designs, complex hardware scheduling designs evolve that begin to involve interrupts and timers. A group from Georgia Tech[83] takes a look at moving the scheduler and the periodic timer tick into hardware with interrupts at eight priority levels. Their scheduler is

flexible and can be configured to three algorithms: Priority, Earliest Deadline First and Rate Monotonic. This is an advantage provided by their hardware platform of FPGAs. The speed development and dynamic programmability of FPGA allow their scheduler to be reprogrammed to the best algorithm. Their synthesis hardware supports up to 16 tasks and eight external interrupt sources. It is fairly extensive hardware and supports queue operations in hardware. This advanced functionality has not been seen in previous generation projects due to lack of hardware resources. This work improves hardware schedulers to a more modern scheduler design with run queues and sleep queues. This group's work shows significant improvement over earlier designs and has set the foundation for future works.

More recently, a project called Hybridthreads[23] takes a more general approach to the problem. The Hybridthreads project originated at the University of Kansas and has recently migrated to the University of Arkansas. The Hybridthreads group created a hardware scheduler and operating system that abstracted the notion of thread. This thread can be either a hardware or software thread and is treated the same through the operating system. Hybridthreads are a unique approach to hardware scheduling as it is geared toward FPGA hybrid hardware software systems. Its hardware design is also unique in it is similar to the O(1) linux schedulers. It has a queue for each priority level and is extremely fast and efficient in scheduling operations. The Hybridthreads system has expanded into semaphores, timers and interrupts to support its hardware scheduler, and extends its new hardware/software neutral threaded computational model.

This thesis is a branch off of the Hybridthreads project at the University of Kansas. One of many projects taking the concepts of Hybridthreads and extending them [84]. The lessons learned from creating a efficient, timely, and predictable hardware operating system in a hybrid environment of hardware and software are being applied towards more general systems. More specifically, Hybridthreads is ported to a mainstream platform, Linux. The knowledge learned from this is used to redesign for software threads, removing the ability to support hardware threads. Although new FPGA computational models are being developed and advancing quickly,

their affect on general computing in the near future will probably be limited, hence, the redesign of a scheduler to focus on software threads. The goal of this dissertation is to show that the inclusion of a lightweight hardware scheduling core into the next generation of processor is merited due to the increase in performance and response of all types of systems.

Chapter 3

Solution

3.1 Problem Statement

Question: How can the knowledge gained from real-time hardware scheduler research be used to improve a general-purpose operating system?

Hypothesis: Unifying scheduling policy by merging the mechanisms of hardware interrupts, timers, and a software scheduler through creation of new hardware will facilitate improvements to scheduling policy in efficiency, timeliness and predictability, while maintaining a full featured general purpose scheduler.

3.2 Hybridthreads

Microkernels are changing the design of operating systems, however FPGAs are making it easier to change the hardware software boundary in today's operating systems. FPGAs do this by allowing quick turn-around between hardware design and implementation. The logic fabric of an FPGA can be reprogrammed in seconds, although typically taking a few hours to synthesize the logic gates from hardware description languages. This new hardware testing paradigm has led to

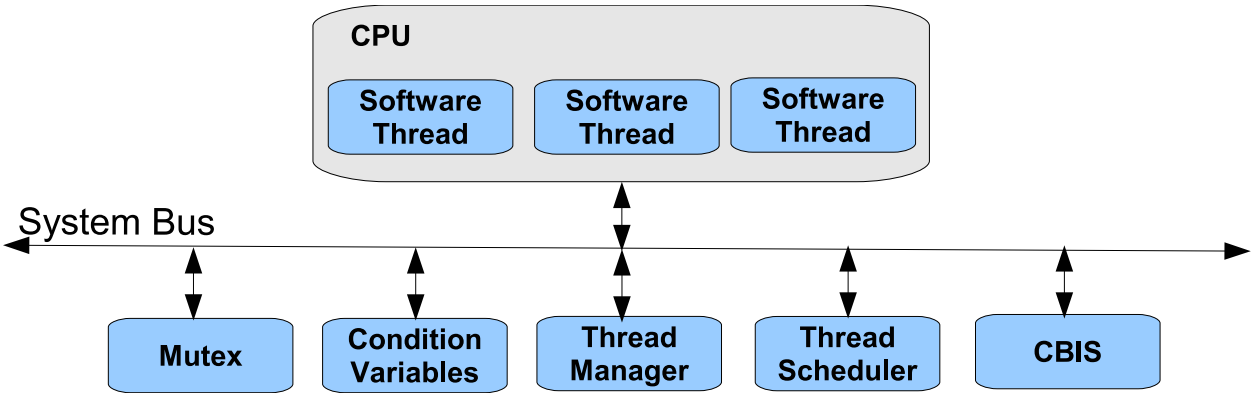

Figure 3.1: Hybridthreads Design

much experimentation in the hardware software boundary for a kernel. The Hybridthreads microkernel from the University of Kansas[85] is a development of this new process. Through experimentation with the hardware software boundary of operating system, the Hybridthreads kernel was developed as an FPGA operating system that maximizes the parallel nature of its specialized hardware circuit to execute common operating system tasks.

Each of these common system tasks are completed in an independent hardware core. The cores independent design allow them to act in parallel. The hardware cores can communicate with each other without processor intervention, enabling a variety of improvements. These improvements include faster and more predictable responses from operating system service to advanced interaction with specialized hardware cores[85]. The Hybridthreads system has improved performance and predictability of the operating system through moving specialized pieces of an operating system into hardware. Utilization of these hardware components to off-load scheduling decisions has been shown to be a practical way to reduce OS jitter[85].

The Hybridthreads system is built with predictability as a design goal. The modules have deterministic execution times and can all act in parallel to each other. They do not depend on the CPU for execution and lack global shared data structures. The Hybridthreads operating system is shown in Figure 3.1 and currently contains dedicated hardware components including:

- Thread Manager

- Thread Scheduler
- Synchronization Manager
- Condition Variable Manager

With the exception of virtual memory, which typically has its own hardware unit, these cores form a basic microkernel. These dedicated cores are programmed in VHDL. They are implemented as custom finite-state machines, realized in hardware on an FPGA. The Hybridthreads cores communicate over the FPGA's internal bus and work together to form the standard set of Operating System primitives, while being able to act independently of each other.

3.2.1 Memory Mapped I/O

Communication and interaction between Hybridthreads components happen through memory mapped I/O over the system bus. The Hybridthreads system exploits the design of the master memory bus to do the majority of its operations atomically through a load command. System buses are typically designed to be atomic, only allowing a single operation at a time and blocking access until that operation is done. Essentially, Hybridthreads is using the parallel arbitration mechanism of the system bus, rather that establishing a software protocol to accomplish the same idea. Parameters for the call are encoded into the lower bits of the load address, and the return of the load indicate the results of this single atomic operation. This can be seen through a typical memory map, as seen in Figure 3.2

While this operation does have the advantage of being atomic, it limits the amount and size of arguments as a tradeoff for defined memory space. This can be seen in Figure 3.2. The Thread Scheduler has been assigned 65KB of address space, giving it 16 bits of possible arguments. This 16 bits of arguments can be divided among command and arguments. In addition, the CBIS has been assigned 1MB of address space, giving it 20 bits of possible arguments. Considering the practical maximum of a 32 bit system, assigning 2GB or half the address space, gives 31 bits of

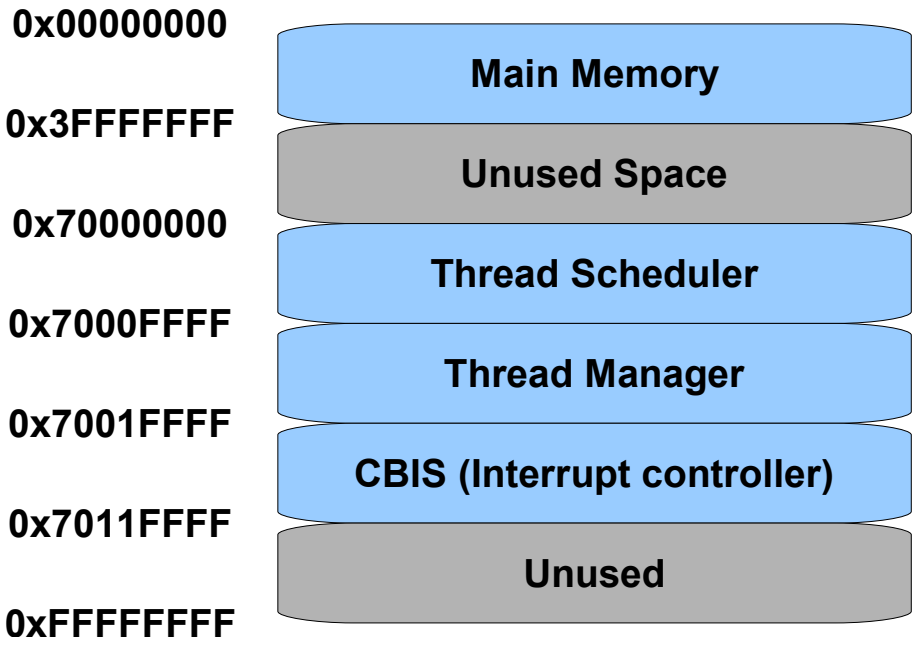

Figure 3.2: Address Map

arguments. Giving the thread scheduler, this amount of space would allow us to have 16 commands, 1024 priority levels, and 131072 threads. While this can be a slight constraint in a 32 bit system, 64 bit systems relax most of the design challenges this proposes. The extra address space in a 64 bit system allows larger parameters without monopolizing the address space. Alternatively, this limit can be worked around with store commands, which not only have an address, but a value, thus doubling the amount of data one can send atomically. This extra encoding comes at a cost because store commands do not allow a return value.

Memory mapped interface operating services have an advantage of allowing access from software and hardware services. These hardware services, including operating system components and other devices, can request service without stopping the CPU. This direct hardware to hardware interaction for operating system services can facilitate new and interesting interactions between hardware, including in Heterogeneous computing environments. In the growing embedded System on a Chip market, this could include advanced interaction with

Graphic Processing Units(GPUs)[86], Digital Signal Processors(DSPs)[87] and Modems[88]. The design of Hybridthreads takes full advantage of direct hardware to hardware communications, with scheduling commands flowing from the synchronization, interrupt and timer cores directly to the thread scheduler.

3.2.2 Thread Manager

The first core of Hybridthreads is the thread manager. The thread manager is responsible for creating, deleting and maintaining state for the threads in the system. Internally, the thread manager keeps track of state information, including running, suspended, exited or ready-to-run, and parent/child relationship. The thread manager is, also, a liaison to the thread scheduler because many thread management operations result in a scheduling decision. The thread manager has two internal data structures, the thread status array and the thread identifier stack.

The thread identifier stack is a data structure that keeps track of unused thread IDs. It does this through a LIFO (Last In First Out) stack created in a BRAM. When initialized, all the free thread IDs are written into the BRAM in order. The stack pointer is initialized to the final entry. As requests for new threads come in, the stack pointer, stored in a register, is deferenced into the BRAM to grab the top of the stack. This value is returned for the new thread and the stack pointer is decremented to the next BRAM value. As threads are deleted, the stack pointer is incremented and the deleted thread ID is placed in the BRAM at the new stack pointer address for reuse.

The Thread Status Array, as seen in Table 3.1, is responsible for storing all the parameters and run time status of a specific thread. It is implemented in a BRAM , with the thread ID as the index. This includes thread ID, parent/child information, joinability, and exit status. Commands to change and look up these data are simple reads and writes from the BRAM with the thread ID as the index. This information is shared with the thread scheduler through the BRAM second access port. This allows the thread scheduler to check on thread validity without contacting the thread manager.

Table 3.1: Thread Status and Identifier BRAM Layout

Field	Width	Bits
Stack Thread ID Entry	8 bits	0-7
Unused	8 bits	8-15
ParentID	8 bits	16-23
Detached	1 bit	24
Joined	1 bit	25
Used	1 bit	26
Exited	1 bit	27
Unused	4 bits	28-31

To save space and be more efficient in the FPGA implementation, the Thread Status and Thread Identifier array share the same BRAM, as seen in Table 3.1. Xilinx Virtex 2 Pro has a 32-bit wide BRAM, natively. Since the arrays require 8 and 12 bits respectively, they can be compressed into one BRAM. Conceptually, they are treated separate, but in the implementation are shared. This does limit simultaneous access to the BRAM, but normal operations can serialize their access.

Internally, a finite state machine responds to commands and compiles information into these two data structures that are shared with the thread scheduler. For example, a Create call will cause the Thread Manager to communicate that the thread needs to be enqueued with the scheduler. These states form all the basic commands to create, delete and change status on threads. There are, also, control signals that go between the thread manager and the thread scheduler, as shown in Figure 3.3. These signals tell the scheduler when to update its status. A list of the thread manager operations is in Table 3.2

3.2.3 Hardware Thread Interface (HWTI)

The Hybridthreads Thread Manager supports an abstract notion of a threads. The Threads Manager implements a pthread interface to support software threads. In addition, it can support

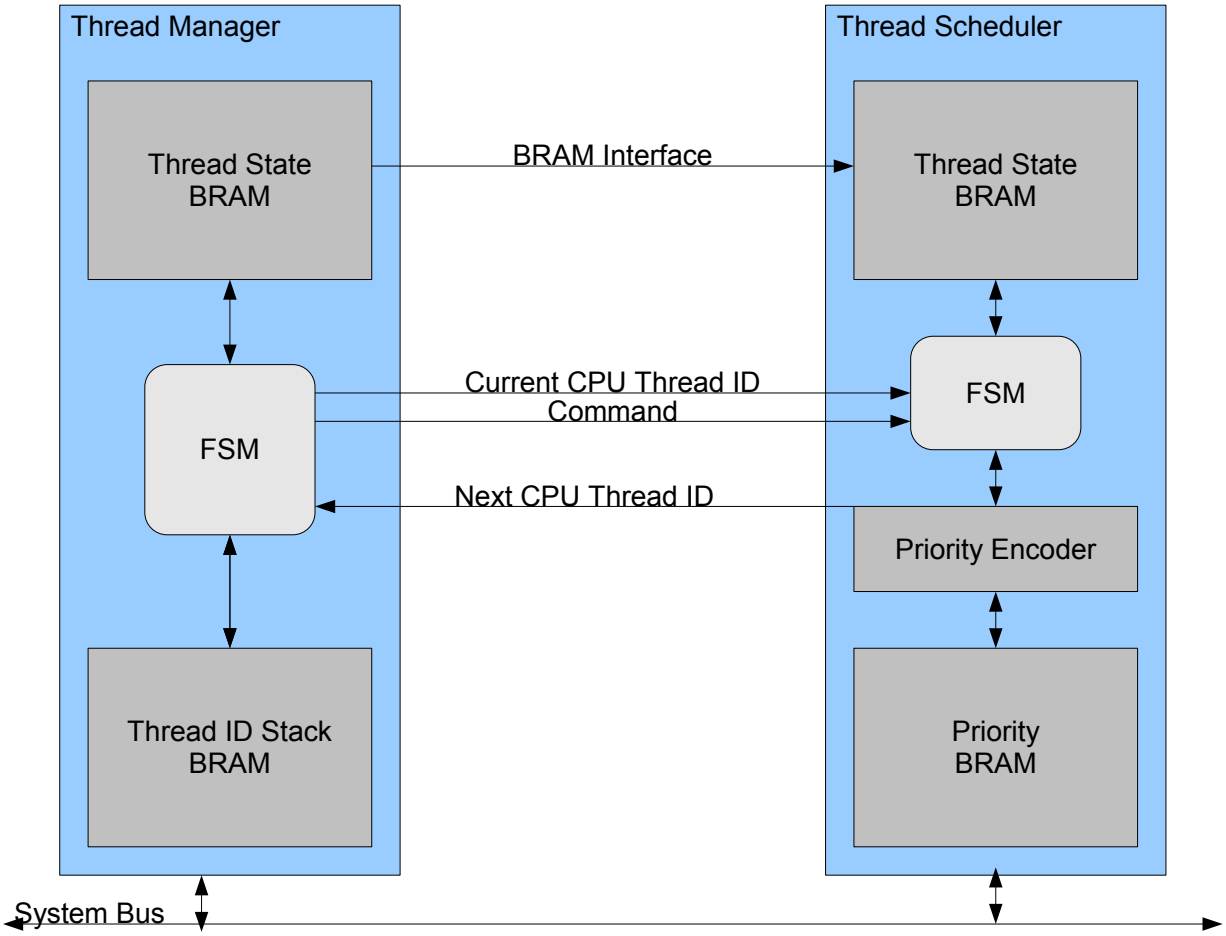

Figure 3.3: Block Diagram of Thread Manager and Scheduler

hardware threads. The concept of hardware threads is a bit different because hardware threads are on uninterruptible and dedicated hardware computational units. The Thread Manager and the other cores allow these hardware threads to interact with operating system primitive through the Hardware Thread Interface (HWTI) core. This is a hardware core that abstracts access to thread and synchronization operations. Allowing access to the same primitives, such as mutexes, through a similar interface helps in porting algorithms from hardware to software. Increasingly, complex interactions of specialized hardware units with the operating system are allowed with operating system primitives accessible without interrupts.

Table 3.2: Thread Manager Commands

Command	Description
Create	Create new thread and Reserve a new TID
Current	Returns the current thread running
Next	Gets the next thread to be run and sets it to current thread
Clear	Deletes and thread and recycles TID
Is_Queued	Query if thread is in ready to run queue
Yield	Yield Current thread to other threads of same priority
Exit	Ends thread and notifies parent of exit status
Join	Blocks parent thread until child exits
Detach	Changes from joinable to detached thread

3.2.4 Thread Scheduler

Originally, the Hybridthreads system uses a simple FIFO scheduler built into the Thread Manager[89]. However this poses a problem in which the storage for thread management and scheduling operations are in the same state machine and these operations cannot happen in parallel. To accommodate for more complex schedulers, the Hybridthreads teams decide to separate the scheduling mechanism into a coprocessor. This thread scheduler can act independently, but is often controlled by the thread manager through a special interface as to not monopolize the system bus. This, also, allows a convenient interface for multiple scheduling algorithms, since the scheduling algorithm is in a separate core. The growth of partially reconfigurable FPGAs[90][91][92] allows changing the algorithm at run-time.

The second core in the Hybridthreads system is the thread scheduler. The thread scheduler works in combination with the thread manager to control which thread is running on the CPU. It is internally organized like the O(1) scheduler of Linux. It keeps track of a doubly linked list for each priority level and uses a high speed priority encoder to determine the highest outstanding priority in the system. It shares information with the thread manager through a few signals and a shared BRAM. In addition, it has two additional data structures, a priority BRAM and another thread scheduling parameter BRAM.

Table 3.3: Thread Manager and Thread Scheduler Shared Signals

Signal	Width
Next_CPU_Thread_ID	8 bits
Current_CPU_Thread_ID	8 bits
Command	12 bits

Table 3.4: Thread Scheduling Parameter BRAM

Field	Width	Bits
Queued	1 bit	0
Priority Level	8 bits	1-8
Ready to Run Queue Next Pointer	8 bits	9-16
Ready to Run Queue Previous Pointer	8 bits	17-24
Unused	7 bits	25-31

The shared signals are Next_CPU_Thread_ID, Current_CPU_Thread_ID and Command. Next_CPU_Thread_ID is a signal from the thread scheduler to the thread manager, indicating the next highest priority thread to be run. Current_CPU_Thread_ID is a signal from the thread manager to the thread scheduler, indicating the currently running thread. These two signals are constantly monitored to determine the Next Thread, of higher priority than the current thread. When this happens, a traditional interrupt is sent to the CPU to kick off changing threads. The Command signal includes both an action, Enqueue_Request or Dequeue_Request and a datum value, a thread ID. These commands are done when the a thread manager operation needs to enqueue or dequeue a thread, such as a delete thread call would dequeue a thread. This command signal will kick off the appropriate thread scheduler logic to respond to the command.

The Priority BRAM, seen in table 3.5, is used to keep track of the data of the threads at each priority level. It does this through a doubly linked list of each priority. The data structure is indexed by priority level. The entry for each priority level contains a head and tail pointer to the list of threads currently queued at that priority level.

Table 3.5: Priority BRAM

Field	Width	Bits
Priority Queue Head Pointer	8 bits	0-7
Priority Queue Tail Pointer	8 bits	8-15
Unused	16 bits	16-31

These head and tail pointers point into the Thread Scheduling Parameter BRAM, seen in Table 3.4, which is indexed by thread ID. The entries in this BRAM are priority, parameters, previous thread, and next thread, creating a doubly linked list of all threads in a priority level. Most operations are fairly quick given this doubly linked list. To add a thread to the queue, simply deference the priority of the thread into the priority BRAM to find the tail pointer of the priority list. With that tail pointer, index into the thread scheduling parameter BRAM and change the next pointer in that entry to the newly added thread. Change the tail pointer to the thread being added in the priority BRAM to complete the operation. More complex operations like dequeue, require walking the list to find the entry and removing it. This is still fairly fast, given that the list is in local BRAM memory.

A 256 bit register keeps track of the priorities with enqueued threads. The priority register is updated on each thread operation to reflect the current state of the doubly linked priority list. If a priority level has no threads in the list because no threads are ready to run, the corresponding bit in this register will be 0. If the priority level has threads ready to be run, the corresponding bit in the register will be 1.

A high speed encoder takes this 256 bit register and determines the highest bit set to 1. It outputs the number of this highest bit as an integer representing the index of the highest priority in the queue. This highest priority level is compared to the priority level of the current running thread. If priority indicates a switch in running process, the thread scheduler will directly interrupt the CPU and switch to the new thread. In parallel, it will fetch the Thread ID of the pending processes from the priority BRAM head pointer for that priority level. The high speed

Table 3.6: Thread Scheduler Commands

Command	Description
Enqueue	Add thread to ready to run queue
Dequeue	Remove thread from the ready to run queue
Get_Entry	Return a threads attributes
Toggle_Preemption	Allow the scheduler to preempt the CPU
Get_Priority	Sets a thread's priority
Set_Priority	Sets a thread's priority
Is_Queued	Test if thread is Queued
Is_Empty	Status Bit to see if threads are queued
Set_SchedParam	Return Scheduling Parameter Entry
Set_Idle_Thread	Set Idle Thread ID
Get_Idle_Thread	Set Idle Thread ID

priority encoder is able to calculate the highest priority level very fast in 5 clock cycles. This allows for extremely fast and jitter free scheduling decisions.

The biggest advantage of this scheduler is that preemption only happens when a thread change is in-progress. Scheduling operations do not speculatively have to include a check for preemption with calls to the core schedule function. The preemption check is always done in parallel. The thread scheduler allows many standard thread management calls to enqueue, dequeue and set priority over the memory bus, allowing both hardware and software to do scheduling operations. The operations of the thread scheduler are listed in Table 3.6

3.2.5 Linux Software Modifications

To get the Thread Scheduler and Thread Manager ported into Linux, many adjustments had to be made to the Linux kernel. This includes modifications to both the Linux Scheduling function, as seen in Table 3.7, and to the internal Linux structures

Table 3.7: Modified Scheduler Functions

Command	Description
sched_init	Initialization of Scheduler
enqueue_task	Add a task into scheduling queue
dequeue_task	Remove a task from scheduling queue
requeue_task	Move a task to the back of its priority queue
sched_fork	Creation of a thread
sched_exit	Thread ending execution
scheduler_tick	Periodic timer tick for updating jiffy
schedule	Decide and switch threads
thread_scheduler_interrupt	Custom Interrupt Handler for the Thread Scheduler

3.2.5.1 Data Structures

To support the Thread Manager, first, a correlation between the Linux thread ID and the Hybridthreads hardware thread ID must be made. This is done by adding a hardware thread ID integer to the task_struct for each linux thread. This allows the internal scheduling functions to easily access the Hybridthreads hardware thread ID. Secondly, a hardwareIDtoTaskStruct mapping table is created. This table is updated upon creation or deletion of a thread. During calls to the Hybridthreads cores, a call typically feeds in a hardware ID, or gets a hardware ID in return. The task_struct pointer allows a software call to easily get the hardware ID to make the call. If a call returns a hardware ID, the hardwareIDtoTaskStruct makes conversion easy.

3.2.5.2 Functions Modifications

The basic linux scheduling functions map easily to the previous Hybridthreads Operating System. Sched_init initializes the hardware cores along with the rest of the linux scheduling code. Sched_fork and sched_exit are modified to map task_struct to hardware ID, and call the create and delete commands to the thread scheduler. Enqueue_task, dequeue_task, and requeue_task map directly onto hardware calls. The biggest changes to the kernel come in the schedule call. In the Hybridthreads version schedule becomes a null function. In software, schedule is typically called

Table 3.8: Mutex BRAM

Field	Width	Bits
Mutex Queue Head Pointer	8 bits	0-7
Mutex Queue Tail Pointer	8 bits	8-15
Unused	16 bits	16-31

Table 3.9: Thread BRAM

Field	Width	Bits
Mutex Queue Next Pointer	8 bits	0-7
Mutex Queue Previous Pointer	8 bits	8-15
Unused	16 bits	16-31

to see if a context switch is needed. In the hardware case, this decision is always happening in parallel and does not need to be executed in software. When a scheduling change is needed, the thread_scheduler_interrupt handler will happen and will context switch to the next highest priority in the system. Thread_scheduler_interrupt uses similar code to the context switch code at the end of the schedule call to do this.

3.2.6 Synchronization Manager

The Synchronization Manager is a separate hardware unit to help facilitate synchronization between threads. The manager implements POSIX compliant [93] mutexes are accessible as memory-mapped I/O Commands. The recursive and non-recursive mutexes support locking and unlocking. When locking an already locked mutex, the scheduler will block the current thread until the mutex is free. All the commands are done through a single atomic read instruction on the bus, allowing for simple atomic mutexes. Similar to the thread schedulers, the synchronization manager has two data structures to keep track of mutexes, a mutex BRAM and a thread BRAM.

The mutex BRAM is indexed by the mutex ID, and contains a head and tail pointers to a linked list of threads blocked by the mutex. The thread BRAM is indexed by thread and

Table 3.10: Synchronization Manager Commands

Command	Description
Lock	Blocks thread until mutex can be locked
Try-Lock	Try to lock mutex, but do not block
Unlock	Unlocks a mutex, enqueue next owner
Owner	TID of owner of Mutex
Kind	Type of Mutex, FAST, RECURSIVE
Count	Recursive count of the Mutex

completes the doubly linked list through its previous entry and next entry for each thread.

A call to lock a mutex essentially indexes in the mutex BRAM. If free, the thread ID of the calling thread is marked as the owner of the mutex. If already locked, the thread is added to the doubly linked list through manipulation of the tail pointer, and updating the previous tail pointer thread to point to the newly queued thread. Other operations are fairly similar to the other hardware core, and simple involve manipulating the linked list in local BRAM memory.

As mutexes are unlocked, the synchronization manager will directly issue a call to the thread manager to enqueue the next waiting thread. This causes a parallel scheduling decision and possibly a preemption of the current thread.

With this internal organization, the synchronization manger is very fast and completes operations in 5 clock cycles or less. This, combined with the single atomic read instruction on the bus, allow for incredibly efficient synchronization. A list of the operations of the synchronization manager is included in Table 3.10.

3.2.7 Condition Variable Manager

Similar to the synchronization manager, the condition variable manager helps facilitate coordination between threads. It implements the concept of POSIX compliant condition variables and allows the management of scheduling threads based on the condition variables. A condition

Table 3.11: Condition Variable Manager Commands

Command	Description
Wait	Blocks thread until condition is meet
Signal	Signal and unblock single thread on condition
Broadcast	Signal and unblock all threads on condition

variable allows a thread to wait on a certain condition to exist. Many threads can block on the same condition and eventually the condition is signaled or broadcasted. A signal will enqueue a signal thread waiting, while a broadcast will enqueue all threads waiting. Like the synchronization manager, it directly communicates to the thread scheduler dequeuing and enqueueing threads as the condition variable paradigm dictates.

The condition variable manager has data structures similar to the synchronization manager. Through the use of BRAMs, a linked list of the threads, waiting on a condition variable is made and the threads are dequeued from that list as signal or broadcast operations are made. This is done through a condition variable BRAM, containing the head and tail pointer of the doubly linked list, and a Thread BRAM containing the next pointer. Most operations for the Condition Variable manager are fast, however a broadcast call can take additional time, depending on the number of threads that need to be enqueued. The operations of the condition variable manager are listed in Table 3.11

3.3 Extensions to Hybridthreads

From these original cores, this work extends Hybridthreads through a new CPU Bypass Interrupt Scheduler (CBIS) for interrupt management. The original Timer Core for Hybridthreads is also extended for better integration with Linux.

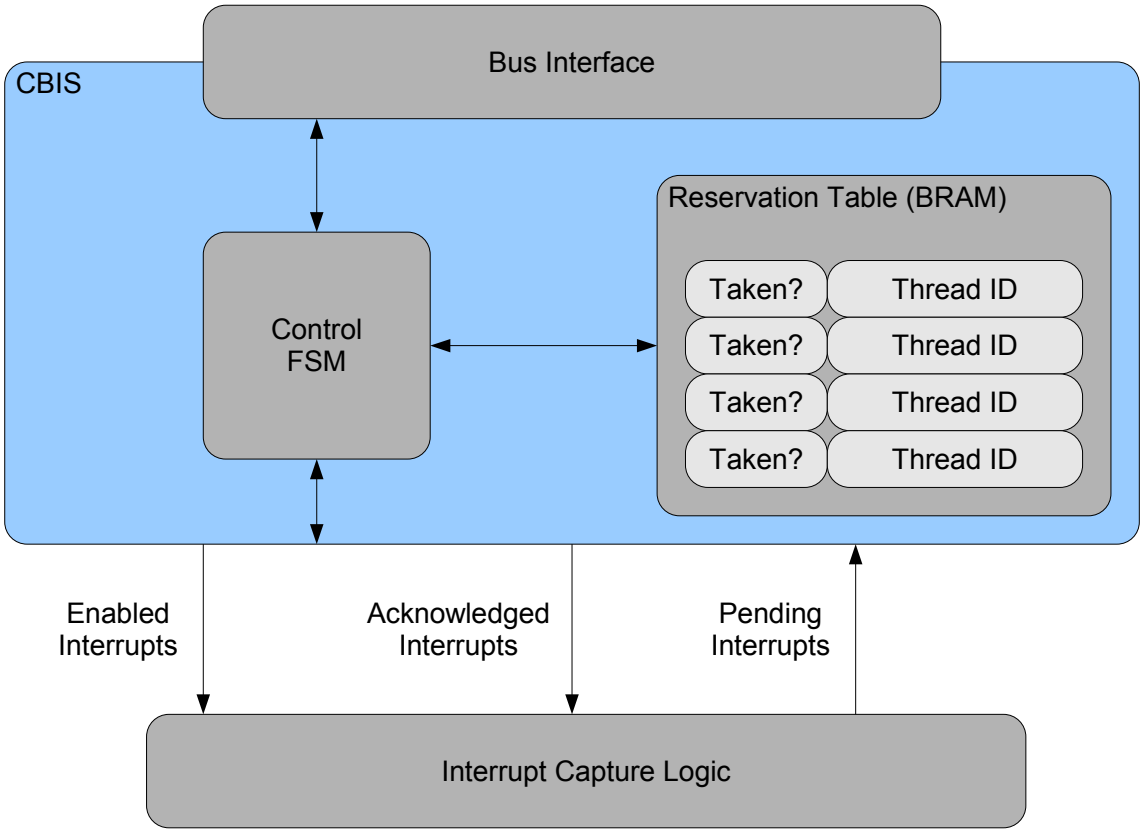

Figure 3.4: CPU Bypass Interrupt Scheduler

3.3.1 CPU Bypass Interrupt Scheduler (CBIS)

The CBIS, in combination with the hardware thread scheduler, allows the concept of a unified priority model. Because interrupts go into the CBIS and directly to the thread scheduler over the bus, the CPU does not need to process the interrupt, unless the thread scheduler determines that the interrupt threads are a higher priority than the running threads. Essentially, this brings a new unified priority model that allows a programmer the choice to have interrupts at high priority, emulating a traditional system, or intermixed with real time threads.

Table 3.12: CPU Bypass Interrupt Scheduler Commands

Command	Description
associate	Blocks thread until interrupt occurs
enable	Enables an interrupt
disable	Disables an interrupt
clear	Clears pending interrupt bit
read_associated	Returns thread ID associated to thread
read_pending	Reads status of pending interrupt registers
read_enabled	Reads status of enabled interrupt registers

Table 3.13: CBIS BRAM

Field	Width	Bits
Reserved	1 bits	0
Thread ID	8 bits	1-8
Unused	23 bits	9-31

3.3.1.1 Overview

The design of the CBIS is fairly simple and a list of the operations of the CPU Bypass Interrupt Schedulers is included in Table 3.12. A block diagram of the CBIS can be seen in Figure 3.4. The CBIS consists of two data structures, an interrupt reservation table BRAM and an interrupt capture logic.

The interrupt reservation table is a simple BRAM that is indexed by an interrupt number. In each entry, there is a marker if the interrupt is reserved, and what thread ID is associated with that interrupt. This interrupt reservation table is updated on the blocking associate command. Essentially, an interrupt processing thread will atomically execute the associate call while waiting for more interrupts. During the associate call, the CBIS will check if there is a pending interrupt. If there is a pending interrupt, the call will return immediately for processing. Otherwise, the thread will be blocked and associated with the interrupt in the internal BRAM. The thread scheduler will run another thread until an interrupt occurs.

The second data structure is the interrupt capture logic. Similar to most interrupt controllers, the interrupt capture logic responds to either a level triggered interrupt or an edge triggered interrupt. Once an interrupt is noticed, a pending bit is set. This pending bit is logically ANDed with the enabled bit. An enabled bit is only set when an interrupt has been associated with it. The combination of these tell the CBIS when an interrupt should be processed. This logic is fed through a priority encoder to go from the interrupt line to highest interrupt number pending. When the interrupt happens, the priority encoder determines the number of the interrupt to be processed. This kicks off the control FSM and indexes into the BRAM to receive the thread associated with the interrupt. This thread is sent to the thread scheduler to be enqueued. Typically, once the thread scheduler decides to run the interrupt service thread, the thread will clear the pending bit, do the work of the interrupt, and reassociate to the interrupt completing the cycle.

3.3.1.2 Hardware Implementation

The CBIS is implemented on a Xilinx ML310 platform. The core is written in VHDL and follows Xilinx's OPB User Core Template. The OPB, On Chip Peripheral Bus, is Xilinx's way to create custom core and have them access the same address spaces as the PowerPC 405 chip. The OPB bus is 32 bits and supports multiple master devices along with slave devices. This OPB bus is available on many Xilinx Platforms.

For our CBIS implementation, there will need to be responses for a slave device in order to read commands and responses for a master device in order to write to the bus to communicate with the Thread Scheduler. The VHDL implementation instantiates an OPB Master core with the IPIC bus interfaces, as seen in Table 3.14. This interfaces arbitrates bus access and translates commands into easier signals.

To first respond to commands from the bus, the CBIS will watch for the Bus2IP_RdReq line to go high, indicating that someone on the bus has requested access to an address in our assigned range. When the pin goes high, the CBIS immediately pulls the IP2Bus_ToutSup line high to

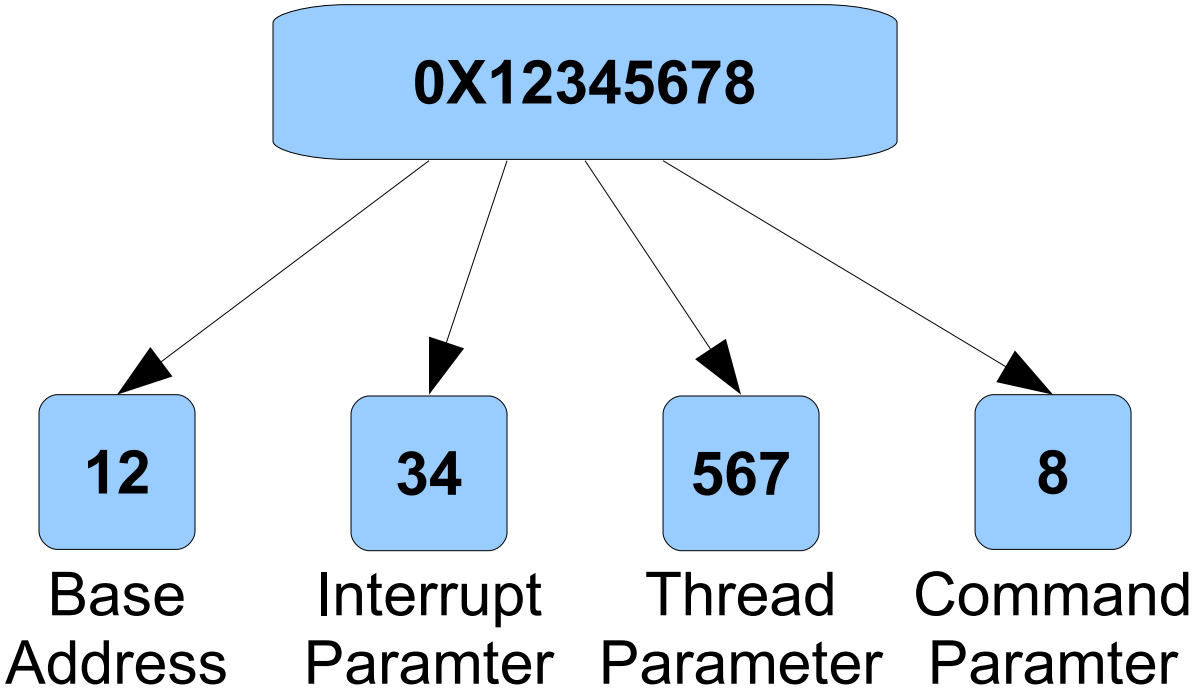

Figure 3.5: CBIS Address Breakdown

prevent a timeout. By default, the Xilinx bus will fail a transaction that takes longer than 8 clock cycles, if IP2Bus_ToutSup is not high. While some of the transactions are less than 8 clock cycles, pulling IP2Bus_ToutSup high while we are busy is simplified by not being timed out. In parallel to this, the address lines are now valid and the CBIS triggers its finite state machine to start processing the command.

From the 32 address lines, the address data are logically sorted into groups that represent the command and arguments as describe previously in the Memory Mapped IO section. The CBIS breakdown of the address can be seen in Figure 3.5. The bottom 4 bits of the bus form the commands, allowing up to 16 commands. The second 12 bits form the thread ID parameter for the commands, allowing 4096 threads. The next 8 bits form the interrupt ID parameter, allowing up to 256 interrupts. In total, there are 24 bits of address space used by the CBIS, taking 16MB of the space away from the total of 4GB in the 32 bit system.

At this point, the finite state machine decides what state to jump into from idle. Some commands are very simple and just modify or return the status of a register. The enable, disable and clear commands simply update a register. This is done by feeding the parameters of the command into the register latching the results. The read_pending and read_enable return the current value of that register. To return the status of a register, the CBIS follows the Xilinx paradigm and connects the output of the register to the IP2Bus_Data lines, raises the IP2Bus_RdAck, and lowers the IP2Bus_ToutSup back to 0. This completes a full read cycle for these simple commands. For the simple command the finite state machine never leaves the idle state. Everything is done in the single clock cycle with no need to leave the idle state.

For more advanced commands, the finite state machine must read and write to the Xilinx Block Ram, BRAM, data storage. A Xilinx BRAM is a dual ported memory that is hard coded into the FPGA Fabric. BRAMs are configurable to many different sizes and can be combined into larger BRAMs. The signal to control a BRAM is seen in Table 3.16. The CBIS only uses a single port of the BRAM and uses it to store thread data that are associated with the interrupt. The BRAM has a few additional complexities to note. To read from the BRAM requires two cycles, a setup cycle and an idle cycle. The next cycle has the data available. This expands the finite state machines to handle this extra BRAM complexity.

The first command that can write to the BRAM is the Associate Command. On the associate command, the thread parameter is first compared to the corresponding bit in the pending interrupt register to see if there is a pending interrupt. If not, it raises BRAM_WEN_A to indicate a write, uses the interrupt parameter as the address to BRAM_Addr_A and the thread parameter as the value in BRAM_Din_A. This signals are held for a full clock cycle, the associate_write state, to commit them to the BRAM.

The second command that will access the BRAM is the read_associated command, which will return the thread currently associated with an interrupt. From idle, the CBIS jump into the rtn_thread_bram. This sets up the BRAM for reading by raising BRAM_EN_A and setting the

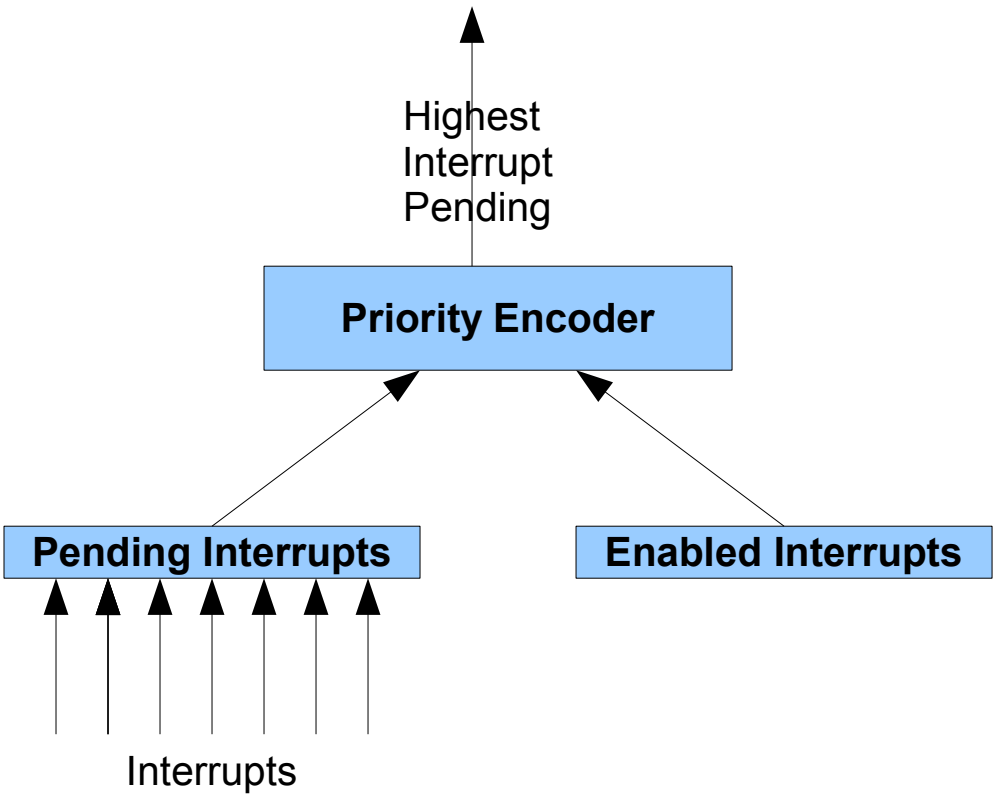

Figure 3.6: CBIS Interrupt Logic Diagram

interrupt parameter for BRAM_Addr_A. Then the CBIS transitions into the rtn_thread_idle state, allowing for the 2 cycle read from the Xilinx BRAM. Lastly, the CBIS transitions into the rtn_thread_bus which connects the BRAM_Dout_A pins to the IP2Bus_Data to return the BRAM value over the bus.

Lastly, outside of the commands, an asynchronous interrupt can trigger the FSM machine to change. When an interrupt occurs, it is latched in the pending interrupts registers. This triggers the interrupt logic, as seen in Figure 3.6. The pending interrupts are ANDed with the enabled interrupts and feed into the priority encoder. This priority encoder changes from an interrupt pin number into an integer. When this priority encoder outputs an integer the CBIS FSM transitions from idle into interrupt_read_bram to process the interrupt. In the interrupt_read_bram the highest interrupt pending integer is connected into the BRAM_Addr_A while the BRAM_EN_A is pulled

```
#define CBIS_BASE_ADDRESS          0x71000000

#define COMMAND_SHIFT              0
#define COMMAND_MASK               0x0000000F
#define THREAD_SHIFT               4
#define THREAD_MASK                0x00000FFF
#define INTR_SHIFT                 16
#define INTR_MASK                  0x000000FF

#define CBIS_COMMAND(command, thread, intr) \
        CBIS_BASE_ADDRESS |
        (((command)&COMMAND_MASK)<<COMMAND_SHIFT)   |
        (((thread)&THREAD_MASK)<<THREAD_SHIFT)) |
        (((intr)&INTR_MASK)<<INTR_SHIFT))
```

Figure 3.7: Memory Mapped I/O Macro

high to signal a read. The FSM transitions into the interrupt_read_bram_idle and waits an idle cycle because of the BRAM 2 cycle read. Lastly, the FSM transitions into the interrupt_add_thread state in which it combines the base address of the Thread Scheduler and the output of the BRAM on BRAM_Dout_A to form an address for writing to the bus. This address is connected to IP2IP_Addr. The IP2Bus_MstRdReq is pulled high, indicating a read request generated to the thread scheduler. These are held until Bus2IP_MstAck or Bus2IP_MstTimeOut is pulled high, indicating a Acknowledgement or a Timeout. With the response, the pending bit is cleared and waiting for more interrupts.

3.3.1.3 Software Implementation

The software implementation of the CBIS starts with memory mapped I/O. Essentially, the CBIS has software pieces to generate addresses for specific commands. As shown in Figure 3.5, a command to the CBIS consist of a base address, thread parameter, interrupt parameter, and command parameter. To form the full address, the command parameter is added to the thread parameter multiplied by 2^4, the interrupt parameter multiplied by 2^{16}, and the base address of the

```
void interrupt_thread( int interrupt )
{
        while(true)
        {
                associate( interrupt )

                //Service Routine
        }
}
```

Figure 3.8: CBIS Interrupt Loop

```
void associate( int interrupt )
{
        int thread = get_thread_id();p
        if( CBIS_COMMAND(ASSOCIATE, thread, interrupt) )
        {
                //Already have pending interrupt service immediately
                return;
        }
        else
        {
                // Block thread and allow others to run
                dequeue_thread(thread);
        }
}
```

Figure 3.9: Associate Call

CBIS. This forms the address for a read command, which is deferenced to make a call to the CBIS. Because of careful choice of the size of the parameters in multiples of 2 and the base address having all lower bits as 0, we can speed up this process of adding and multiplying these 4 items. Multiplying by powers of 2 turns into bit shifts and adding turns into logic OR. Figure 3.7 illustrates the code macro to generate the addresses. Each parameter is masked into range, then shifted and logically ORed together to form the final address.

With this memory mapped I/O macro, most of the accessor functions to the CBIS become wrappers around this macro. The associate command has a bit more logic because of how it

changes interrupts. When a device registers an interrupt service routine, the CBIS will take the handler function and start a new thread for it. The pseudo-code for the thread looks like Figure 3.8. The thread is a loop of associating for the interrupt, blocking on the call until an interrupt happens and calling the service routine in response.

The internals to the associate command look similar to Figure 3.9. With the thread ID of the interrupt thread, an associate call command is made. This hardware call will return a 1, if a pending interrupt is outstanding with the software, immediately returning to process the interrupt. If the hardware call return 0, indicating a successful association, the software will call a dequeue_thread function to the scheduler. This will stop the interrupt thread from running until the CBIS hardware calls add_thread in response to an interrupt.

3.3.2 Timer Core

The timer core is a separate hardware unit to help facilitate threads blocking on timers. It takes requests for threads to block waiting on a timer. When the timer expires, it calls the thread scheduler to make that thread eligible to run again. Internally, it contains one data structure and timer list BRAM.

This timer list BRAM is index by a timer ID. Each entry contains a previous and next pointer, along with a clock cycle of expiration. These pointers make a doubly linked list of timers in sorted order of expiration. When a timer is added into the list, the core iterates through the list to add it in order. The head of the list, or the first to expire, stores its clock cycle of expiration in a register. There is an on chip counter that is compared with this value every clock cycle. Once the timer has expired, an add thread call to the scheduler is made, and the core retrieves the next timer that will expire. After a little more list maintenance, the timer core is waiting for the next timer to expire. The operations for the Timer Core are listed in Table 3.17

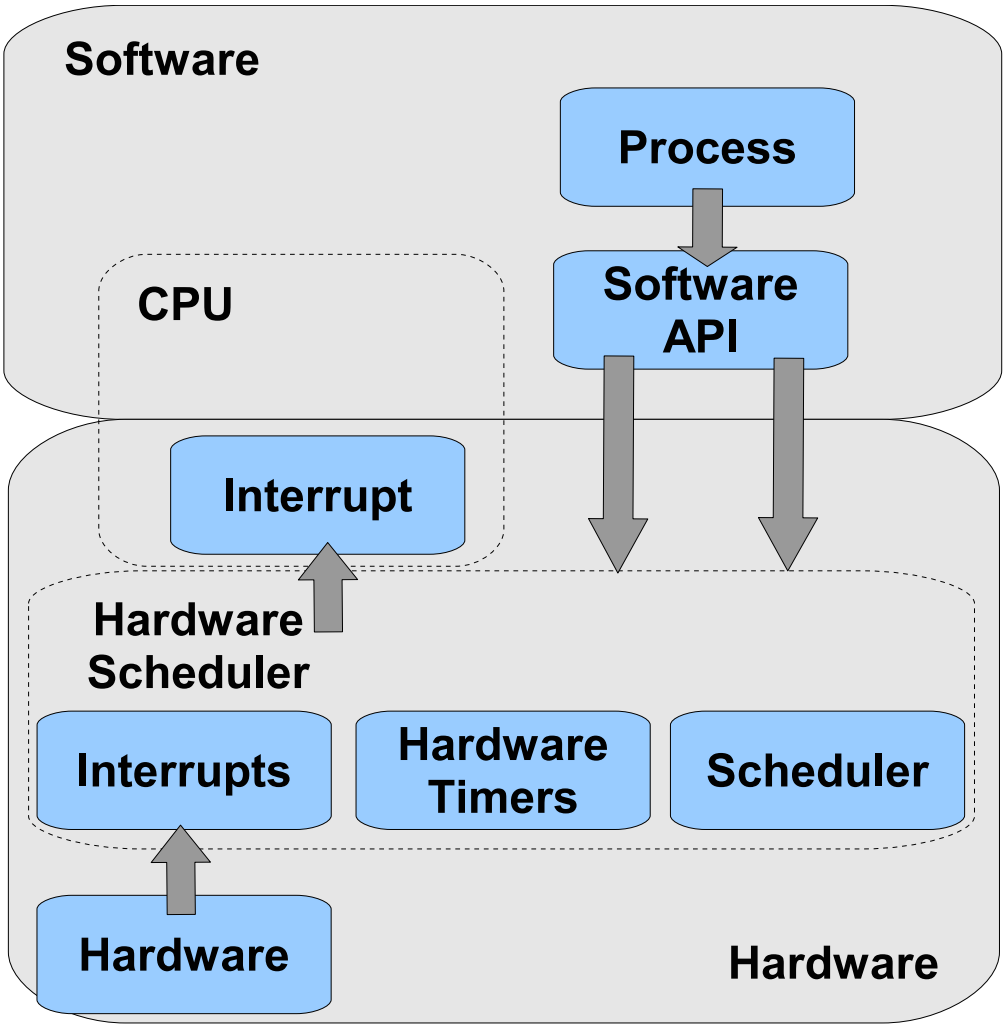

Figure 3.10: Unified Scheduler Model

3.4 Unified Interrupt and Scheduling Controller

After applying the knowledge learned from the CBIS and Hybridthreads, it is decided to design another core with a few different design constraints. Hybridthreads are meant for hybrid hardware/software FPGA type systems. While this is an interesting paradigm, typically, FPGAs are used for prototyping and early development system. Once the hardware has matured, it is typically reenvisioned in an ASIC. The concept of changing hardware threads and their interaction does not make sense for most general purpose systems. Most general purpose systems

routinely have only changing software due to cost. As a result, the Hybridthreads design is taken and adapted to a pure software solution. This has many effects on the size and functionality of this system. The unified scheduler interrupt controller is the evolution of Hybridthreads to a software only system.

The goal of the software only solution is to simplify the hardware to the bare minimum that will make a performance impact. Each call to the hardware scheduler is evaluated for the split of work between the hardware and software portion of the call. For most of the calls, this dissertation's adaptation of the Hybridthreads hardware removed most of the linked list hardware manipulation. While useful and needed when hardware threads are present, the linked list manipulation can often piggyback onto the ends of software scheduling calls at little impact to the predictability.

Through removing list manipulation, the software thread only solution is a simplification of the previous Hybridthreads scheduler design. The scheduler portion just consists of enough information to make and accelerate a scheduling decisions. Since scheduling activities are only invoked from software and interrupts to change the software running, some of the thread and priority linked lists can remain in main CPU memory. This means it needs to know the current running thread and a list of the ready to run queue. The list of the ready to run queue includes the first thread in each linked list per priority level. Since inter-level process switching is initiated by a software yield command, timer interrupt, disk interrupt, or another blocking call, processing this linked list for each priority level in software saves a significant amount of logic. The standard implementation has a large state machine to perform linked list manipulation from the BRAM. The unified implementation only keeps the ready to run information stored in a BRAM, and there is a corresponding bitfield of occupied priority levels. Like the previous Hybridthreads scheduler, a priority encoder determines the top bit of this bitfield to determine the top priority of the thread ready to run next. This is seen as the Software Task Priorities block in Figure 3.11.

To integrate interrupts into this logic a similar top interrupt priority is determined. First a table

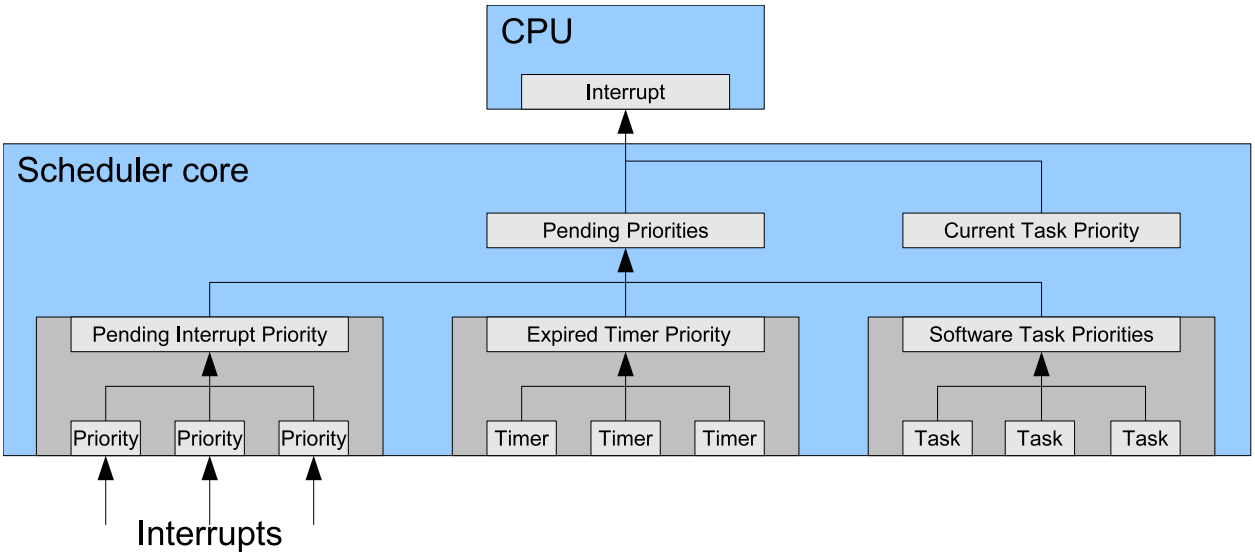

Figure 3.11: Unified Scheduler Interrupt Controller

mapping interrupt thread IDs and priority is made. This is done in a BRAM similar to the previous CBIS. Interrupt detection logic is ANDed with Enabled and then priority encoded to find the outstanding interrupts. This outstanding interrupt number is used to index the table and lookup the priority of the interrupt to be added to a interrupt priority bitfield. Similar to the scheduler, this interrupt priority bitfield is priority encoded to determine the highest priority interrupt. This is seen as the Pending Interrupt Priorities block in Figure 3.11.

The timer portions function in a similar way as the interrupt. A group of timers, their priorities, and thread associations are stored in a BRAM As the timers expire, the priority is added to the timer interrupt priority bitfield. This bitfield is priority encoded into the highest priority timer thread register, as seen as the Expired Timer Priority block in Figure 3.11

The combination of these three highest priority threads ready to run, and the knowledge of the current thread and its priority, allow this Unified Scheduler Interrupt Controller to have enough information to make scheduling decisions, as seen in Figure 3.11. With a portion of the logic removed, is this still a scheduler? The Unified Scheduler Interrupt Controller is a hybrid scheduler and interrupt controller. Although, the design leans closer to an interrupt controller. It is a balance

of keeping more information available for making scheduling decisions outside the CPU.

Table 3.14: Xilink IPIC Bus Interface[1]

Signal Name	Range	I/O	Description
Bus2IP_Addr	0:C_<bus>_AWIDTH-1	I	Address to User Logic
Bus2IP_BE	0:C_<bus>_DWIDTH/8-1	I	Byte enables to User Logic
Bus2IP_Burst	none	I	Burst-mode qualifier to User Logic
Bus2IP_Clk	none	I	IPIC clock. Identical to the <bus>clock
Bus2IP_CE	0:C_NUM_CE-1	I	chip enable to User Logic
Bus2IP_CS	0:C_NUM_CS-1	I	chip select to User Logic
Bus2IP_Data	0:C_<bus>_DWIDTH-1	I	Data to User Logic
Bus2IP_Freeze	none	I	Tells the User Logic to freeze
Bus2IP_RdCE	0:C_NUM_CE-1	I	Read enables to User Logic
Bus2IP_Reset	none	I	Signal to reset the User Logic
Bus2IP_RNW	none	I	Read/Not Write Signal to User Logic
Bus2IP_WrCE	0:C_NUM_CE-1	I	Write enables to User Logic
IP2Bus_Ack	none	O	Acknowledgement from User Logic
IP2Bus_Data	0:C_<bus>_DWIDTH-1	O	Data from IP
IP2Bus_Error	none	O	Error response
IP2Bus_Intr	0:C_IP_INTR_NUM-1	O	Interrupt event signals from User Logic
IP2Bus_PostedWrInh	none	O	Posted write inhibit from User Logic
IP2Bus_Retry	none	O	Retry response from User Logic
IP2Bus_ToutSup	none	O	Timeout suppress from User Logic
Bus2IP_MstError	none	I	Master Error from IPIF
Bus2IP_MstLastAck	none	I	Master Last Acknowledge from IPIF
Bus2IP_MstAck	none	I	Master Acknowledge from IPIF
Bus2IP_MstRetry	none	I	Master Retry from IPIF
Bus2IP_MstTimeOut	none	I	Master Timeout from IPIF
IP2Bus_Addr	0:C_<bus>_AWIDTH-1	O	<bus>address for the master transaction
IP2Bus_Clk	none	O	Future signal to allow for dualclock- domain
IP2Bus_MstBE	0:C_<bus>_DWIDTH/8-1	O	Byte-enables qualifiers from User Logic
IP2Bus_MstBurst	none	O	Burst qualifier from User Logic
IP2Bus_MstBusLock	none	O	Bus-lock qualifier from User Logic
IP2Bus_MstReq	none	O	Master request from User Logic
IP2Bus_MstRNW	none	O	Read/Not Write from User Logic
IP2IP_Addr	0:C_<bus>_AWIDTH-1	O	Device address for the master transaction

Table 3.15: CBIS FSM states

State	Description
idle	Default idle state
associate_write	Check Pending and Write to BRAM
interrupt_read_bram	Index into BRAM based on highest interrupt pending
interrupt_read_bram_idle	Wait idle cycle for BRAM
interrupt_add_thread	Write to bus for addThread command
rtn_thread_bram	Setup BRAM for reading
rtn_thread_idle	Idle cycle for BRAM read
rtn_thread_bus	Return result of BRAM read on bus

Table 3.16: Xilink BRAM Interface[2]

Signal Name	**Interface**	**I/O**	**Description**
BRAM_Rst_A	Port A	I	BRAM Reset, Active High
BRAM_Clk_A	Port A	I	BRAM Clock
BRAM_EN_A	Port A	I	BRAM Enable, Active High
BRAM_WEN_A	Port A	I	BRAM Write Enable, Active High
BRAM_Addr_A	Port A	I	BRAM Address
BRAM_Din_A	Port A	O	BRAM Data Input
BRAM_Dout_A	Port A	I	BRAM Data Output
BRAM_Rst_B	Port B	I	BRAM Reset, Active High
BRAM_Clk_B	Port B	I	BRAM Clock
BRAM_EN_B	Port B	I	BRAM Enable, Active High
BRAM_WEN_B	Port B	I	BRAM Write Enable, Active High
BRAM_Addr_B	Port B	I	BRAM Address
BRAM_Din_B	Port B	O	BRAM Data Output
BRAM_Dout_B	Port B	I	BRAM Data Input

Table 3.17: Timer Commands

Command	**Description**
addTimer	Sleep a thread for specified clock cycles
timerStatus	Number of clock cycles remaining until wake up
cancelTimer	Cancel timer in progress

Table 3.18: Unified Scheduler Commands

Command	**Description**
Current	Returns the current thread running
Next	Get the next thread to be run and sets it to current thread
SetThread	Sets a thread to the top of it's priority level in the ready to run queue
Associate	Block thread until interrupt occurs
Clear	Clears pending interrupt bit
Read	Reads status of pending interrupt bit
addTimer	Sleep a thread for x clock cycles
timerStatus	Number of clock cycles remaining until wake up
cancelTimer	Cancel timer in progress

Chapter 4

Evaluation

We evaluate our hypothesis that the unified hardware software scheduling policy has larger performance benefit compared to it's hardware cost by experimentation. In particular we perform a comparative analysis of efficiency and effectiveness across four different configurations, all based on a standard Linux kernel. All analysis was performed on a Xilinx ML310 platform[94]. The Hybridthreads kernel as well as all hardware modifications for this work were implemented in VHDL. All interfaces to the hardware resident components are register-based and accessible by the PowerPC processor across a standard bus (OPB). All tests were conducted with the Power PC core running at 100 MHz. The Linux 2.4 kernel was ported onto the PPC, and then additional modifications to the lower level mechanisms were performed to invoke the hardware mechanisms of the modified Hybridthreads kernel. The use of the older 2.4 instead of a more recent Linux kernel was an imposed limitation of using the ML310 board. Xilinx only provides a Board Support Package(BSP) for the 2.4 Linux kernel. A rudimentary 2.6 kernel for the ML310 board does exist, but it does not support a hard disk, network, video and other peripherals needed for many of the testing scenarios below.

These tests run on 4 different systems for comparison and evaluation as listed in Table 4.1. The systems show a progression of different ways to reduce the performance on impact of the

Table 4.1: Test Systems

System	Full Hardware OS	Partial Hardware OS	Interrupts as Threads
Base Linux	No	No	No
PREEMPT_RT	No	No	Yes
Unified	No	Yes	Yes
HybridThreads	Yes	Yes	Yes

scheulder and interrupts. The base control system is the pure software Linux 2.4 kernel. The PREEMPT_RT system is also a software system, but adds in a better scheduler and the interrupts as threads concept. The Unified system takes this one step further by integrating the Scheduler and Interrupt Controller into a single hardware core. Lastly, the Hybridthreads system completes the hardware software transition by integrating a majority of its OS primitives into hardware.

A part of this work was modifying the existing hardware based Hybridthreads scheduler. The original hardware based scheduler included additional logic to handle the scheduling of custom accelerator threads synthesized as hardware circuits. The original scheduler was re-designed and simplified to schedule only software threads. The resulting Unified Scheduler Interrupt Controller resulted from this redesign as a minimalistic approach to hardware schedulers. Reducing the size of the original cores and reducing the amount of linked list logic significantly reduced the size of the original scheduler core. The Unified Scheduler Interrupt Controller was then integrated into the Linux Kernel as our second test system.

We wanted to run our applications on the original Linux 2.4 Kernel unaltered for comparative purposes. We also wanted to perform as fair a comparison as possible, which motivated us to use the PREEMPT_RT patch. The PREEMPT_RT patch represents as close an approximation as is possible using software methods to mimic the hardware based Unified Scheduler Interrupt Controller. The PREEMPT_RT patch is mainly a 2.6 kernel patch and was not available for our 2.4.18 kernel. To circumvent this issue we performed an approximate port of the patch. The approximation used for testing on the older 2.4.18 PREEMPT patch, along with a backporting of

Table 4.2: Test Scenarios

Test	Type	Description
Base Operation	Overhead	Measurements of frequent scheduling calls
Overhead of an Interrupt	Overhead	Impact of interrupts on high-priority threads
Hackbench	Overhead	Measurement of overall scheduling overhead
Dhrystone MIPS	Overhead	Measurement of overall performance
Apache Benchmark	Latency	Measurement of network response for Web Pages
Cost of Timer	Latency	Measurement of Timer impact on high-priority threads
Balancing Processing and Interrupts	Latency	Performance under DDoS attack

the code enabled interrupts to be preemptable and runnable within a processes context. While not an exact match of the PREEMPT_RT, it non the less gives a fair approximation of a pure software configuration to minimize jitter. This represents our fourth and final system.

A basic Linux userspace was compiled for the board for testing purposed along with our benchmarks. All tests were run on all four systems. The first system consists of an unmodified Linux system. The second system is the 2.4.18 PREEMPT_RT approximation port. The third system is the Hybridthreads implementation of Linux. The final system is the Unified System. The tests are divided into two broad categories labeled Efficiency and Control. They are further separated by synthetic benchmarks versus real scenarios. Lastly, in the conclusion, a hardware evaluation compares the size of hardware and relates it to the performance gains found.

4.1 Efficiency

4.1.1 Base Operation Timings

The first benchmark is a synthetic benchmark for analyzing various scheduling operation latencies. Timing is achieved by calling an on-chip counter at the beginning and end of each scheduling operation. The length value reported is an average over a set of 100,000 measurements. The operations being measured are the schedule call, an interrupt and the

addThread call. While schedule and addThread are simple operations to measure, the interrupt is more difficult. The interrupt is timed by reading the on-chip counter and then triggering an interrupt with a memory-mapped I/O call. Once the interrupt handler answers the interrupt, the counter is again read and the delay value computed.

4.1.1.1 Analysis

Figure 4.1 shows the results of this test. The two software schedulers, Software and PREEMPT_RT, show similar timing delays. This is to be expected as the PREEMPT_RT kernel only minimally changes the base scheduler call. The two hardware schedulers show an improvement of approximately 20 microseconds, This represents about a 50% reduction in latency compared to the two software schedulers. This can be attributed to the fact that the hardware based systems have already identified the next highest priority thread to be scheduled before the run_sched call is invoked. Conversely, the software schedulers must performing iterative processing to determine the next scheduling decision.

The interrupt latency timing results show a similar trend. The two software schedulers show similar timings with PREEMPT_RT, a slight 0.5 microseconds faster. The hardware schedulers both finish approximately 10 microseconds faster, or in 45% of the software scheduler times. It is worth pointing out that the software system shows a slight advantage over the hardware systems in the time taken to deliver the interrupt to the CPU. The hardware based systems route the interrupt request through an additional hardware module before it is delivered to the CPU. The PREEMPT_RT Kernel adds a small delay to change the interrupt into the context of a user process. Even with the delay, the PREEMPT_RT patch shows a 0.5 microsecond faster response that the software scheduler without the patch. Overall the software schedulers, after interrupting the CPU, must perform processing to decide if the interrupt should be delayed. This extra processing causes an increase in interrupt latency when compared with the hardware based Hybridthreads and Unified Scheduler Interrupt Controller systems.

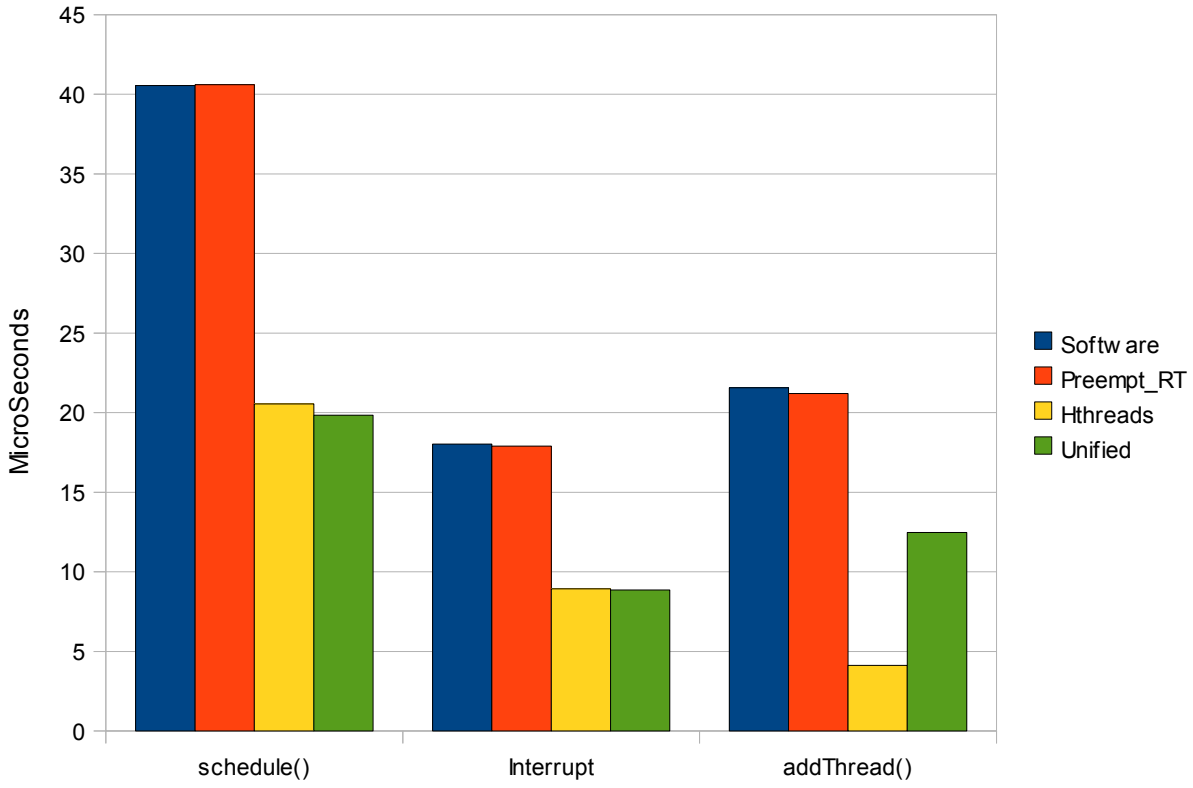

Figure 4.1: Base Operation Timings

Finally, addThread shows the best result for the Hybridthreads, with a 5x latency reduction compared to the software approaches, and a 3x reduction compared to the Unified Scheduler Interrupt Controller system. This is a result of the Hybridthreads system performing the addThread command in hardware, including manipulation of the linked list in the priority queue. Latency reductions are further achieved by eliminating the need to perform accesses to the main memory. The Unified system takes a hybrid approach by doing most of the work in software with only the last few steps in hardware. This saves still provides latency reduction when compared to the software approaches. The two software kernels are the slowest, with none of the work accelerated by hardware.

4.1.2 Overhead of an Interrupt

This benchmark shows the effect of high priority interrupts on the throughput of a critical process. Interrupts asynchronously trigger workloads of device drivers which run at higher priority than application threads in the system. The unified hardware software priority model and PREEMPT _RT help to relieve this interference of general execution from interrupts by allowing system designers to set interrupt priority with respect to software threads priority. The hardware scheduler goes one step further by not having to interrupt the CPU to mark an interrupt pending. PREEMPT_RT still must minimally interrupt the CPU to mark each pending interrupt.

This additional control is demonstrated through the overhead of an interrupt benchmark. This benchmark consists of two simple programs. The first simulates a generic application program by timing a simple loop that performs integer operations. The loop is unrolled until the program size is approximately 16Kb, the same size of the PowerPC instruction cache. Integer operations are performed across a 16Kb memory segment, the same size as the data cache. This is unrolling is done to try and keep the program running in purely cache memory, avoiding some of the uncertainty associated with slower memory accesses due to unpredictable cache misses. The second program provides stimulus by triggering a hardware core that was designed to generate periodic interrupts. To demonstrate the effect of a top-half interrupt routine, the servicing routine for these interrupts is kept very small and only increments a count of the interrupts received.

The benchmark is run under three scenarios. In the first scenario no interrupts are generated. This provides a base case for evaluating the performance degradation in the presence of interrupts. The second scenario models a fairly low rate of interrupts, 10 per second. The third increases the interrupt rate to 30 per second. 10 and 30 were chosen because they were measured as an average idle network interrupts on the system.

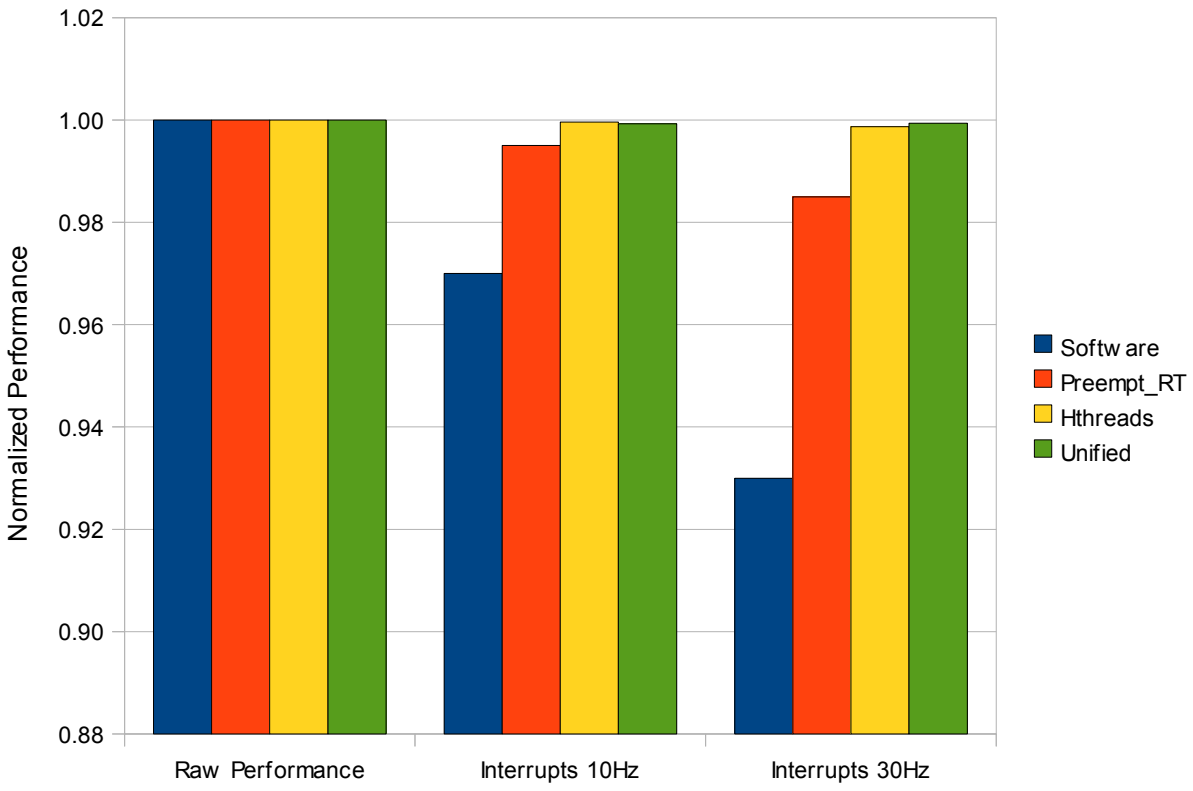

Figure 4.2: Overhead of Interrupt

4.1.2.1 Analysis

The benchmark results show fairly predictable behaviors given the design on the individuals systems. The raw performance run shows no difference between the 4 systems. This is expected as the performance number is from a single threaded benchmark in a otherwise idle system. The operating system should have little effect on the performance of a run without interrupts. Next the interrupts at 10 Hz run shows how the systems react slightly differently to this low priority interrupt. The traditional Linux system shows a degradation of 3%. This degradation can be attributed to the overhead in processing the interrupts. The PREEMPT_RT system does quite a bit better only showing a 0.5% decrease in performance at 10 interrupts a second. PREEMPT_RT still does stop the processor during an interrupt, but for only minimal processing. The difference in processing makes it faster than the base Linux system. Next the HThreads and the Unified system

show almost no change in performance as they are within 0.05% of the original numbers. Given the setup of this system, with interrupts lower priority then the performance thread, interrupts are essentially ignored. They should have no effect on the performance. Moving on to the interrupts at 30Hz run, we see a continuation of what has happened in the 10Hz run. The base Linux system has a 7% drop from the baseline performance and PREEMPT_RT patch has a 1.5% drop. The Hthreads and Unified system, once again, have little to no effect on overall performance.

These results reconfirm the theoretical design of the hardware schedulers, CPUs are interrupted only when a higher priority process or interrupt is ready to run on them. They never stop to check or decide if they should switch. This is a stark comparison to the software schedulers who are only trying to work around the limitations of the hardware designs they run on. PREEMPT_RT is one of the best software designs to avoid this and while it does noticeably better than the base Linux system, it cannot fully match the hardware schedulers.

4.1.3 Hackbench

Hackbench is a standard benchmark used for evaluating different Linux schedulers[24]. It was developed by Rusty Russel for performance testing of the Linux 2.4 kernel scheduler in comparison to what is now the scheduler in the Linux 2.6 kernel. It has served as a major benchmark for showing the performance improvements in the Linux 2.6 kernel. Hackbench can be described as a producer and consumer test. Producers provide data as fast as consumers can handle using a greedy algorithm. Hackbench creates producer and consumer threads in groups of 20. For this benchmark, Hackbench is run many times with varying amounts of threads in order to compare the Traditional, PREEMPT_RT, Hthread and unified kernels.

4.1.3.1 Analysis

The results in Figure 4.3 show an evaluation of the four schedulers. Since hackbench is primarily a series of calls invoking the scheduler, it can yield insight into the efficiency of each of

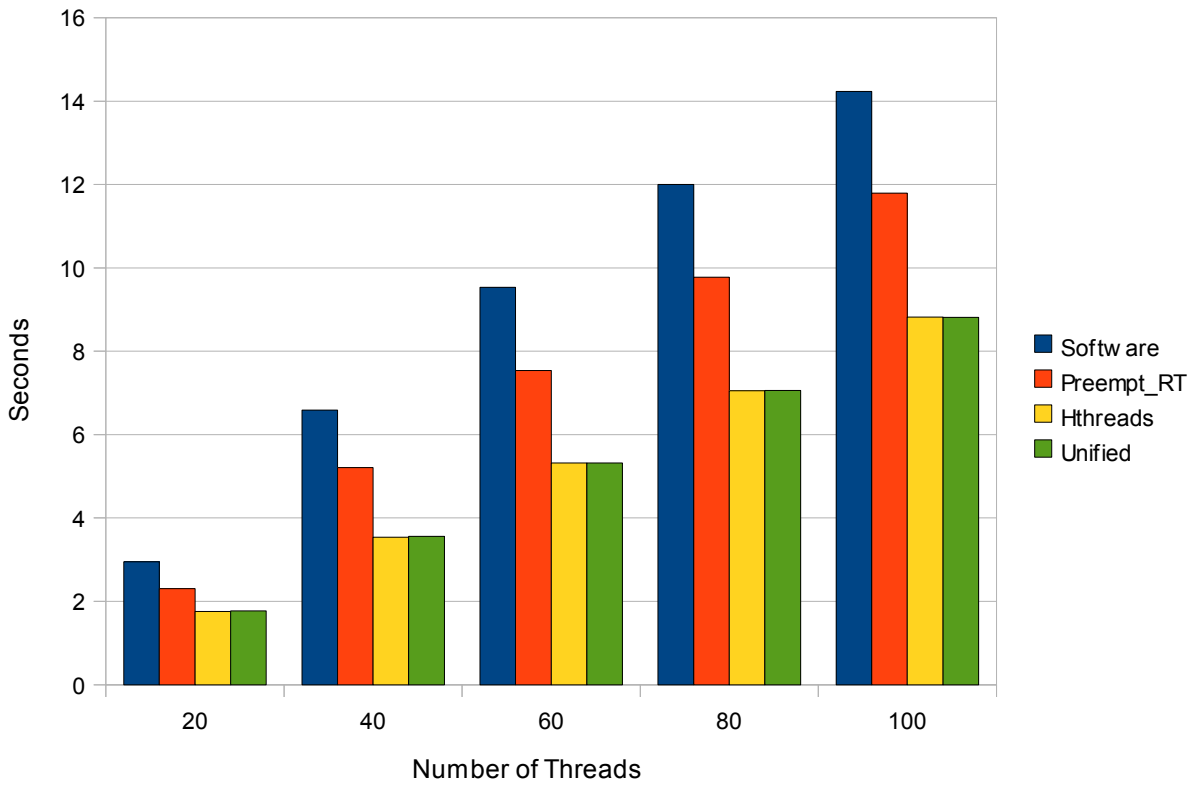

Figure 4.3: Hackbench

our scheduler configurations. Both hardware schedulers show a 40% decrease in scheduling time across the board compared to the software scheduler within the PREEMPT_RT patch, with a slight gain as the number of threads increases. The PREEMPT_RT kernel enjoys approximately a 20% decrease compared to the standard software scheduler. Given that hackbench is a very scheduler intensive benchmark, these improved results correlate with earlier scheduling timing, with the exception of PREEMPT_RT. While the calls measured earlier show PREEMPT_RT at the same, this benchmarks seems to be faster due to other kernel efficiencies in PREEMPT_RT. This is likely due to file and pipe operations between threads.

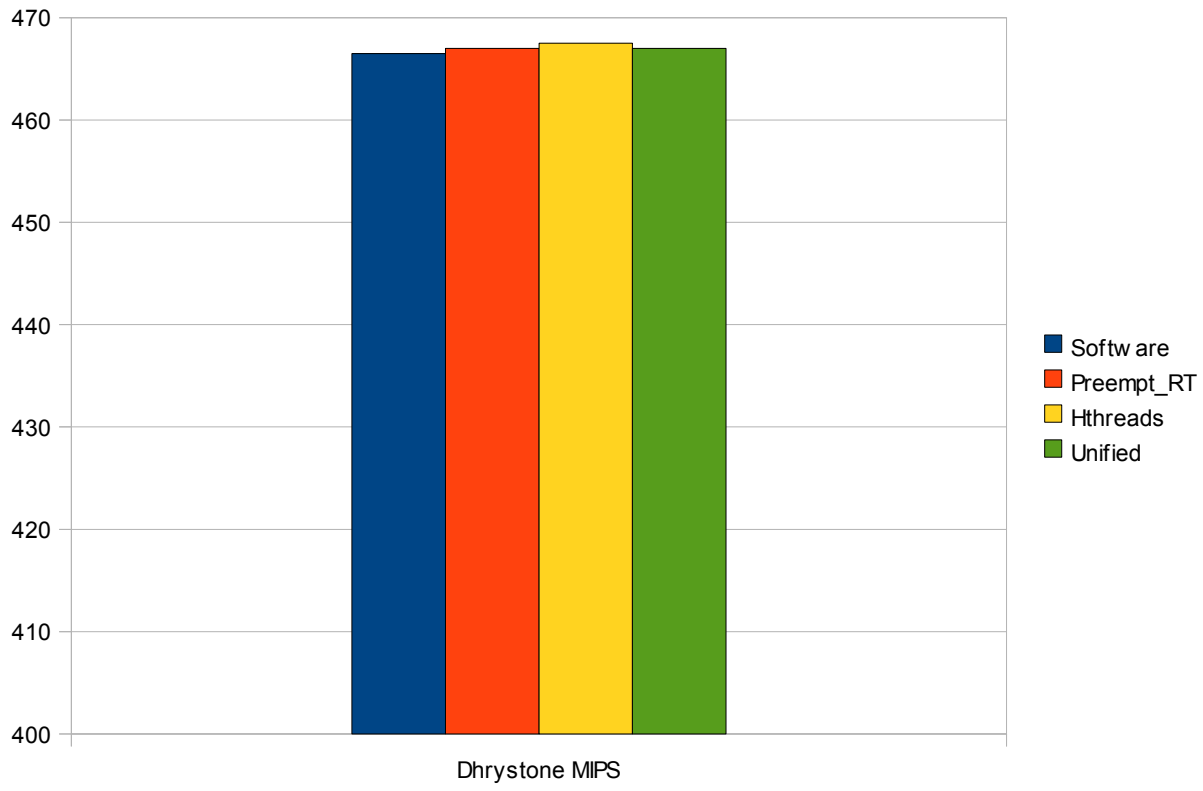

Figure 4.4: Dhrystone MIPS

4.1.4 Dhrystone MIPS

Dhrystone MIPS is a standard synthetic benchmark made by Reinhol P. Weicker[95]. It measures the ability to process integer programming in a standard way. It is been used for many years to evaluate the performance of processors. It is a measure of the number of Dhrystones per second executed with a Dhrystone being the iteration of the main code loop. It is used here to evaluate the single threaded performance affect of the schedulers.

4.1.4.1 Analysis

The results in Figure 4.4 show that there is little to no advantage in the different types of schedulers. All end up in the 466 Dhrystone MIPS range, with variance of 1 Dhrystone MIPs.

Table 4.3: Apache Benchmark Averages

System	Average
Software	69.89ms
Preempt_RT	63.86ms
HThreads	54.98ms
Unified	54.85ms

Dhrysone MIPs is a single threaded benchmark. In a system with only one thread running, the scheduler should not impact this benchmark, as shown in the results. More complex systems with multiple threads will have different results.

4.2 Control

4.2.1 Web Server

Websites are a very common asynchronous workload for servers. A Web server waits for asynchronous requests on the network adapter. In response to these requests, it serves Web pages and files. It can handle many concurrent connections and deliver many files at once. This workload represents a very asynchronous multithreaded simulation.

To test this work flow, an Apache Webserver is compiled for the device. The Apache HTTP server benchmarking tool, AB (Apache Benchmark)[25], is used to test the throughput on the device for Apache Webserver. A simple webpage is served up, and a second computer is used to test the throughput and latency of the Webserver. The primary objective of this benchmark is to compare the variation in latencies between the approaches when running an the Apache Webserver.

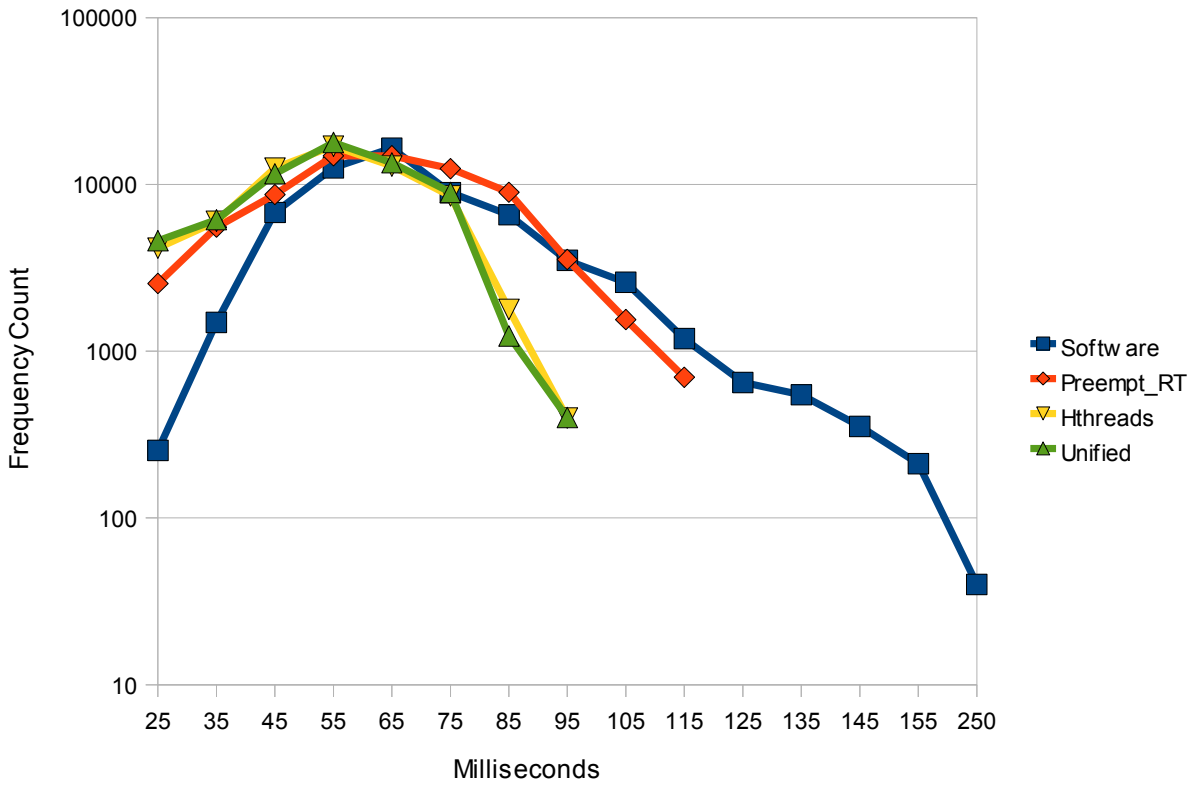

Figure 4.5: Apache Benchmark

4.2.1.1 Analysis

The results of AB are shown in Figure 4.5. A histogram of the timings is shown from 70,000 results. As a note, the last bucket in the histogram includes a few outliers up to 250 milliseconds from the software kernel. This modification makes the histogram not drawn to scale in order to make the histogram a bit more readable. Lastly, it is important to note that the frequency count axis is on the logarithmic scale.

Several interesting results can be seen from the histogram. First, the average response time is actually fairly close between the systems. The Unified and Hybridthreads system had similar averages of 59.98 and 59.85 milliseconds. PREEMPT_RT has a slightly slower 61.86 millisecond average and the base Linux system was the slowest at 63.89 milliseconds.

The second interesting results is the worst case response time. The worst case response time is

critically important in real-time scenarios. A single delayed result can be an incorrect result. Worst case response time on the base linux system was poor compared to the unified and hybridthreads systems. The slowest base Linux response was 2.5 times as long as the slowest hybridthreads and unified response. In fact, 9% of the responses on the base Linux system came in slower than the slowest unified and hybridthreads response. PREEMPT_RT improved upon the base Linux system, but still had a poor worst case response time. The longest PREEMPT_RT result was 20% slower than the unified and hybridthreads systems. 3% of the total PREEMPT_RT responses came in slower than the slowest unified and hybridthreads response.

Overall, this is a good benchmark to demonstrate how the randomness of jitter affect the performance of a system. There are many causes of random jitter that create these distribution of responses. Network hardware, interrupt, cache misses and many other complex systems can add jitter to a system. However, the hybridthreads and unified system show that they have reduced the overall jitter of the system through careful reduction of scheduling jitter. This is especially apparent with the reduction of worst case execution time.

4.2.2 Cost of a Timers

While external interrupts are a common source of jitter, timers also add to the problem. Depending on the system hardware implementation of timers, they can contribute significantly to the jitter in a system without respect to priority. Most general purpose operating systems set the timers to generate periodic interrupts. During these periodic interrupts the operating system checks to make sure timers have not expired. Alternatively, they can sort timers and put the closest expiring timer in a processor timer interrupt. Due to hardware limitations, many of these timers are often multiplexed onto the same hardware timer interrupt without respect for the priority of the timer. This causes problems that, when the task with low priority is preempted, a timer for that task can still be set to go off at the interrupt level, and be queued in the kernel. Unimportant timers, such as TCP timeouts and low priority periodic checks, cause unnecessary

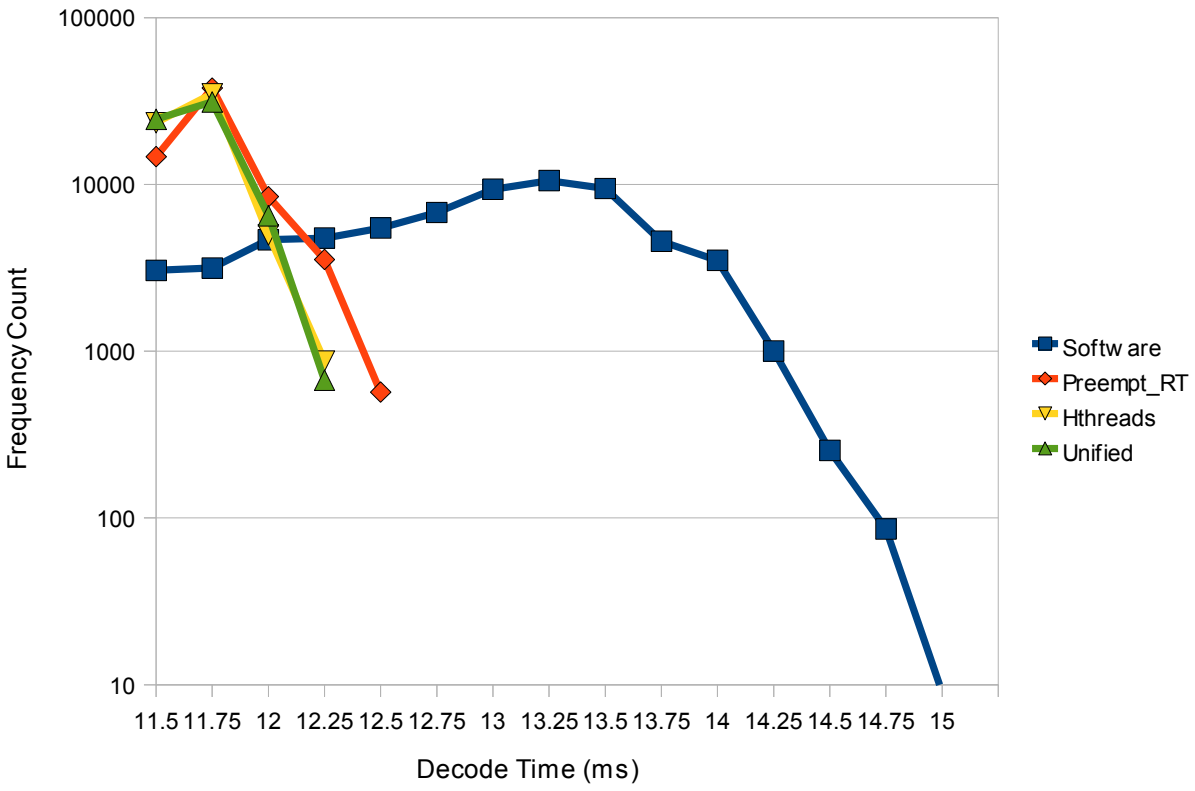

Figure 4.6: Cost of a Timer

jitter in the system through hardware timers interrupting higher priority processes.

This benchmark shows the effects of timer overhead. The benchmark emulates a conceptual embedded device that is decoding an MP3, a soft real-time use case[96], and periodically checks for network updates. The periodic task is set up to decode MP3 frames every 26.12 ms at a high priority. During this decoding, TCP timeouts are simulated from a low priority process by trying to connect repeatedly to an unresponsive computer. The TCP timeout is adjusted to randomize its affect on the MP3 decoding. Although the TCP process is a lower priority, the network adapter driver is not adjusted from its original form. The time to decode each MP3 frame is measured and plotted in a histogram. This benchmark is run on the four schedulers to measure their effects on decode time. In PREEMPT_RT and the hardware schedulers, the interrupt priority is set lower than the decode priority.

Table 4.4: Cost of a Timer Averages

System	Average
Software	12.92ms
Preempt_RT	11.76ms
HThreads	11.69ms
Unified	11.68ms

4.2.2.1 Analysis

The next benchmark tries to show the ability for a lower priority process to subvert the priority model utilizing timers. The results are shown in a histogram in Figure 4.6. The histogram shows two important ideas. There is some natural variance in decoding MP3s. Different frames have different complexities and can thus take different times to decode. This is seen across all schedulers. Both of the hardware schedulers show similar histogram curves with the PREEMPT_RT having only a few frames take slightly longer to decode. This is attributed to the minimal interrupt processing of PREEMPT_RT. Lastly, the traditional linux kernel has results in Figure 4.6, showing the unpredictability added to the system by a seemly harmless timer into a traditional scheduling model.

4.2.3 Balancing Processing and Interrupts

Many systems are susceptible to interrupt overload. Interrupt overload is a condition in which the frequency of interrupts to the system is significantly beyond the capacity to process those interrupts. Interrupt overload will often cause failure of a device because the massive amounts of requests will be queued in the system until the system runs out of memory or other resources. This eventually runs into design constraints, like static number of buffers or file handles.

Misbehaving and faulty hardware can cause interrupt overload. However, Denial of Service (DoS) attacks are the most commonly known cause of interrupt overload. Thousands of PCs will

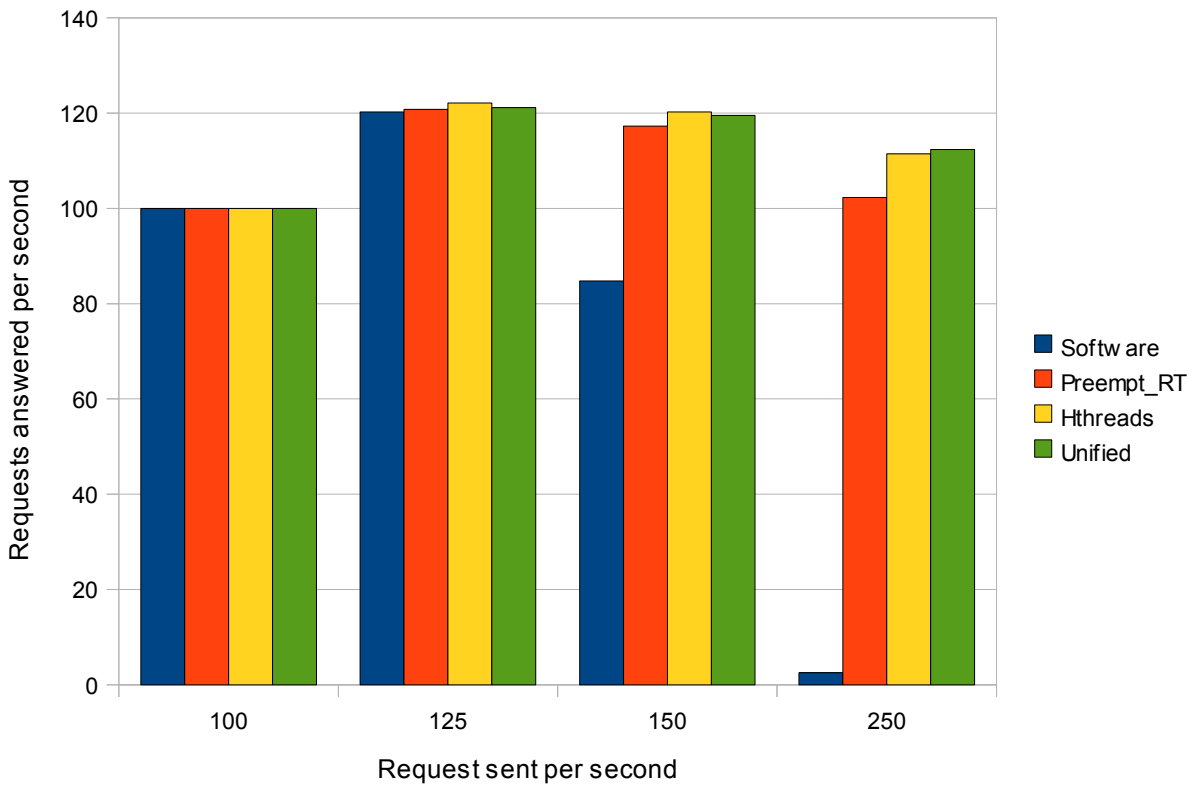

Figure 4.7: Interrupt Overload

try to repeatedly request data from a web server simultaneously resulting in the failure of the Web Server. With network cards and other interrupt generating devices becoming capable of tens to hundreds of thousands of interrupts a second, it can become necessary to balance the initial processing of these interrupts versus the completion of software work. The unified scheduling model allows users to manipulate network interrupt priorities to balance processing requests.

To model the flexibility of defense against interrupt overload, and specifically a DoS attack, the second benchmark consists of a simple Webserver that listens for connections and starts a thread to answer the requests from that connection. Another computer is generating and trying to complete these connections at a specified rate. From this second computer, successful requests are measured.

4.2.3.1 Analysis

The last benchmark show a case of interrupt overload, or more specifically, a DoS attack. Figure 4.7 shows the number of completed requests versus total number of requests. The results in Figure 4.7, interestingly, illustrate results similar to the previous test. The PREEMPT_RT only suffers mildly. The original scheduler suffers much more dramatically once this experiment is past the maximum capacity of the system, approximately 120 requests per second. In the extreme case of 250 requests per second, the traditional system fails during the test because of memory constraints and the inherent preference of interrupt processing. The hardware scheduler maintains near the maximum throughput. A bit of processing is lost, likely due to packet loss causing minor overhead.

Chapter 5

Conclusion

This dissertation explores the benefits of a minimal hardware scheduler in general purpose computing, breaking the traditional mold of the hardware interrupt paradigm, and unifying hardware and software scheduling. The HybridThreads kernel is ported to Linux, showing its versatility to integrate into a general purpose operating system. Interrupts are integrated into the Hybridthreads kernel, allowing thread level control over the priority of interrupts. Finally, the Hybridthreads cores are reenvisioned and redesigned to apply better to software only systems, forming the Unified Scheduler Interrupt Controller. These hardware operating system schedulers are benchmarked and analyzed to breakdown the performance difference between the two

Table 5.1: Hybridthreads Hardware Sizes

Core	**Slices**
Mutex	344 Slices
Conditional	294 Slices
Thread Manager	500 Slices
Scheduler	1455 Slices
Timer Queue	842 Slices
CBIS	667 Slices
Total	4102 Slices

approaches in comparison to an older traditional Linux kernel and a backport of the PREEMPT_RT kernel.

The experiments illustrate that both hardware kernels show performance advantages over both software linux kernels. The efficiency benchmarks show that, in raw scheduling benchmarks, this Unified scheduler has a 50% speed increase on the order of tens of microseconds. Benchmarks with heavily scheduling, like Hackbench, saw 40% speed increases. However, PREEMPT_RT also saw a 20% speed increase over the legacy kernel. While these are increases in heavy scheduling, typical multithreaded scenarios show a more moderate increase because scheduling is only a small fraction of their overhead. The biggest benefit from this design is shown as jitter reduction in the control benchmarks. This jitter reduction has a wide range of effects in real-time performance. The scheduling of interrupts, often the largest source of jitter in operating system, extends the ability of systems to more and more real time tasks. In addition, interrupt overload can be migrated by these thread level scheduling semantics, helping to avoid common problems like DDOS attacks.

These scheduler gains are small and noteworthy, but at what cost? All hardware additions must be evaluated for the performance measurement versus the hardware cost (gates and power). The industry and the open market are the best reference to evaluate this ratio. Intel's lead chip designer, Anand Chandrasehker, recently shared his "One Percent" design rule in reference to their latest microprocessors. The "One Percent" rule states that adding new features to most processors require that a one percent increase in performance results in no more than one percent of power increase [97]. Intel's latest microprocessor are actually more limited by power than in cost, and are making many of the design decisions in their latest "Core" micro processors to minimize power. This is true for many processors these days, as they approach 100+ watts of peak power[98]. While the test setup does not allow us to evaluate power, the gate count of the Hardware Cores versus the processor should give a rough estimate of the power consumption.

In evaluating the hardware cores, the cost in size and power of such a device should be very

Table 5.2: Unified Hardware Size

Core	**Slices**
Unified	887 Slices

small. From Table 5.1, the total original Hybridthreads system is approximately 4102 slices. Table 5.2 shows that the size of the unified model reduces this to 887 slices on a Xilinx Virtex 2 pro.

The 887 slices of Xilinx Virtex logic has a very rough estimate of 100,000-300,000 gates needed to implement it in a traditional ASIC design. This is an order of magnitude simpler than some of the more complex hardware schedulers[89], and small enough that it would take into account only a small portion of the gates on a system on a chip. Even simple modern processors have a range of 25 to 100 million gates.

With this hardware cost in mind, is this hardware scheduler a good choice for next generation processors? It speeds up scheduling, but has a much smaller speedup on the overall system. It reduces jitter slightly better than one of the best software approximations in PREEMPT_RT. There seems to be value in the Unified Scheduler Interrupt Controller, given the hardware cost, but not overwhelmingly so as it is. While next generation processors likely do not need hardware schedulers like Hybridthreads, developing smarter interrupt controllers to integrate with thread scheduling and timers seem appropriate, even if the full Unified Scheduler Interrupt Controller is not included.

Bibliography

[1] Xilinx, "User core templates reference guide," April 2003, ftp://ftp.xilinx.com/pub/documentation/misc/user_core_templates_ref_guide.pdf.

[2] Xilinx, "Ip processor block ram block," April 2003, http://www.xilinx.com/support/documentation/ip_documentation/bram_block.pdf.

[3] P. B. G. Abraham Silberschatz and G. Gagne, *Operating System Concepts*, 6th ed. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[4] C. Boyer, "The 360 revolution," http://www-306.ibm.com/software/os/zseries/pdf/360Revolution_0406.pdf.

[5] Johnston, "Vse: A look at the past 40 years," *z/Journal*, April 2005, http://www.zjournal.com/index.cfm?section=article&aid=293.

[6] M. A. Ertl and D. Gregg, "The behavior of efficient virtual machine interpreters on modern architectures," in *In Euro-Par 2001*. Springer LNCS, 2001, pp. 403–412.

[7] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*, D. Penrose, Ed. Morgan Kaufmann, 2003.

[8] M. D. Natale and J. A. Stankovic, "Scheduling distributed real-time tasks with minimum jitter." *IEEE Trans. Computers*, vol. 49, no. 4, pp. 303–316, 2000. [Online]. Available: http://dblp.uni-trier.de/db/journals/tc/tc49.html#NataleS00

[9] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta*, 1998, pp. 119–129.

[10] I. Molnar, "Linux low latency patch for multimedia applications," 2001, http://people.redhat.com/mingo/lowlatency-patches.

[11] F. M. Proctor, "Measuring Performance in Real-Time Linux," in *Presentation from the Proceedings of the 3rd Real-Time Linux Workshop (RTLW)*, Milan, Italy, Nov. 2001.

[12] A. Kumar, "Multiprocessing with the completely fair scheduler," Jan 2008, http://www.ibm.com/developerworks/linux/library/l-cfs/index.html.

[13] D. Bovet and M. Cesati, *Understanding the Linux Kernel, Second Edition*, 2nd ed., A. Oram, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[14] J. Regehr and U. Duongsaa, "Preventing interrupt overload," *SIGPLAN Not.*, vol. 40, pp. 50–58, June 2005. [Online]. Available: http://doi.acm.org/10.1145/1070891.1065918

[15] M. L. Mark Pavlidis, Alexandre Korobkine, "Linux real-time platform:34 real-time variant evaluation," 2002, http://www.cas.mcmaster.ca/~pavlidmh/linux.pdf.

[16] I. Molnar, "Preempt rt patches," July 2002, http://www.kernel.org/pub/linux/kernel/projects/rt/.

[17] A. Bouchhima, X. Chen, F. Pétrot, W. O. Cesário, and A. A. Jerraya, "A unified hw/sw interface model to remove discontinuities between hw and sw design," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 159–163. [Online]. Available: http://doi.acm.org/10.1145/1086228.1086258

[18] H. R. Arabnia, Ed., *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, Nevada, USA*. CSREA Press, 2000.

[19] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, and W. Luk, "Reconfigurable computing: architectures and design methods," in *IEE Proceedings - Computers and Digital Techniques*, 2005, pp. 193–207.

[20] T. Saxe and B. Faith. Less is more with fpgas., ee times. [Online]. Available: http://www.eetimes.com/electronics-news/4050083/Less-is-More-with-FPGAs

[21] T. Dorta, J. Jiménez, J. L. Martín, U. Bidarte, and A. Astarloa, "Overview of fpga-based multiprocessor systems," in *Proceedings of the 2009 International Conference on Reconfigurable Computing and FPGAs*, ser. RECONFIG '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 273–278. [Online]. Available: http://dx.doi.org/10.1109/ReConFig.2009.15

[22] E. K. Anderson, "Abstracting the hardware/software boundary through a standard system support layer and architecture," Ph.D. dissertation, Lawrence, KS, USA, 2007, aAI3258307.

[23] "Hthreads," December 2011, http://hthreads.csce.uark.edu/wiki/About_Hthreads.

[24] R. Russell, "hackbench: New multiqueue scheduler benchmark," December 2001, http://lkml.org/lkml/2001/12/11/19.

[25] Apache, "ab - apache http server benchmarking tool," December 2011, http://httpd.apache.org/docs/2.0/programs/ab.html.

[26] D. L. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving Programming Model Abstractions for Reconfigurable Computing," *IEEE Transactions on VLSI Systems*, vol. 16, no. 1, pp. 34–44, 2008.

[27] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. L. Andrews, "Hthreads: A computational model for reconfigurable devices," in *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL), Madrid, Spain, August 28-30, 2006*. IEEE, 2006, pp. 1–4.

[28] J. A. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7053

[29] L. Rizvanovic, "Comparison between real time operative systems in hardware and software," Master's thesis, Malardelens University, Sweden, 2001.

[30] D. D. Silva, O. Krieger, R. W. Wisniewski, A. Waterland, D. Tam, and A. Baumann, "K42: an infrastructure for operating system research," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 34–42, April 2006. [Online]. Available: http://doi.acm.org/10.1145/1131322.1131333

[31] D. R. Engler, M. F. Kaashoek, and J. O. Jr, "Exokernel: an operating system architecture for application-level resource management," in *In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 95), pages 251266, Copper Mountain Resort*, 1995.

[32] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility safety and performance in the spin operating system," vol. 29. New York, NY, USA: ACM, December 1995, pp. 267–283. [Online]. Available: http://doi.acm.org/10.1145/224057.224077

[33] C. Small and M. Seltzer, "Vino: an integrated platform for operating systems and database research," Tech. Rep., 1994. [Online]. Available: http://www.dogfish.org/chris/papers/tr-30-94.pdf

[34] D. Probert, J. L. Bruno, and M. Karaorman, "Space: A new approach to operating system abstraction," in *In International Workshop on Object Orientation in Operating Systems*, 1991, pp. 133–137.

[35] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kal, "Namd: biomolecular simulation on thousands of processors," in *Supercomputing Conference*, 2002, pp. 1–18.

[36] J. charles Tournier, P. G. Bridges, A. B. Maccabe, P. M. Widener, Z. Abudayyeh, R. Brightwell, R. Riesen, and T. Hudson, "Towards a framework for dedicated operating systems development in high-end computing systems," *Operating Systems Review*, vol. 40, pp. 16–21, 2006.

[37] S. R. Wheat, A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup, "PUMA: An operating system for massively parallel systems," in *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*. IEEE Computer Society Press, 1994, pp. 56–65.

[38] C. Chang, J. Wawrzynek, and R. W. Brodersen, "Bee2: A high-end reconfigurable computing system," *IEEE Des. Test*, vol. 22, pp. 114–125, March 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1058221.1058286

[39] S. Chakravorty, C. L. Mendes, L. V. Kal, T. Jones, A. Tauferner, T. Inglett, and J. E. Moreira, "Hpc-colony: services and interfaces for very large systems," *Operating Systems Review*, vol. 40, pp. 43–49, 2006.

[40] B. Maccabe, "Forum to address scalable technology for runtime and operating systems," July 2007, http://www.cs.unm.edu/~fastos/.

[41] A. E. haj mahmoud and E. Rotenberg, "Safely exploiting multithreaded processors to tolerate memory latency in real-time systems," in *In Proc. of the 2004 Intl Conf. on*

Compilers, Architecture, and Synthesis for Embedded Systems. ACM Press, 2004, pp. 2–13.

[42] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "Litmusrt : A testbed for empirically comparing real-time multiprocessor schedulers," in *IEEE Real-Time Systems Symposium*, 2006, pp. 111–126.

[43] S. K. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Syst.*, vol. 32, pp. 9–20, February 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1120037.1120055

[44] S. Brosky and S. Rotolo, "Shielded processors: guaranteeing sub-millisecond response in standard linux," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, april 2003, p. 9 pp.

[45] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *Proceedings of the 2008 Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 157–169. [Online]. Available: http://dl.acm.org/citation.cfm?id=1475690.1475899

[46] J. Chen, P. Juang, K. Ko, G. Contreras, D. Penry, R. Rangan, A. Stoler, L. shiuan Peh, and M. Martonosi, "Hardware-modulated parallelism in chip multiprocessors," in *SIGARCH Comput. Archit. News*, 2005.

[47] P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in *in Conference on Hardware/Software codesign and system synthesis of contents*. ACM Press, 2003, pp. 45–51.

[48] E. T. S. Jr., F. R. Wagner, E. P. Freitas, and C. E. Pereira, "Hardware support in a middleware for distributed and real-time embedded applications," in *sbcci*, 2006, pp. 149–154.

[49] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "General purpose timing: the failure of periodic timers," School of Computer Science & Engineering, the Hebrew University, Tech. Rep. 2005-6, Feb. 2005. [Online]. Available: http://www.cs.huji.ac.il/~feit/papers/Timing05TR.pdf

[50] J. Lions, *Lions' commentary on UNIX 6th edition with source code*. San Jose, CA, USA: Peer-to-Peer Communications, Inc., 1996.

[51] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "Effects of clock resolution on the scheduling of interactive and soft real-time processes," in *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '03. New York, NY, USA: ACM, 2003, pp. 172–183. [Online]. Available: http://doi.acm.org/10.1145/781027.781049

[52] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, os clock ticks, and fine-grained parallel applications," in *International Conference on Supercomputing*, 2005, pp. 303–312.

[53] J. Regehr, "Safe and structured use of interrupts in real-time and embedded software." [Online]. Available: http://www.cs.utah.edu/~regehr/papers/interrupt_chapter.pdf

[54] J. Regehr and U. Duongsaa, "Preventing interrupt overload," *Sigplan Notices*, vol. 40, pp. 50–58, 2005.

[55] M. Lewandowski, M. J. Stanovich, T. P. Baker, K. Gopalan, and A.-I. A. Wang, "Modeling device driver effects in real-time schedulability analysis: Study of a network driver," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 57–68. [Online]. Available: http://dl.acm.org/citation.cfm?id=1263546.1263967

[56] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet," in *Proceedings of the 5th annual Linux Showcase & Conference - Volume 5*. Berkeley, CA, USA: USENIX Association, 2001, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268488.1268506

[57] J. Regehr, "Random testing of interrupt-driven software," in *International Workshop on Embedded Systems*, 2005, pp. 290–298.

[58] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt management for real time kernels over conventional pc hardware," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 14–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1128017.1128423

[59] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt scheduling with low overhead for real-time kernels," in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, ser. RTCSA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 385–394. [Online]. Available: http://dx.doi.org/10.1109/RTCSA.2006.51

[60] F. Scheler, W. Hofer, B. Oechslein, R. Pfister, W. Schröder-Preikschat, and D. Lohmann, "Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 167–174. [Online]. Available: http://doi.acm.org/10.1145/1629395.1629419

[61] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "Sloth: Threads as interrupts," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, ser. RTSS

’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 204–213. [Online]. Available: http://dx.doi.org/10.1109/RTSS.2009.18

[62] W. Hofer, D. Lohmann, and W. Schr andder Preikschat, “Sleepy sloth: Threads as interrupts as threads,” in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, 29 2011-dec. 2 2011, pp. 67 –77.

[63] D. Niehaus, R. Menon, F. Ansari, J. Keimig, and A. Sheth, “Utime patch microsecond resolution timers for linux,” http://www.ittc.ku.edu/utime/.

[64] A. C. Heursch and H. Rzeha, “Rapid reaction linux: Linux with low latency and high timing accuracy,” http://www.linuxshowcase.org/full_papers/heursch/heursch.ps.

[65] V. Yodaiken, “The RTLinux Manifesto,” in *Proceedings of The 5th Linux Expo, Raleigh, NC*, Mar. 1999. [Online]. Available: citeseer.ist.psu.edu/yodaiken99rtlinux.html

[66] D. Lazenby, “Timesys linux/rt (professional edition),” *Linux J.*, vol. 2000, Sep. 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=350314.350399

[67] “Rtai - the realtime application interface for linux from diapm,” December 2011, https://www.rtai.org/.

[68] L. Dozio and P. Mantegazza, “Linux Real Time Application Interface (RTAI) in low cost high performance motion control,” *Motion Control*, pp. 27–28, 2003.

[69] J. Ortiz, “Hardware/software co-design of schedulers for real time and embedded systems,” Master’s thesis, University of Kansas, Lawrence, KS, 2004. [Online]. Available: http://www.ittc.ku.edu/research/thesis/documents/jorge_ortiz_thesis.pdf

[70] T. Gleixner and D. Niehaus, “Hrtimers and Beyond: Transforming the Linux Time Subsystems,” in *Proceedings of the Linux Symposium, Ottawa, Canada*, vol. 1, 2006, pp.

333–346. [Online]. Available:
http://www.linuxsymposium.org/2006/linuxsymposium_procv1.pdfANDhttps:
//ols2006.108.redhat.com/reprints/gleixner-reprint.pdf

[71] P. McKenney, "A realtime preemption overview," August 2005,
http://lwn.net/Articles/146861/.

[72] R. Love, "Preempt kernel patches," March 2002,
http://www.kernel.org/pub/linux/kernel/people/rml/preempt-kernel/.

[73] P. Regnier, G. Lima, and L. Barreto, "Evaluation of interrupt handling timeliness in real-time linux operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 52–63, October 2008. [Online]. Available: http://doi.acm.org/10.1145/1453775.1453787

[74] J. Lee, V. John, A. Daleby, K. Ingstrm, T. Klevin, L. Lindh, and V. J. M. III, "A comparison of the rtu hardware rtos with a hardware/software rtos," in *In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC), pages 683688, Kitaskyushu International Conference*, 2003, pp. 683–688.

[75] V. J. M. III and D. M. Blough, "A Hardware-Software Real-Time Operating System Framework for SOCs," *IEEE Design & Test*, vol. 19, no. 6, pp. 44–51, Nov/Dec 2002.

[76] S. Chandra, F. Regazzoni, and M. Lajolo, "Hardware/software partitioning of operating systems: a behavioral synthesis approach," in *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, ser. GLSVLSI '06. New York, NY, USA: ACM, 2006, pp. 324–329. [Online]. Available: http://doi.acm.org/10.1145/1127908.1127983

[77] S. W. Moore and B. T. Graham, "Tagged up/down sorter - a hardware priority queue," *The Computer Journal*, vol. 38, pp. 695–703, 1995.

[78] J. A. Stankovic and K. Ramamritham, "The spring kernel: A new paradigm for real-time systems," *IEEE Software*, vol. 8, pp. 62–72, 1991.

[79] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, and C. C. Weems, "The spring scheduling co-processor: A scheduling accelerator," in *International Conference on Computer Design*, 1993, pp. 140–144.

[80] S. Sez, J. Vila, A. Crespo, and A. Garcia, "A hardware architecture for scheduling complex real-time task sets," *Journal of Computing and Information Technology*, vol. 8, pp. 235–247, 2000.

[81] J. Adomat, J. Furuns, L. Lindh, and J. Strner, "Real-time kernel in hardware rtu: A step towards deterministic and high performance real-time systems," in *Proceedings of EURWRTS 96*, 1996, pp. 164–168.

[82] S.-W. Moon, K. G. Shin, and J. Rexford, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Trans. Comput.*, vol. 49, pp. 1215–1227, November 2000. [Online]. Available: http://dx.doi.org/10.1109/12.895938

[83] P. Kuacharoen, M. Shalan, and V. J. Mooney, "A configurable hardware scheduler for real-time systems," in *Engineering of Reconfigurable Systems and Algorithms*, 2003, pp. 95–101.

[84] B. Zhou, W. Qiu, and C. Peng, "An Operating System Framework for Reconfigurable Systems," in *The 5th International Conference on Computer and Information Technology (CIT)*, 2005, pp. 788–792.

[85] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel," in *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, September 19-22, 2005.

[86] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879 –899, may 2008.

[87] C. Panis, J. Hohl, H. Gruenbacher, and J. Nurmi, "xicu - in interrupt control unit for a configurable dsp core," in *System-on-Chip, 2003. Proceedings. International Symposium on*, nov. 2003, pp. 75 – 78.

[88] S. Han, J. Song, X. Zhu, A. K. Mok, D. Chen, M. Nixon, W. Pratt, and V. Gondhalekar, "Wi-htest: Compliance test tool for real-time wirelesshart tm mesh network devices," 2007.

[89] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden, "Programming models for hybrid fpga-cpu computational components: A missing link," *IEEE Micro*, vol. 24, no. 4, pp. 42–53, 2004.

[90] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit, "A dynamic reconfiguration run-time system," in *Satnam Singh, Mark de Wit The Department of Computing Science The University of Glasgow Glasgow*, 1997.

[91] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," 2000.

[92] H. K. hay So, B. An, O. System, F. based Reconfigurable, and H. K. hay So, "Borph: An operating system for fpgabased reconfigurable computers," Tech. Rep., 2007.

[93] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[94] "Xilinx ml310 documentation and tutorials," December 2011, http://www.xilinx.com/products/boards/ml310/current/.

[95] A. R. Weiss. (2002, Oct.) Dhrystone Benchmark History, Analysis, "Scores" and Recommendations. White Paper.

[96] J.-C. Kim, D. Lee, C.-G. Lee, K. Kim, and E. Y. Ha, “Real-Time Program Execution on NAND Flash Memory for Portable Media Players,” Nov. 2008, pp. 244–255. [Online]. Available: http://dx.doi.org/10.1109/RTSS.2008.27

[97] T. Smalley, “Intel to deliver mids to market within 60 days,” April 2008, http://www.bit-tech.net/news/hardware/2008/04/02/intel_to_deliver_mids_to_market_within_60_days/1.

[98] K. Olukotun and L. Hammond, “The future of microprocessors,” *Queue*, vol. 3, pp. 26–29, September 2005. [Online]. Available: http://doi.acm.org/10.1145/1095408.1095418