Run-Time Scheduling Support for Hybrid CPU/FPGA SoCs

Jason M. Agron

Submitted to the Department of Electrical Engineering & Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master's of Science

Thesis Committee:

Dr. David Andrews: Chairperson

Dr. Perry Alexander

Dr. Ron Sass

Date Defended

The Thesis Committee for Jason M. Agron certifies
That this is the approved version of the following thesis:

Run-Time Scheduling Support for Hybrid CPU/FPGA SoCs

Committee:

Chairperson

Date Approved

Abstract

Minimization of system overhead and jitter is a fundamental challenge in the design and implementation of a Real-Time Operating System (RTOS). Modern FPGA devices, which include (multiple) processor core(s) as diffused IP on the silicon die, provide an excellent platform for embedded systems and offer new opportunities to meet these fundamental RTOS challenges. In particular, it is possible to use the hardware resources of an FPGA to handle scheduling of threads. This paper presents the design of such a scheduler for a multithreaded RTOS kernel built on a hybrid FPGA/CPU system. The scheduler module currently provides FIFO, round-robin, and preemptive-priority scheduling services that coordinate both software-resident threads (SW threads) and threads implemented in programmable logic (HW threads). The design has been implemented and experiments show that the hardware-based scheduler module is able to significantly reduce system overhead and jitter due to the ability of the scheduler to field scheduling requests in parallel with application execution. The scheduler module provides constant time scheduling services for up to 256 active threads with a total of 128 different priority levels, while using uniform APIs for threads requesting OS services from either side of the hardware/software boundary.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems are becoming more sophisticated and universal in our society, and are increasingly relying on Real-Time Operating Systems (RTOS) to deliver precise, predictable, and robust behavior. Two fundamental challenges in the design of RTOS kernels are the minimization of system overhead and jitter [17]. With a shared computational resource such as a CPU, the execution of system services themselves takes away from much needed application processing time. By minimizing the system overhead, more computational cycles are available to the application. Minimizing the jitter in the system allows for more precise, deterministic scheduling of threads which allows the system to respond to real-time events and deadlines in a reliable and repeatable fashion. For any given system, there is a certain amount of jitter that must exist. However, this jitter is usually caused by non-deterministic, asynchronous events such as interrupts, branch mispredictions, and cache misses [11].

This minimum level of system overhead and jitter can be dramatically reduced by careful redesign and migration of portions of the RTOS into concurrent co-designed hardware modules [10, 15, 22]. Migrating these services off the CPU also helps in eliminating the hidden overhead of context switch times associated with entering and exiting the RTOS. Hardware module versions of RTOS components can be configured as specialized co-processors that run concurrently with applications running on the CPU. Perhaps the biggest benefit of this approach is the ability to effectively eliminate the overhead and jitter associated with running the actual RTOS scheduler. The key reason for development of our scheduler module is to service *all* scheduling operations from within the FPGA. This allows the CPU and other computational resources, i.e custom hardware components, to use the scheduler module's services without any burden of executing traditional scheduling operations themselves. The HybridThreads system [3, 5, 6, 18] allows for all computations, whether implemented in hardware or software, to use uniform, high-level APIs provided by the HW/SW co-designed components resident within the fabric of the FPGA. This development methodology has also been used in the context of the HybridThreads project for other RTOS services including semaphores, thread management, and hardware thread control mechanisms.

With a software-based scheduler, a timer-interrupt representing a scheduling event goes off in order to provide a trigger that will eventually force the CPU into an interrupt service routine (ISR). The ISR usually invokes the scheduler via a context switch, the scheduler makes its scheduling decision, and another context switch is done to run the thread selected by the scheduler. A hardware-based scheduler is capable of running in parallel with the CPU, thus granting it the ability to calculate a scheduling decision before a scheduling event (i.e. timer-interrupt or change in the ready-to-run queue) takes place. This greatly reduces the system overhead of scheduling because when a scheduling event takes place, the scheduling decision has already been made, and the ISR routine now just makes a single context switch to invoke the newly selected thread. The HW-based scheduler enables the system to perform pre-scheduling without having to periodically interrupt the CPU. This provides serious performance benefit when considering that HW-based computations in a hybrid FPGA/CPU system may want to be scheduled to run, however these computations do not require any CPU resources to do so. Therefore a HW-based scheduler is capable of acting as a non-intrusive scheduling coprocessor, interrupting the CPU only when necessary for preemption; whereas a normal SW-based scheduler would have to be invoked on the CPU in order to schedule routines that do not even require CPU support.

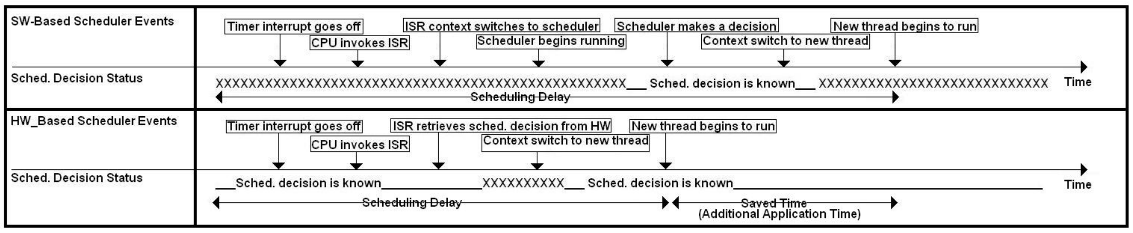

SW-Based Scheduler Events
Timer interrupt goes off
CPU invokes ISR
ISR context switches to scheduler
Scheduler begins running
Scheduler makes a decision
Context switch to new thread
New thread begins to run
Sched. Decision Status
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX___ Sched. decision is known___ XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Time
Scheduling Delay
HW_Based Scheduler Events
Timer interrupt goes off
CPU invokes ISR
ISR retrieves sched. decision from HW
Context switch to new thread
New thread begins to run
Sched. Decision Status
___Sched. decision is known_____________XXXXXXXXXX___ Sched. decision is known___
Time
Scheduling Delay
Saved Time
(Additional Application Time)

Figure 1.1. Scheduling sequences for SW and HW schedulers

The advantage of a hardware-based scheduler in terms of reduction of system overhead is demonstrated in figure 1. This figure contrasts the ordering of events between a system with a parallel hardware scheduler and a traditional software-based scheduler. The act of fielding an interrupt and jumping into an ISR is the same in both types of schedulers, however the actions of the ISRs are quite different. In a traditional SW-based scheduler, the ISR must first invoke the scheduler in order to calculate the next thread to run before it can context switch to that thread. With our HW-based scheduler, the job of the ISR is to read the ID of the next thread to run that has already been calculated by the scheduler module, and then context switch to that thread. Thus, the HW-based scheduler reduces system overhead and jitter by allowing the usage of a simple, deterministic ISR that simply fetches the predetermined scheduling decision from the scheduler module.

Our system does not simply use a HW-based scheduler for just acceleration, but rather for the way that the system is always preparing itself for the next scheduling event, rather than waiting for the scheduling event to occur *and* then preparing itself. This preparedness is demonstrated in figure 1 by the status of the scheduling decision. With a SW-based scheduler, the scheduling decision is not known until it is absolutely required for the context switch to occur. Once the context switch is complete, the scheduling decision for the next event remains unknown until that event actually occurs. With a parallel HW-based scheduler, the scheduling decision is always known, except immediately following the a status change in the ready-to-run queue caused by enqueue and dequeue events. Changes in the ready-to-run queue are immediately followed by re-calculation of the next scheduling decision which readies the system for the next scheduling event before it actually occurs. During a context switch, the act of reading the scheduling decision from the scheduler module signifies that a context switch is currently occurring, so a new scheduling decision needs to be made. A hardware-based scheduler allow for the new scheduling decision to be calculated concurrently with context switch execution. Our HW-based scheduler is capable of calculating a scheduling decision in a shorter amount of time than a comparable SW-based scheduler, but its ability to do so in parallel with CPU execution drastically reduces the variability and absolute amount of scheduling overhead in our system.

The use of a hardware-based scheduler can also reduce the amount of scheduling jitter in an RTOS. The scheduling jitter in an RTOS can be thought of as the unwanted variation in timing for a periodic thread [21]. In an RTOS with a software-based scheduler, scheduling jitter can be introduced by the cache, mispredicted branch instructions, variation in branch instruction length in the scheduler code, and dependence on the number of active threads in the system [21]. The hardware-based scheduler described in this paper is designed as a finite state-machine (FSM) that takes a fixed number of clock cycles to complete, regardless of the number of active threads in the system. The predictability [8] of execution time for scheduling operations along with calculation of scheduling decisions in parallel with the CPU allows for a drastic reduction of scheduling jitter and scheduling overhead and thus provides more precise scheduling within the HybridThreads operating system.

The rest of the paper is organized as follows. The problem statement of this thesis work is developed in chapter 2. The subject of improved precision and predictability of scheduling within an RTOS are discussed in chapter 3. Related works in the subject area are discussed in chapter 4. Chapter 5 describes the iterative redesigns of the scheduler module over the course of my thesis work. The initial design of the scheduler module [2] is discussed in section 5.1. The second redesign of the scheduler module is covered in section 5.2, which includes the work done to further reduce scheduling overhead and jitter. Section 5.3 covers the final scheduler module redesign which extends thread management and scheduling services to both hardware and software-based threads in the HybridThreads system. Performance results of the various versions of the scheduler module, both as an independent entity and actual integrated performance measurements are shown in chapter 6. The evaluation of the design and implementation of the scheduler module, and comments on future work to be done on the scheduler module and other components within HybridThreads, a multithreaded RTOS kernel [3, 5, 6, 18, 20], are discussed in chapter 7.

Chapter 2

Statement of Problem

2.1 Programmability of Hybrid Systems

Hybrid CPU/FPGA systems form a union between two different areas of expertise: hardware design and software programming. These two fields have began to converge with the advent of modern high-level synthesis (HLS) tools that allow hardware to be "programmed" in a fashion similar to that of software. For decades it was the job of electrical and computer engineers to build digital systems. The designers of these systems worked at very low levels of abstraction, with RTL being the highest. Compilers are then required in order to make the hardware accessible by traditional software programmers working at the abstraction level of high-level languages (HLLs). A traditional software-compiler breaks the HLL constructs down into smaller portions easily translatable to the instruction-set, or the abstraction level, of the hardware itself. This is the point at which FPGAs differ from a typical programmable microprocessor. The functionality of a microprocessor is abstracted by an instruction-set architecture (ISA), but what is the abstraction level of an FPGA? Current practices use netlists to describe what components should be used and describes how these components are connected within the fabric of an FPGA at an RTL level. Traditional software-compilers take a program that describes the functionality of an algorithm as input, and map that algorithm onto a fixed architecture known during compile-time. HLS compilers take a program that describes the architecture of an algorithm, either structurally or behaviorally, as well as the functionality of the algorithm and produce a netlist that represents the datapath and the control flow of the source program that is essentially at the same abstraction level as RTL [9]. This shows why HLS tools have such a higher complexity that that of traditional software-compilers: software-compilers can target an ISA which is at a higher abstraction level than that of RTL, while HLS tools must target the RTL level themselves. The additional complexity of HLS techniques for FPGAs has made it extremely difficult for HLLs to be translated into hardware - thus requiring the designer of a hybrid system to have expertise in both software and hardware design techniques.

2.2 Uniformity of Services in a Hybrid System

FPGAs provide a ”blank” slate of computational components. Standard interfaces and services have not yet been established, thus forcing hybrid system designers to literally build systems from ”scratch”. On the converse, many common interfaces and services have been established in the world of software, and these have been standardized in the form of operating systems (OSes). The purpose of an operating system is to provide common services to the user in a convenient and efficient manner [25]. Operating systems often work as an intermediary that manages the lower-level details of the hardware in a system. An OS provides services that enforce how programs, or more abstractly, computations, execute and communicate. Operating systems typically provide the idea of a task in the form of threads or processes. Traditionally, tasks are defined in software, and are able to execute, usually on a set of CPUs, in a pseudo-concurrent fashion. Additionally, operating systems are typically implemented solely in software, which means that the OS requires CPU-execution time in order to provide services to tasks executing in the system. Hybrid CPU/FPGA systems provide the ability for tasks to execute in either hardware or software. This means that OS services must be uniformly accessible to both hardware and software-based computations. If an OS is solely implemented in software, management of hardware-based computations will require CPU time to execute, therefore software tasks must be interrupted in order to field calls to the OS coming from hardware tasks. This is a very inefficient method of organizing OS services that must be uniformly accessible by all tasks in a hybrid system. In order to provide a uniform service interface for both software and hardware-resident tasks the OS must not be tied to a shared computational component (i.e. CPU) that the OS also relies on for task execution. This means that an additional CPU can be used in a system as an OS "coprocessor"; a CPU on which the OS runs and all OS service calls are directed towards it. This approach does indeed provide a uniform service interface for both software and hardware-based tasks, however the allocation of a microprocessor used solely for OS service execution is quite wasteful in terms of hardware utilization and does not allow for full exploitation of parallelism within OS services. Instead a specialized OS coprocessor capable of all typical OS functions could be developed. This would allow the ISA of the OS coprocessor to be tailored to perfectly suit OS services, thus raising the abstraction-level of the hardware itself to that of the level of an OS. Additionally, what if each component of the OS is implemented as a separate hardware component within an FPGA; allowing each portion of the OS to execute in parallel with application execution? This would provide uniform OS services to both software and hardware resident tasks, however the concurrent execution of different OS services allows for a large reduction in system overhead and jitter without the need of a general-purpose processor to perform OS services. Additionally, the parallel nature of FPGAs allow for the exploitation of both coarse-grained (task-level) and fine-grained (instruction-level) parallelism. Therefore the operations within each OS component can be parallelized, thus improving overall system performance by reducing the overhead and jitter normally incurred by executing these OS services on a sequential CPU [19]. For instance, bit-shifting and boolean arithmetic are highly sequential operations when executed on a CPU, but are highly parallel operations that often require little or no intermediate steps (states) when executing within an FPGA.

2.3 Contributions of this Thesis

The main goal of this work is to develop IP cores capable of providing run-time scheduling services to all threads, both software and hardware resident, in a hybrid CPU/FPGA system. Additionally, the scheduling services are to be designed in such a way that maximizes parallelism within the OS and within the components that comprise the scheduler IP itself. This will hopefully result in a massive reduction in the amount of overhead and jitter normally incurred through executing traditional OS scheduling services on a CPU, thus producing a more deterministic and accurate OS. Additionally, the scheduling services resident within the FPGA will provide support for standard scheduling services, thus improving accessibility to the FPGA fabric in a general purpose fashion. The run-time scheduling services also provide a platform in which hybrid FPGA/CPU systems can be viewed as standard multi-threaded environments in which threads are not restricted to only running in software. Hybrid CPU/FPGA environments allow for threads can be implemented in either hardware or software, and the OS services implemented within the FPGA provide all of the standard mechanisms and interfaces for communication and synchronization; thus allowing the system designer to focus on the application program instead of worrying about providing standard support services that an application may need in order to execute. The accessibility and programmability of hybrid systems can be greatly increased by providing IP cores capable of performing operating system services to computations based in either hardware, software, or a combination of the two. The main goal for this thesis work is to provide scheduling services for threads resident in either hardware or software that are accessible through uniform, high-level APIs. Additionally, the design of the scheduling services themselves should provide their functionality with low overhead and jitter in order to increase the predictability of the HybridThreads operating system.

Chapter 3

Background

There are both obvious short-term performance advantages, but more importantly subtle significant long-term advantages that can be gained by migrating an operating system scheduler into hardware. First, migration of functionality from software to hardware should result in decreased execution times associated with the targeted software-based scheduling methods [1]. Iterative control loops, used for searching through priority lists, can easily be decomposed into combinational circuits controlled by finite state machines that perform parallel sorting using fast decoders in a fixed number of clock cycles. Additionally, the parallel sorting hardware can run concurrently with an application program resulting in a significant reduction in overhead and jitter. The literature reports many examples of a hardware/software co-designed scheduler providing lower overhead and jitter than that of a traditional software-based scheduler. As a good example, [10] reports a hardware scheduler implementation that incurs zero overhead for a hardware-based scheduler that runs in parallel with the currently running application. Although an important result, these short-term advantages only minimize the overhead and jitter associated with the scheduling method itself and do not dramatically affect the precision of the entire system overall. In fact, existing software techniques for systems that require precise scheduling times can minimize the overhead and jitter by calculating overhead delay times and presetting the next event timer to expire a set number of timer ticks before the next event should be scheduled.

A second subtle but more significant advantage a hardware-based scheduler offers is the ability to modify the semantics of traditional asynchronous invocation mechanisms that introduce the majority of system jitter. By creating a hardware component that manages the scheduling of threads, the asynchronous invocations can be directed to the scheduler and not the CPU. From a system perspective, this has the positive effect of resolving the relative inconsistencies that exist between the scheduling relationships of the application threads and the ability of external interrupt requests from superseding and perturbing this relationship. Resolving these inconsistencies can be achieved by translating interrupt requests into thread scheduling requests directed towards the hardware-based scheduler. The external interrupt device is simply viewed as requesting a scheduling decision for a device handler "thread" relative to all other threads. These device handler "threads" often do the work of interrupt service routines (ISRs), so these "threads" are often referred to as interrupt service threads (ISTs) [12]. This use of ISTs allows the scheduler to consider the complete state of the system in determining when an interrupt request should be handled because ISTs and user-level threads can be viewed as falling in the same class of priorities.

This approach also results in a changing of the order of the traditional scheduler invocation mechanism itself, consisting of a timer interrupt expiring, followed by the execution of the scheduler. Under this new approach, the timer interrupt request is simply an event, similar to the unblocking of a semaphore, that is directed to the scheduler and factored into the scheduling decision. As the scheduling decision should have been made before the timer interrupt expires, a software scheduler interrupt routine then reduces down to a simple context switching routine with the next thread to be scheduled already known. More far reaching is this approaches alteration of the semantics of current software-based operating systems that must allow external devices to interrupt an application program in order to determine the necessity of servicing the request. By eliminating the need to perform a context switch for every possible interrupt request, the jitter associated with non-deterministic invocations of an ISR to determine if an interrupt request should be processed or simply marked as pending with control returned back to the original application program is eliminated. These capabilities can only occur if the scheduler is truly independent of the CPU and the interrupt interfaces are transformed into scheduling requests. Additionally, traditional interrupt semantics treat interrupt requests as threads with a priority level of infinity, while the HybridThreads approach transforms interrupt requests into ISTs that have priority levels falling within the ranges of typical user-level threads. Thus allowing the system to take into account the importance of interrupt requests in a framework that treats all threads in the system as equals.

This new approach should provide a significant increase in scheduling precision, and reduction of overall system jitter over current software-based methods that must rely on interrupts. Most real-time operating systems, such as RTLinux [28], attempt to minimize the jitter introduced by asynchronous requests by preprocessing external requests within a small, fast interrupt service routine that determines if the request should be immediately handled, or can simply be marked as pending for future service. While this approach does reduce jitter, it is still based on the semantics of allowing asynchronous invocations to interrupt the CPU, and incurring the overhead of a context switch to an interrupt service routine.

Chapter 4

Related Works

The migration of scheduling functionality into hardware is not a a new technique in the world of real-time and embedded systems as shown in [1, 14, 23]. All of these approaches use a high-performance systolic array structure to implement their ready-to-run queues in a similar fashion as the structure described in [16]. Systolic arrays are highly scalable through the process of cell concatenation due to the uniform structure of their cells. However each cell in the array, as shown in figure 4.1, is composed of storage registers, comparators, multiplexers, and control logic; which requires a non-trivial amount of logic resources to implement. Systolic arrays enable parallelized sorting of entries in a priority queue, however this comes in the form of high cost in terms of logic resources within an FPGA. In a system such as HybridThreads, operating system components as well as user-defined hardware threads must share the logic resources of an FPGA, therefore conservation of logic resources within each operating system component (such as the scheduler) becomes an issue. Logic resources within operating system components must be minimized in order to maximize the amount of logic resources available for user-defined hardware threads. Although systolic array cells are fast and allow parallel accesses, they take up a considerable amount of space within an FPGA. This is evident in Georgia Tech's hardware-based scheduler [14], where a systolic array implementation of a ready-to-run queue structure requires 421 logic elements (slices) and 564 registers for a queue size of only 16 entries. In the HybridThreads system, on chip BRAMs are used to store data structures, such as the scheduler module's ready-to-run queue, in order to conserve logic resources (slices, registers, etc.). The first implementation of the scheduler for the HybridThreads system, described in sections 5.2 and 6.2 only requires 484 logic slices and 573 flip-flops for a queue size of 256. BRAMs allows for excellent scalability of queue size with minimum effects on logic resource usage. Although more BRAMs might be used to increase the size of the ready-to-run queue, only slightly more logic resources are used for decode logic and pointer registers. Additionally, BRAM access times are almost on par with that of registers; only requiring 1 clock cycle for writes and 2 clock cycles for reads.

The HybridThreads scheduler differs from other existing hardware-based schedulers in that it must be able to perform scheduling operations for both software and hardware threads. This requires the HybridThreads scheduler to incorporate new properties that distinguish between hardware and software threads, as well as different mechanisms that handle the scheduling of each type of thread.

Additionally, the entire HybridThreads system is built around APIs that are fully compatible with the POSIX thread (pthread) standard [7]. Both Malardalen University's Real-Time Unit (RTU) and Georgia Tech's configurable hardware scheduler [14] use their own custom APIs for interacting with OS services. Using POSIX thread compatible APIs has enabled the HybridThreads system to be extremely accessible to those familiar with the POSIX thread standard. Addition-

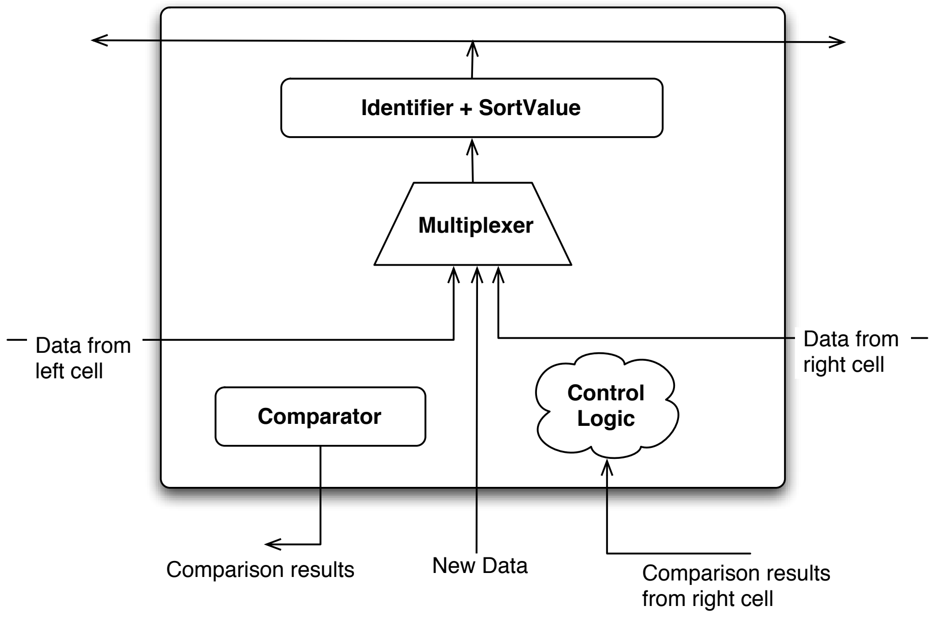


Figure 4.1. Structure of Typical Systolic Cell

ally, simple library wrappers can be used to port applications written for use on the HybridThreads system to a POSIX thread application capable of running on a desktop computer equipped with POSIX thread libraries. This allows for rapid prototypes of HybridThreads applications to be debugged on a typical desktop machine, and then later ported for use on the HybridThreads architecture at no cost to the system programmer.

Chapter 5

Design & Architecture

5.1 First Scheduler Redesign

The initial scheduler in the HybridThreads system used a simple FIFO scheduling mechanism that was internal to the Thread Manager (TM) [4]. The add_thread system call would be routed to the Thread Manager and the TM would then insert the thread into a singly-linked FIFO ready-to-run queue. The main problem with this scheduling structure is that the ready-to-run queue was built in to the Thread Manager's attribute structures used for data storage for thread management status. This means that thread management (allocation, creation, and control) and thread scheduling (ready-to-run queue management, and scheduling decisions) activities could not occur in parallel. Additionally, any changes in scheduling mechanisms and scheduling data arrangement would also affect the management mechanisms, and vice-versa, so maintenance of the management and scheduling services would be difficult, cumbersome, and error-prone.

Both the thread management and thread scheduling mechanisms would have to be modified if either were to be extended in their functionality or data storage requirements. The scheduling mechanism was going to be upgraded to allow for priority scheduling and eventually would have to handle the scheduling of both SW and HW threads, so it was decided to make the scheduler a separate IP module that would have its own interface to the HybridThreads system bus. Many thread management operations result in the invocation of scheduling operations, so essentially the TM uses the scheduler module as a coprocessor for any and all scheduling operations. Many of these coprocessor operations can only occur as a result of a management operation so the TM will always be the ”caller” in these cases. This, in conjunction with the scheduler becoming a separate module, means that all outgoing operations from the TM to the scheduler will result in a bus operation; however if the TM is using the scheduler to complete a management operation, then the bus will already be locked by the caller of the TM operation. This meant that a special interface must be created between the TM and the new scheduler module to allow access to scheduling operations while the system bus was locked. Additionally, since other scheduler specific operations are not ever called as the result of a thread management operation, then these scheduling operations can be called via the new interface used to attach the independent scheduler module to the HybridThreads system bus.

It was decided that the new scheduler module must be able to schedule threads according to priority-levels to allow the user to tailor the order in which threads in the system run. This meant that APIs must be defined to allow the user to both change and read priority-levels for threads. Additionally, APIs are required to add threads to the ready-to-run queue, remove threads from the ready-to-run queue, and to calculate a new scheduling decision. The addition, removal, and scheduling decision operations are only called as the result of thread management operations, so these operations will be callable through the TMcom interface. Operations used to change and read thread's priority-levels are callable from any user thread, therefore these operations will be callable through the bus interface which connects the scheduler module to the HybridThreads system bus. The architecture of the new scheduler module consists of the TM command (TMcom) interface, the Bus command (BUScom) interface, as well as a single ready-to-run queue as shown in figure 5.1.

The TMcom interface is a dedicated hardware interface between the scheduler module and the TM that consists of a total of seven control and data signals as well as access to a read-only interface (B-port) of the TM's Block RAM (BRAM). The data signals include `Next_Thread_ID`, `Current_Thread_ID`, and `Thread_ID_2_Sched`. The `Next_Thread_ID` signal represents the identifier of the thread chosen to run next on the CPU. This signal is writable by the scheduler module and readable by the TM. The `Current_Thread_ID` signal represents the identifier of the thread that is currently running on the CPU (PowerPC 405). This signal is readable by the scheduler module and writable by the TM. The `Thread_ID_2_Sched` signal contains the identifier of the thread that is being added to the ready-to-run queue by the TM. This signal is readable by the scheduler and writable by the TM. The control signals include `Next_Thread_Valid`, `Dequeue_Request`, `Enqueue_Request`, and `Enqueue_Busy`. The `Next_Thread_Valid` signal represents whether or not that the scheduling decision available from the scheduler module on the `Next_Thread_ID` signal is valid or not (Valid = 1, Invalid = 0). This signal is writable by the scheduler module and readable by the TM. The `Dequeue_Request` signal is used by the TM to request the scheduler module to perform a dequeue operation of the thread whose identifier is on the `Next_Thread_ID` signal. This signal is readable by the scheduler module and writable by the TM. The `Enqueue_Request` signal is used by the TM to request the scheduler module to perform an enqueue operation of the thread whose identifier is on the `Thread_ID_2_Sched` signal. This signal is readable by the scheduler module and writable by the TM. The `Enqueue_Busy` signal represents whether or not the scheduler is currently busy performing an enqueue operation (Busy = 1, Not Busy = 0). This signal is writable by the scheduler module and readable by the TM. The B-Port interface to the TM's BRAM allows the scheduler module to query thread management information in order to perform error-checking that concerns the parent-child relationships of threads that the TM's data structures hold. The purpose of the TMcom interface is to allow the TM to request scheduling operations as a result of thread management operations whose side-effects alter the scheduling status of the system (i.e. the status of the ready-to-run queue). The operations available through the TMcom interface can be seen in table 5.1.

Table 5.1. Command Set of Scheduler Module, Build 1

Type	**Name**	**Actions**
TMcom	Enqueue	Schedules a thread
TMcom	Dequeue	Removes a thread from the ready-to-run queue
BUScom	Get_Entry	Returns a thread's table attribute entry
BUScom	Toggle_Preemption	Toggle preemption interrupt on/off
BUScom	Get_Entry	Returns a thread's table attribute entry (for debug use)
BUScom	Get_Priority	Returns the priority-level of a thread
BUScom	Set_Priority	Sets the priority-level of a thread
BUScom	Set_Default_Priority	Sets the priority-level of a thread (no error-checking)

The BUScom interface allows the services provided by the scheduler module to be accessed through bus transactions on the HybridThreads system bus (IBM

CoreConnect On-Chip Peripheral Bus, OPB). This interface consists of an OPB-IPIF attachment that is configured to be a slave on the bus. Scheduling operations accessible over the BUScom interface are memory-mapped in the address space of the scheduler module in the form of opcodes and parameter fields encoded in the address and data lines of the bus. All BUScom operations are atomic in that they keep the bus in a locked state until completely finished. This is done by suppressing bus timeouts and making a guarantee that the scheduler module will not acknowledge (ACK) the bus until its operation is complete. The operations available through the BUScom interface can be seen in table 5.1.

During the design phase of the ready-to-run queue (R2RQ) there was much debate as to whether the R2RQ should be kept in FIFO or sorted order. A ready-to-run queue kept in sorted order allows for quick scheduling decisions to be made because the highest-priority entry is kept at the head of the queue, however all enqueue operations require a sorted insert operation that involves list traversal. A FIFO ordered ready-to-run queue provides quick enqueue operations (adding to the tail of the queue), however scheduling decisions require list traversal to find the queue element with the highest priority-level. An enqueue operation occurs as the result of an add_thread operation to the Thread Manager, which occurs during the time in which a thread is doing useful work (i.e. not context switching). A state-machine diagram of the enqueue operation in a FIFO ordered queue can be seen in figure 5.3.

A dequeue operation occurs as the result of a scheduling decision being used: the next thread scheduled to run has just become the running thread, thus removing it from the ready-to-run queue and calculating a new scheduling decisions as a result. The dequeue operation actually occurs as the result of a context switch,

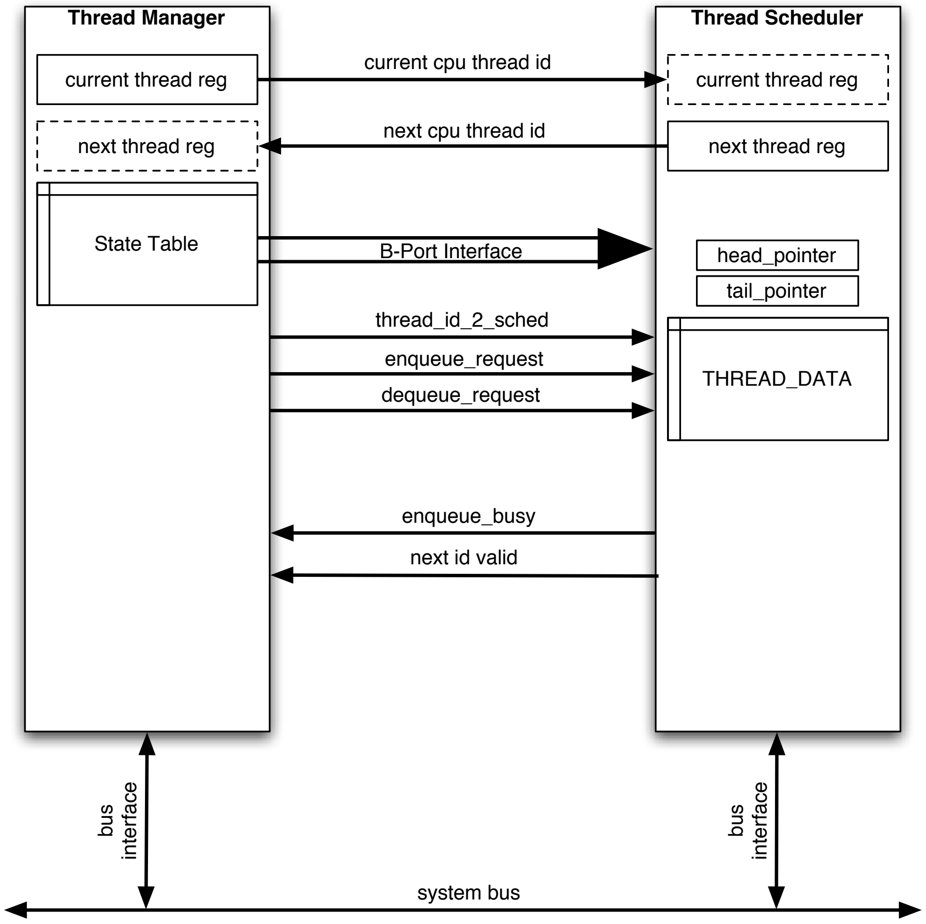


Figure 5.1. Block Diagram of Scheduler Module, Build 1

Thread_Data BRAM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Q?	N0	N1	N2	N3	N4	N5	N6	N7	L0	L1	L2	L3	L4	L5	L6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Field	**Width**	**Purpose**
Q?	(1-bit)	1 = Queued, 0 = Not Queued.
N0:N7	(8-bit)	Ready-to-run queue next pointer
L0:L6	(7-bit)	Scheduling priority-level

Figure 5.2. Scheduler Attribute Table Entry Format, Build 1

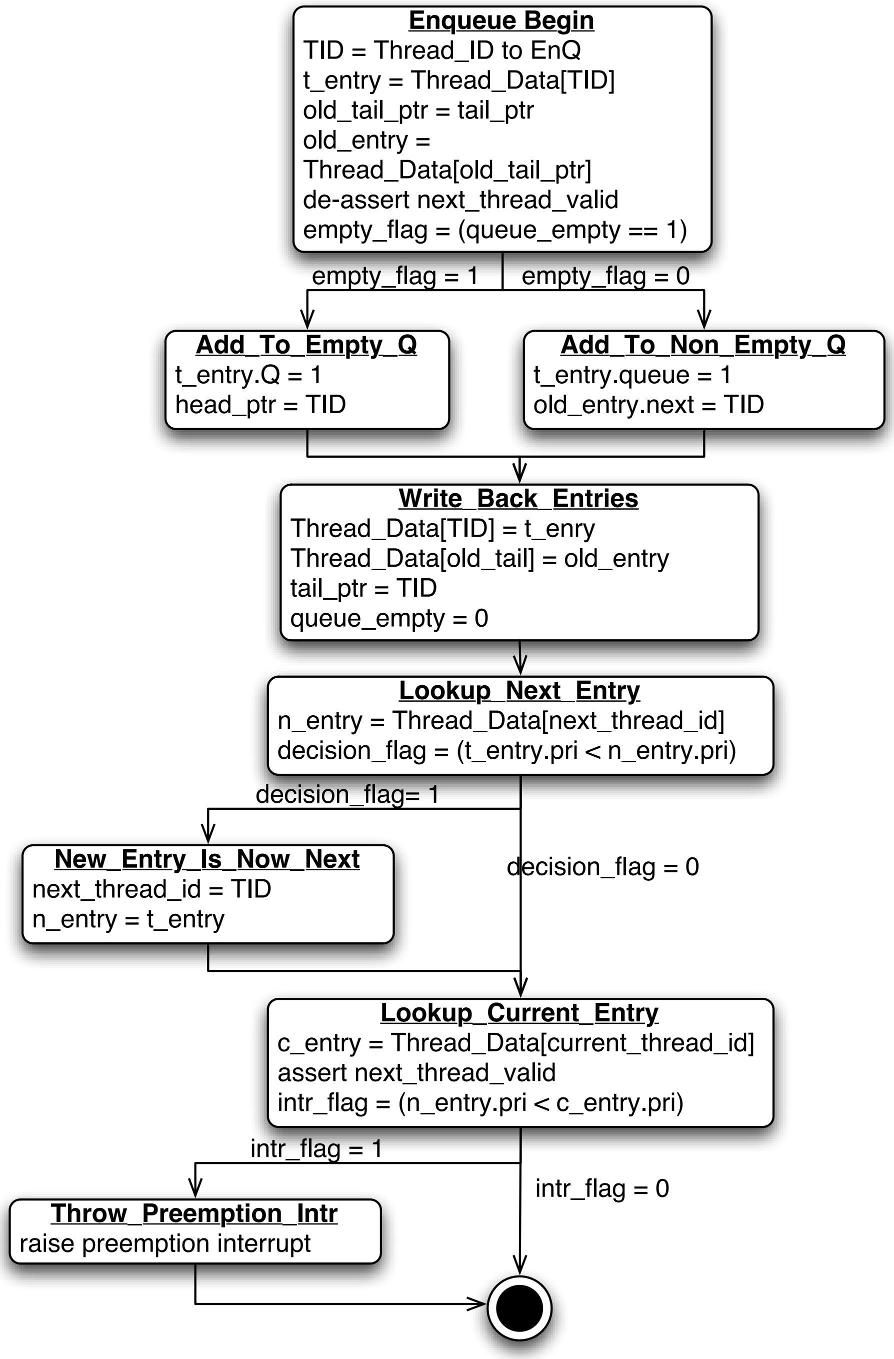


Figure 5.3. State Diagram for Enqueue Operation, Build 1

and is able to be carried out in parallel with setting up a new thread to run on the CPU; therefore allowing the dequeue operation to have higher overhead without actually resulting in any degradation of system performance. This means that a fast enqueue operation paired with a slower dequeue operation represents the best balance in scheduling performance and can be achieved through the usage of a ready-to-run queue implemented as a linked-list kept in FIFO order. A FIFO ordered ready-to-run queue structure is implemented as a singly-linked list data structure resident in on-chip BRAMs as shown in figure 5.2. Additionally, a new scheduling decision must be made when a dequeue operation occurs, so the entire ready-to-run queue must be traversed to find the highest priority-level thread in the system. The currently running thread, which is also the thread to be dequeued, is simply "unlinked" from the ready-to-run queue during this traversal. Therefore only a single, linear queue-traversal, O(n), is used to find the next scheduling decision and to remove the currently running thread from the queue. A state-machine diagram of the dequeue operation showing the traversal of the data structure is shown in figure 5.4.

The scheduler module can constantly monitor the priorities of the threads in the ready-to-run queue as well as the priority-level of the thread currently running on the CPU, so it can determine exactly when the running thread should be preempted. Additionally, it can do so without introducing any scheduling overhead that could adversely affect the execution time of application-threads running on the CPU because it is an independent module executing within the reconfigurable fabric of an FPGA. Preemption occurs only when a thread enters the ready-to-run queue that has a higher-priority level than the currently chosen next thread to run and the currently running thread. This preemption mechanism does not need-

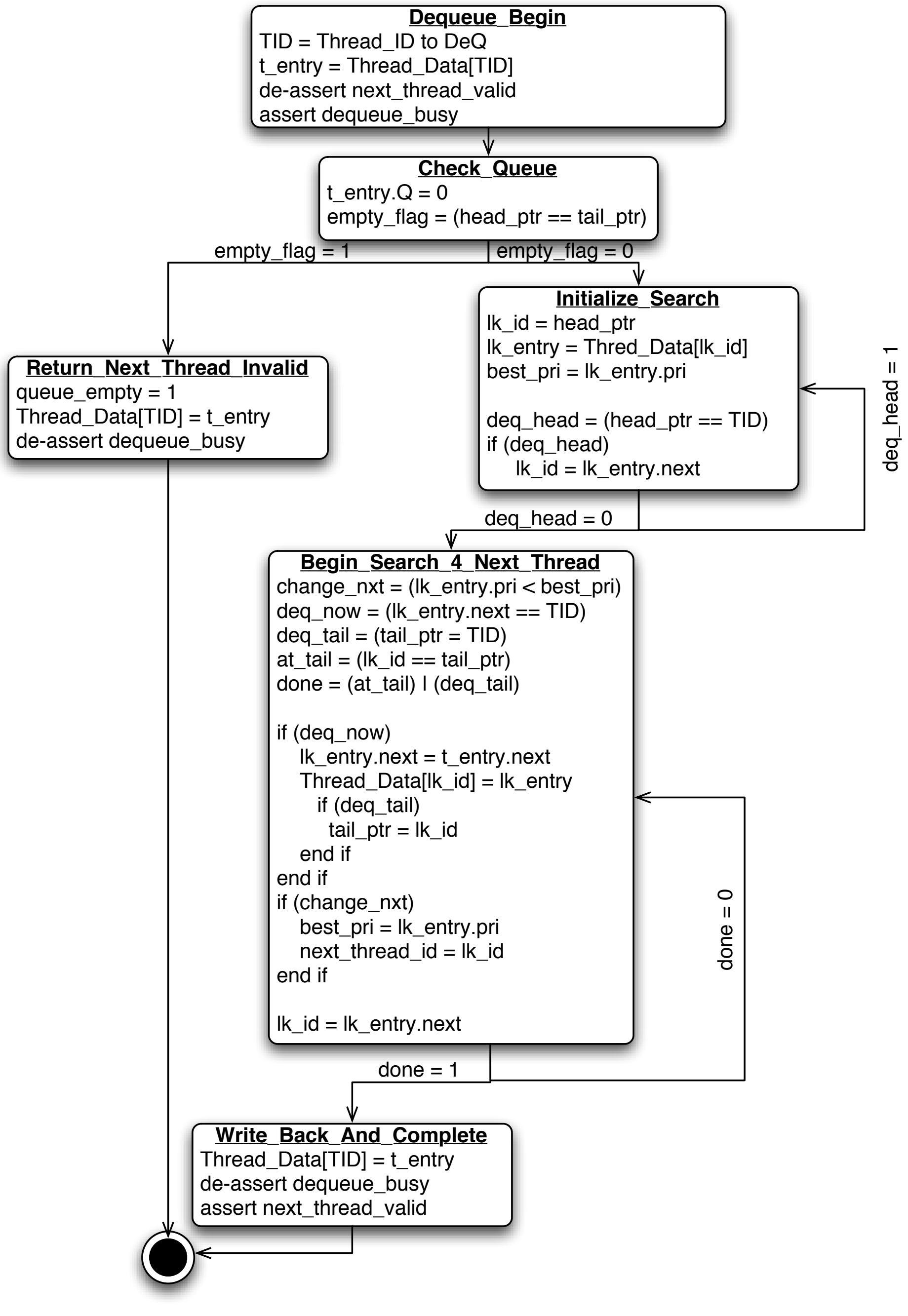
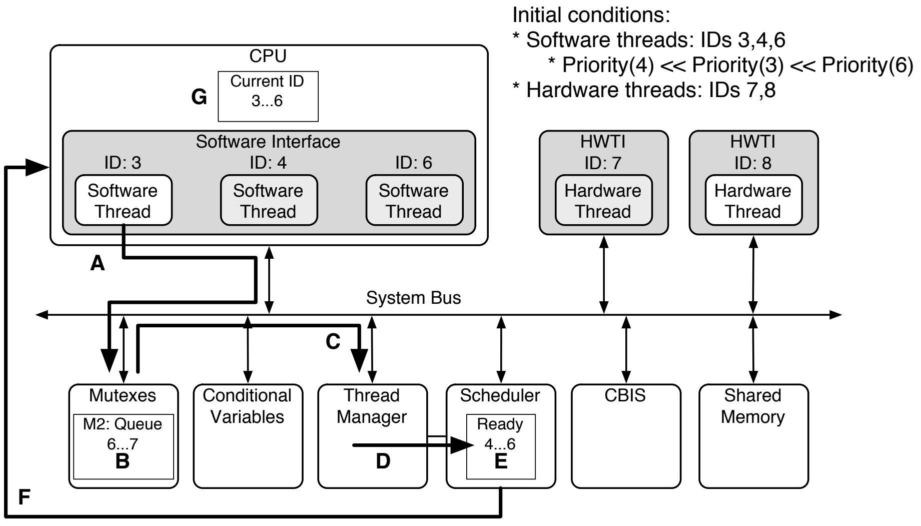


Figure 5.4. State Diagram for Dequeue Operation, Build 1

lessly interrupt the CPU in cases where a context switch is not needed, therefore eliminating jitter usually found in periodically checking to see if the currently running thread should be preempted. An example of the preemption process being invoked is shown in figure 5.5. It shows that the scheduler decides when the CPU should be preempted, unlike a system with a SW-based scheduler that requires a periodic CPU interrupt during application execution used to update scheduling decisions.



A: Software thread 3 unlocks mutex M2 by calling hthread_mutex_unlock(M2), which sends signal to Mutex Manager.
B: Mutex Manager inspects M2's queue and decides ID 6 will own mutex next.
C: Mutex Manager sends add_thread(6) to Thread Manager.
D: Thread Manager sends enqueue(6) to Thread Scheduler.
E: Thread Scheduler adds ID 6 to ready-to-run queue and makes new scheduling decision - ID 6 has highest priority in the system.
F: Thread Scheduler sends preemption interrupt to the CPU.
G: CPU receives preemption interrupt and context switches to software thread ID 6 from software thread ID 3.

Figure 5.5. Scheduler Preemption Example

Once the architecture of the scheduler module was decided upon, it was then implemented in VHDL and incorporated into the HybridThreads system. The

Thread Manager was modified slightly so that it would interact with the new independent scheduler module through the TMcom interface instead of using its previous built-in FIFO scheduling mechanism. The scheduler module was first tested in a ModelSim simulation environment in which the functionality of the module was verified, and then performance figures were collected. Synthesis tests were then conducted using the Xilinx EDK tool set (XPS + ISE) and the resulting bit streams were tested on Xilinx Virtex-II Pro FPGAs. The timing results for the simulation and synthesis tests of the O(n) scheduler module are shown in chapter 6.

5.2 Second Scheduler Redesign

The main goal of the second redesign of the scheduler module is to further reduce the amount of overhead and jitter involved in thread scheduling operations. The first scheduler module, although fast, still required a variable amount of time for it to make scheduling decisions. This variability in operation execution time introduces unwanted jitter into applications running on top of the HybridThreads operating system which can be very harmful to real-time applications in which deterministicity of program execution times is required. The first build of the scheduler module established an interface for all scheduling operations to occur as well as added priority scheduling to the system. The second build would adhere to the interface designed during the first build, however the internal structure of the ready-to-run queue would be altered to allow for the reduction of overhead and jitter involved in all scheduling operations.

The first scheduler module contains a ready-to-run queue composed of a single linked-list that was kept in FIFO order. Enqueue operations for a structure such as this simply add a thread entry to the tail of the list, which can be done in a fixed amount of time. Scheduling decisions, invoked via dequeue operations, in this configuration require a single traversal of the ready-to-run queue from head to tail which occurs in O(n) time, where n is the number of entries in the ready-to-run queue. If the single linked-list was kept in sorted order the complexity of the operations would reverse: dequeue operations would take a constant amount of time, while enqueue operations would require queue-traversal to find where the entry should be inserted. A ready-to-run queue structure with constant time execution for enqueue and dequeue operations is needed for optimal performance in terms of reduction of scheduling overhead and jitter.

A O(1) ready-to-run queue structure would allow for the elimination of all jitter at the hardware execution level of the scheduler module. One such ready-to-run queue structure contains multiple queues, one for each priority-level, paired with some sort of decision module that can calculate the highest priority-level in the system is capable of constant time execution of enqueue and dequeue operations, assuming that the decision module is also O(1). This type of structure assumes that threads within the same priority-level can be thought of as being in the same "class" and can therefore be scheduled in FIFO order. With this assumption, each priority-levels queue can be kept in FIFO order which allows for constant time execution of enqueue and dequeue operations. Enqueue operations simply add an entry to the tail of the queue, while dequeue operations just remove an entry from the head of the queue. Now the scheduling decision can be made by finding the highest priority-level in the system, which is handled by the decision module, and then dequeuing the entry at the head of that priority-level's queue. In our design, the decision module would be implemented as a priority encoder capable of executing in constant time. The input to the priority encoder is a bit field in which each bit represents whether or not a priority-level has any threads resident in its queue. The priority encoder's output represents the highest priority-level in which a thread is queued.

The ready-to-run queue within the scheduler module was first implemented as a single linked-list kept in FIFO order [2] described in section 5.1. This implementation requires that the entire queue be traversed in order to find the highest priority thread in the system, thus making the execution times of scheduling operations vary directly with the number of active threads. The second scheduler module redesign produced a partitioned ready-to-run queue with a parallel priority encoder in order to provide constant time scheduling operations without adding significant complexity to the scheduler module. A system-level block diagram of this scheduler architecture is shown in figure 5.6. This implementation uses two Block RAMs (BRAMs) to implement the ready-to-run queue: the `Priority_Data` BRAM and the `Thread_Data` BRAM. The Priority_Data BRAM is indexed by priority value, and contains the head and tail pointers for the queue for each individual priority level. The `Thread_Data` BRAM is indexed by thread ID, and contains the thread attribute information described in figure 5.8. Additionally, a parallel priority encoder calculates the highest priority level in the system using a 128-bit register field that represents which priority levels have active (queued) threads associated with them. The parallel priority encoder calculates the highest priority in the system only when a change occurs in its 128-bit input register and can do so in a quick and predictable four clock cycles. The parallel priority encoder shown in figure 5.7 is implemented as an FSM combined with a 32-bit priority-encoder. The FSM splits the 128-bit input register into four 32-bit chunks, and in parallel, checks to see which "most significant" chunk has a non-zero value. The "most significant" chunk with a non-zero value is then used as input to the 32-bit priority encoder. The output of the 32-bit priority encoder is then concatenated with the chunk number to then form the highest active priority-level in the system. The partitioned ready-to-run queue along with the priority encoder eliminate the need to traverse the scheduler module's data structures because individual priority queues can be located using the `Priority_Data` BRAM and individual thread information can be located using the `Thread_Data` BRAM. The data formats of the `Priority_Data` and `Thread_Data` BRAMs can be seen in figure 5.8.

With this data organization, enqueue operations work by first adding a thread to the tail of the priority-levels queue in which that thread resides, and adjusting the priority field to show that this priority-level does indeed have active (queued) threads in it. Next the scheduling decision is adjusted by checking the output of the parallel priority encoder, and looking up the head pointer of the queue associated with the encoder's output. The behavior of the enqueue operation can be seen in figure 5.9. Dequeue operations work by removing a thread from the head of the highest priority-levels queue, and then immediately calculating the next scheduling decision. The behavior of the dequeue operation can be seen in figure 5.10. New scheduling decisions are simply calculated by looking up the head pointer of the highest priority-levels queue from the `Priority_Data` BRAM using the output of the priority encoder and placing the head pointer's thread identifier into the `SCH2TM_next_cpu_tid` register. From figure 5.9 and figure 5.10, one can see that the partitioned ready-to-run queue along with the priority encoder allow for extremely simple enqueue and dequeue semantics that are able to operate in constant time regardless of queue length.

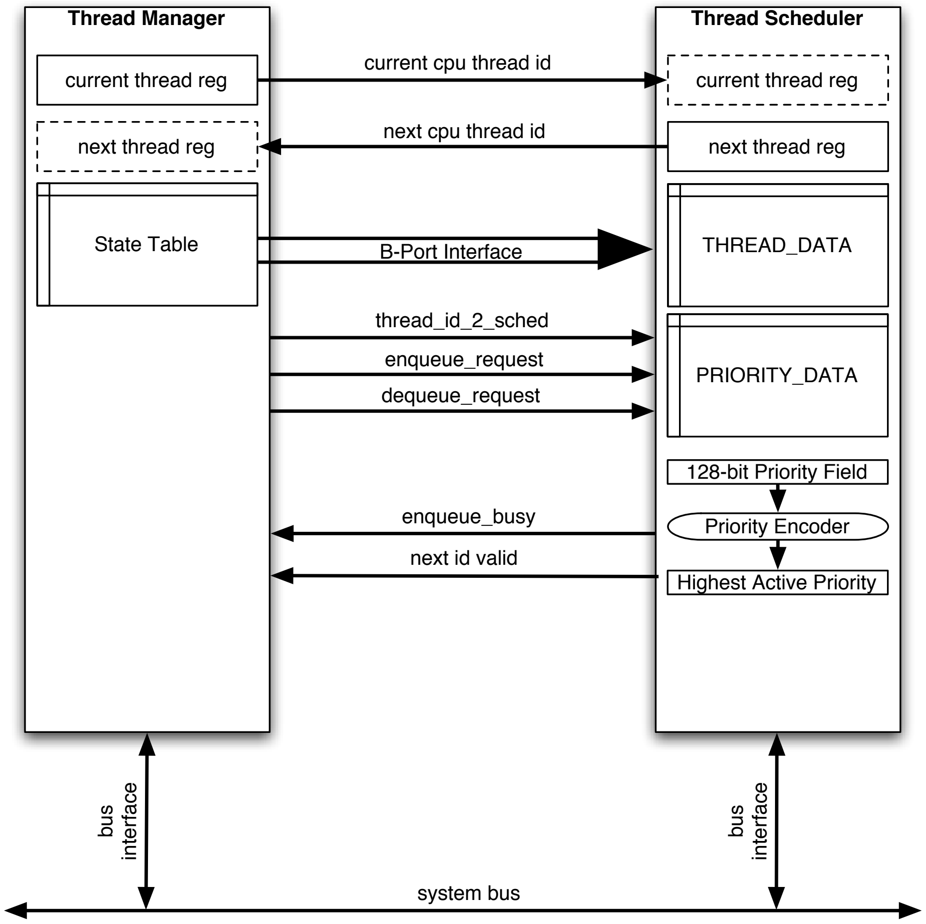


Figure 5.6. Block Diagram of Scheduler Module, Build 2

The new O(1) ready-to-run queue structures were fully integrated into the existing scheduler module. Next the scheduler module was subjected to a variety of simulation and synthesis tests to verify correct functionality and to test the performance of the new O(1) ready-to-run queue structure. The timing results for these tests can be found in chapter 6.

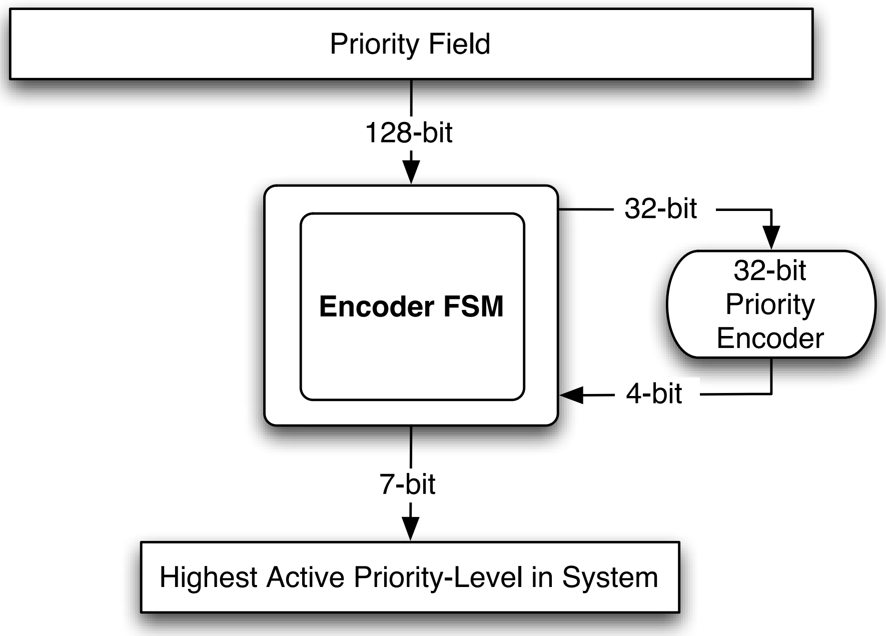


Figure 5.7. Priority Encoder Structure

Thread_Data BRAM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Q?	N0	N1	N2	N3	N4	N5	N6	N7	L0	L1	L2	L3	L4	L5	L6	P0	P1	P2	P3	P4	P5	P6	P7	-	-	-	-	-	-	-	-

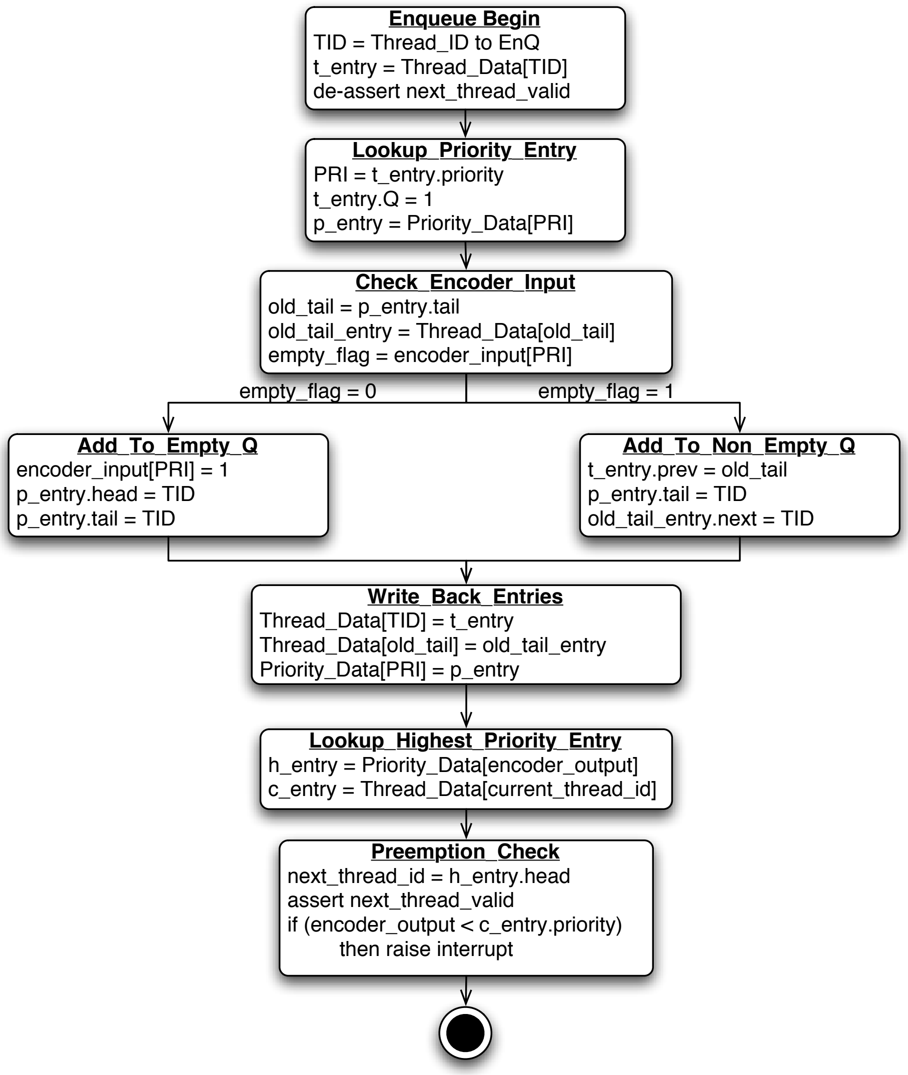
Field	**Width**	**Purpose**
Q?	(1-bit)	1 = Queued, 0 = Not Queued.
N0:N7	(8-bit)	Ready-to-run queue next pointer
L0:L6	(7-bit)	Scheduling priority-level
P0:P7	(8-bit)	Ready-to-run queue previous pointer

Priority_Data BRAM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
H0	H1	H2	H3	H4	H5	H6	H7	T0	T1	T2	T3	T4	T5	T6	T7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Field	**Width**	**Purpose**
H0:H7	(8-bit)	Priority-queue head pointer
T0:T7	(8-bit)	Priority-queue tail pointer

Figure 5.8. Scheduler Attribute Table Entry Format, Build 2



Enqueue Begin
TID = Thread_ID to EnQ
t_entry = Thread_Data[TID]
de-assert next_thread_valid
Lookup_Priority_Entry
PRI = t_entry.priority
t_entry.Q = 1
p_entry = Priority_Data[PRI]
Check_Encoder_Input
old_tail = p_entry.tail
old_tail_entry = Thread_Data[old_tail]
empty_flag = encoder_input[PRI]
empty_flag = 0
empty_flag = 1
Add_To_Empty_Q
encoder_input[PRI] = 1
p_entry.head = TID
p_entry.tail = TID
Add_To_Non_Empty_Q
t_entry.prev = old_tail
p_entry.tail = TID
old_tail_entry.next = TID
Write_Back_Entries
Thread_Data[TID] = t_entry
Thread_Data[old_tail] = old_tail_entry
Priority_Data[PRI] = p_entry
Lookup_Highest_Priority_Entry
h_entry = Priority_Data[encoder_output]
c_entry = Thread_Data[current_thread_id]
Preemption_Check
next_thread_id = h_entry.head
assert next_thread_valid
if (encoder_output < c_entry.priority)
then raise interrupt

Figure 5.9. State Diagram for Enqueue Operation, Build 2

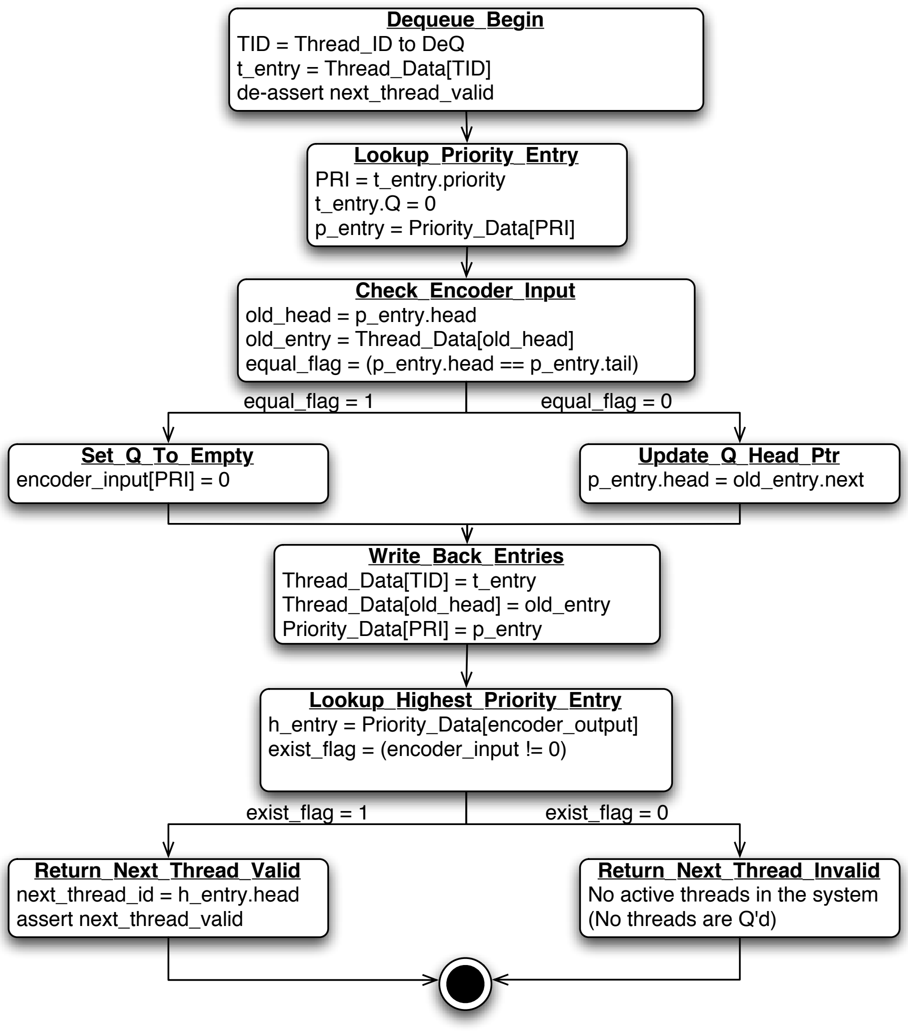

Dequeue_Begin
TID = Thread_ID to DeQ
t_entry = Thread_Data[TID]
de-assert next_thread_valid
Lookup_Priority_Entry
PRI = t_entry.priority
t_entry.Q = 0
p_entry = Priority_Data[PRI]
Check_Encoder_Input
old_head = p_entry.head
old_entry = Thread_Data[old_head]
equal_flag = (p_entry.head == p_entry.tail)
equal_flag = 1
equal_flag = 0
Set_Q_To_Empty
encoder_input[PRI] = 0
Update_Q_Head_Ptr
p_entry.head = old_entry.next
Write_Back_Entries
Thread_Data[TID] = t_entry
Thread_Data[old_head] = old_entry
Priority_Data[PRI] = p_entry
Lookup_Highest_Priority_Entry
h_entry = Priority_Data[encoder_output]
exist_flag = (encoder_input != 0)
exist_flag = 1
exist_flag = 0
Return_Next_Thread_Valid
next_thread_id = h_entry.head
assert next_thread_valid
Return_Next_Thread_Invalid
No active threads in the system
(No threads are Q'd)

Figure 5.10. State Diagram for Dequeue Operation, Build 2

5.3 Third Scheduler Redesign

One of the main goals of the HybridThreads system requires the capability to manage and schedule both SW and HW threads, therefore the thread manager and scheduler module must be aware of all threads, whether they are CPU-bound or not. The third scheduler module redesign was needed to incorporate support for the scheduling and management of both software and hardware-based threads.

Initially, thread management and scheduling activities were only concerned with software-resident threads (SW threads), while hardware-resident threads (HW threads) were just treated as memory-mapped coprocessors that the OS was relatively unaware of. The main goal of this redesign was to allow the OS to control HW threads just as it does SW threads so that they could be managed, scheduled, and synchronized using the standard APIs provided by the HybridThreads operating system. The changes brought about by this redesign will allow for HW and SW threads to be treated as "equals" by modifying the TM and scheduler module so that these components will be aware of all interactions between threads whether they are resident in SW or HW.

By examining the policies of thread management and scheduling it was determined that thread management operations are identical for both SW and HW threads, however, scheduling operations do have different meanings for SW and HW threads. Thread management policies deal with allocation of thread identifiers as well as operations that deal with manipulation of thread status, therefore thread management does not have any concern for *where* a thread is running. Scheduling policies are concerned with calculating scheduling decisions and dispatching threads to run, so scheduling operations must be cognizant of whether a thread is to be run in hardware or software. This allows for the Thread Manager to treat all threads in the same way, and it will be the job of the scheduler module to keep track of which threads are resident in HW or SW. Furthermore, this makes it possible for *all* the thread management and scheduling APIs to remain uniform for both software and hardware threads.

The differences in scheduling operations for SW and HW threads comes about because a HW thread is an independent execution unit: a stand-alone computation that does not require queuing, or mapping of the computation onto a shared resource for execution. Thus, a HW thread can be in either one of two states: running or not-running (stopped or blocked). Because a HW thread has dedicated computational resources, it never needs to be queued. On the other hand, a SW thread is a computation, but it requires some sort of computational unit (i.e. a shared CPU) to execute it. This means that a SW thread can be ready to begin running, but has not yet run, so it can be in one of three states: running, ready-to-run (queued), and not ready-to-run (blocked). A HW thread physically *exists*, it takes up physical space within the reconfigurable fabric of an FPGA, it has a base address, and its execution perfectly represents the program that it was derived from. A SW thread is merely a set of instructions that take up space in memory somewhere, and this group of instructions are scheduled to be executed on a computational unit. When that SW thread is chosen to be run, the instructions are gathered (fetched) and are then executed by a computational unit, thus making the computational unit emulate the behavior of the program that the SW thread was derived from.

The traditional method for scheduling a thread, whether it is a HW or SW thread, is the add_thread command. An add_thread command for a SW thread will add the thread to the ready-to-run queue and change the thread's status to queued. The SW thread's location in the queue is determined by its priority level which is encoded in its scheduling parameter. An add_thread command for a HW thread will do a "pseudo-add" which basically tells the HW thread to start its execution. A HW thread is not queued during an add_thread command because the HW thread itself is not a shared resource and can begin executing immediately. The HW thread's execution is started by writing a start message to the address of its command register which is encoded in its scheduling parameter that is stored within the scheduler module. Because the scheduling parameter is being used to store the address of the command register for HW threads, it is required to be a 32-bit value. HW and SW threads are thus distinguished by their scheduling parameters: a scheduling parameter whose value is between 0 and 127 denotes a SW thread with the parameter being its priority level; a scheduling parameter greater than 127 denotes a HW thread with the parameter being the address of its command register. This method of distinguishing between SW and HW threads allows for all thread management and scheduling APIs to remain the same regardless of the type of thread because the only difference in system operation for the two types of threads is the result of scheduling a thread by invocation of the add_thread command. Additionally, migration of computations between SW and HW can now be done trivially for computations in which both types of implementations exist by simply changing the scheduling parameter during run-time. The status of whether a thread is queued or not used to be known by both the TM and the scheduler module in the first and second builds because all add_thread commands resulted in a thread being added to the ready-to-run queue. The third scheduler build resulted in total management and scheduling control of both HW and SW threads, so add_thread commands do not result in any ready-to-run queue changes for HW threads. This meant that the status of being queued must only be held by the scheduler module, and the TM would have to request this information if it was needed. This change in the behavior of the add_thread operation led to the addition of the is_queued operation as well as the migration of idle thread functionality to the scheduler module. The is_queued operation is used by the TM to check whether a thread is queued or not, and the idle thread functionality is based upon the scheduling of an user-defined idle thread whenever there are no threads in the ready-to-run queue.

Both the Thread Manager (TM) and the scheduler module were implemented in the programmable logic of an FPGA. The system level architecture is shown in figure 5.11. Both components communicate directly and are attached to the HybridThreads system bus (IBM CoreConnect On-Chip Peripheral Bus, OPB) that allows the CPU to communicate with the modules. The purpose of the TM is to control all thread management operations, while the scheduler module controls all thread scheduling operations. The dedicated hardware interface between the scheduler module and the TM consists of a total of eight control and data signals as well as access to a read-only interface (B-port) of the TM's Block RAM (BRAM). This interface has the same capabilities as the previous TMcom interface set forth in section 5.1, however in this redesign it was modified to be more generic and extensible. Four of these interface signals are writable by the TM and readable by the scheduler module (`TM2SCH`), and are used for signaling the scheduler module to perform certain operations on behalf of the TM. The remaining four signals are writable by the scheduler module and readable by the TM (`SCH2TM`), and are used for return values as well as synchronization with the TM. The B-Port interface to the TM's BRAM allows the scheduler module to query thread management information in order to perform error-checking on certain scheduling operations just as in the previous builds.

The `TM2SCH_current_cpu_tid` data signal contains the identifier of the thread currently running on the CPU. The `TM2SCH_opcode` signal contains the encoded form of the operation that the scheduler module is to perform upon the TM's request. The `TM2SCH_data` signal contains any necessary parameters needed for an operation requested by the TM. The `TM2SCH_request` control signal is used to signify that the TM has a pending operation request for the scheduler module.

The `SCH2TM_busy` control signal is used to signify that the scheduler module is currently busy performing an operation and cannot accept any more incoming operations at this time. The `SCH2TM_data` signal is used to carry return value information back to the TM. The `SCH2TM_next_cpu_tid` data signal contains the identifier of the thread that will be scheduled to run next on the CPU. The `SCH2TM_next_tid_valid` control signal is used to signify whether the data found on `SCH2TM_next_cpu_tid` is valid or not.

The eight control and data signals implement a handshake protocol used to reliably coordinate communication between the scheduler module and the TM. The scheduler module has two main categories of operations: bus commands (BUScom), and TM commands (TMcom). The TM commands are only issued from the TM and are requested via the dedicated hardware interface. The bus commands can be issued from any device that is a bus-master, and are requested via the bus command register interface. The command set of the scheduler module can be seen in table 5.2.

The third scheduler redesign uses the O(1) ready-to-run queue structure described in section 5.2 with augmentations so that both SW and HW threads can be controlled by the scheduler module. This implementation uses a partitioned ready-to-run queue with a parallel priority encoder in order to provide constant time scheduling operations for both HW and SW threads without adding significant complexity to the scheduler module. This implementation uses three Block RAMs (BRAMs) to implement the ready-to-run queue: the `Priority_Data` BRAM, the `Thread_Data` BRAM, and the `Param_Data` BRAM. The `Param_Data`

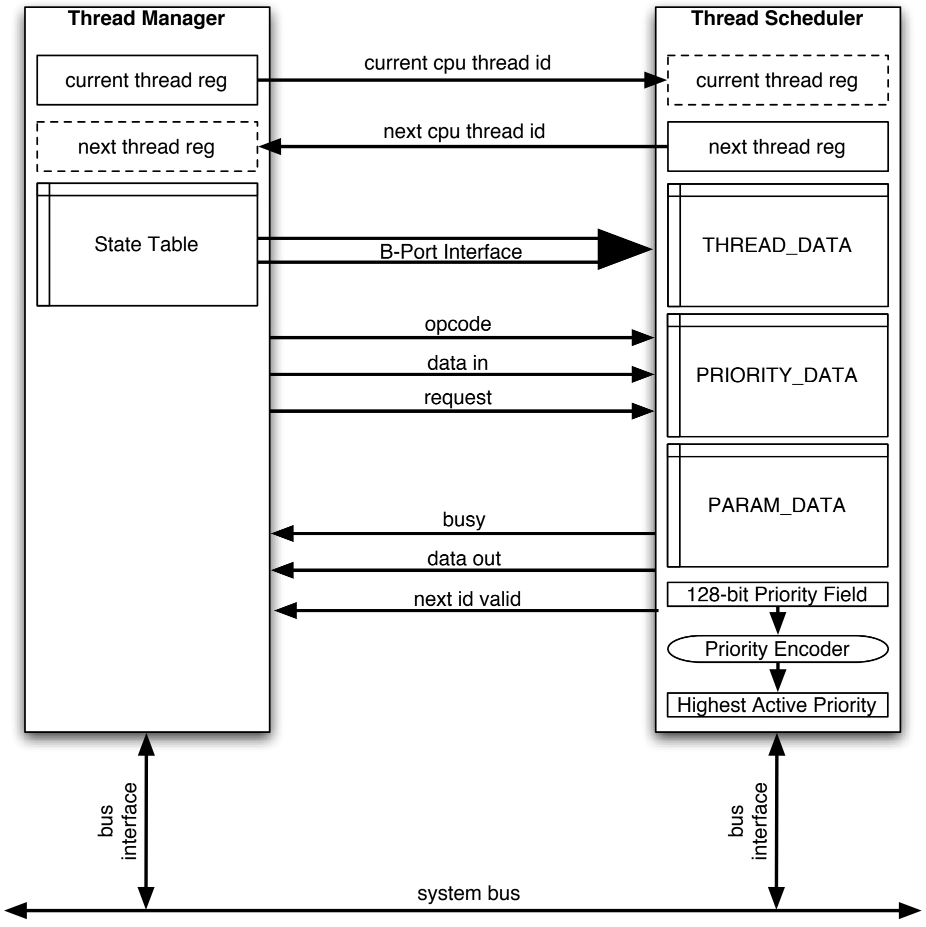
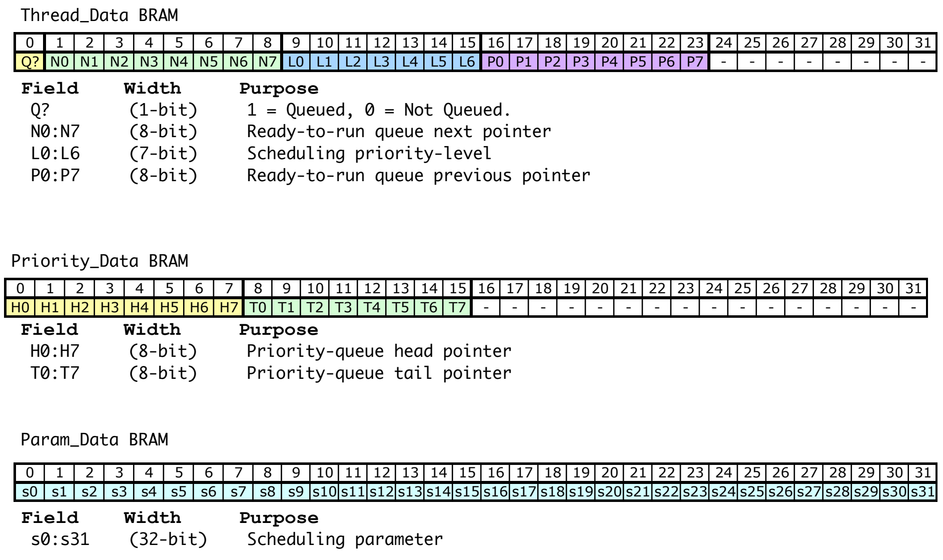


Figure 5.11. Block Diagram of Scheduler Module, Build 3

BRAM is used to store the 32-bit scheduling parameter for each thread which is an overloaded field used to distinguish between software or hardware resident threads. The `Thread_Data` and `Priority_Data` BRAMs, as well as the parallel priority encoder remain unchanged and are used in the exact same fashion as described in section 5.2. The `Param_Data` BRAM is indexed by thread identifier and can the addition of this BRAM can be seen as an extension to the width of the `Thread_Data` BRAM without affecting existing logic that interfaces to the `Thread_Data` BRAM.

Table 5.2. Command Set of Scheduler Module, Build 3

Type	Name	Actions
TMcom	Enqueue	Schedules a thread
TMcom	Dequeue	Removes a thread from the ready-to-run queue
TMcom	Is_Queued	Checks to see if a thread is queued
TMcom	Is_Empty	Checks to see if the ready-to-run queue is empty
BUScom	Toggle_Preemption	Toggle preemption interrupt on/off
BUScom	Get_Entry	Returns a thread's table attribute entry
BUScom	Set_Idle_Thread	Sets the identifier of the idle thread
BUScom	Get_Sched_Param	Returns the scheduling parameter of a thread
BUScom	Check_Sched_Param	Error-checks a thread's scheduling parameter
BUScom	Set_Sched_Param	Sets the scheduling parameter of a thread



Field	Width	Purpose
Q?	(1-bit)	1 = Queued, 0 = Not Queued.
N0:N7	(8-bit)	Ready-to-run queue next pointer
L0:L6	(7-bit)	Scheduling priority-level
P0:P7	(8-bit)	Ready-to-run queue previous pointer

Field	Width	Purpose
H0:H7	(8-bit)	Priority-queue head pointer
T0:T7	(8-bit)	Priority-queue tail pointer

Field	Width	Purpose
s0:s31	(32-bit)	Scheduling parameter

Figure 5.12. Scheduler Attribute Table Entry Format, Build 3

With this data organization, enqueue operations work by first determining whether a thread is resident in HW or SW by examining its scheduling parameter. If it is a hardware-resident thread (HW thread) then a RUN command is sent to the hardware thread and no ready-to-run queue manipulation is needed. If it is a software-resident thread (SW thread) then an enqueue operation results in adding a thread to the tail of the respective priority levels queue, and then adjusting the scheduling decision if needed based on the priority levels of the currently running thread and the thread that is scheduled next to run. The behavior of the enqueue operation can be seen in figure 5.13. Dequeue operations work by removing a thread from the head of the highest priority levels queue, and then immediately calculating the next scheduling decision. The behavior of the dequeue operation can be seen in figure 5.14. A new scheduling decision is calculated by looking up the head pointer of the highest priority queue from the `Priority_Data` BRAM using the output of the priority encoder and placing the head pointer's thread identifier into the `SCH2TM_next_cpu_tid` register. From figure 5.13 and figure 5.14, one can see that the partitioned ready-to-run queue along with the priority encoder allow for extremely simple enqueue and dequeue semantics that are able to operate in constant time regardless of queue length in the presence of both software and hardware resident threads. One important fact to note is that once the scheduler module was modified to support both HW and SW threads the entire HybridThreads system is capable of supporting HW and SW threads. This is due to the fact that thread management, status, and synchronization policies are not concerned with where a thread runs; only scheduling policies need to be cognizant of where a thread is running. All pre-existing APIs that result in the calling of scheduling operations now have support for both SW and HW threads, thus making "hybridizing" the entire system.

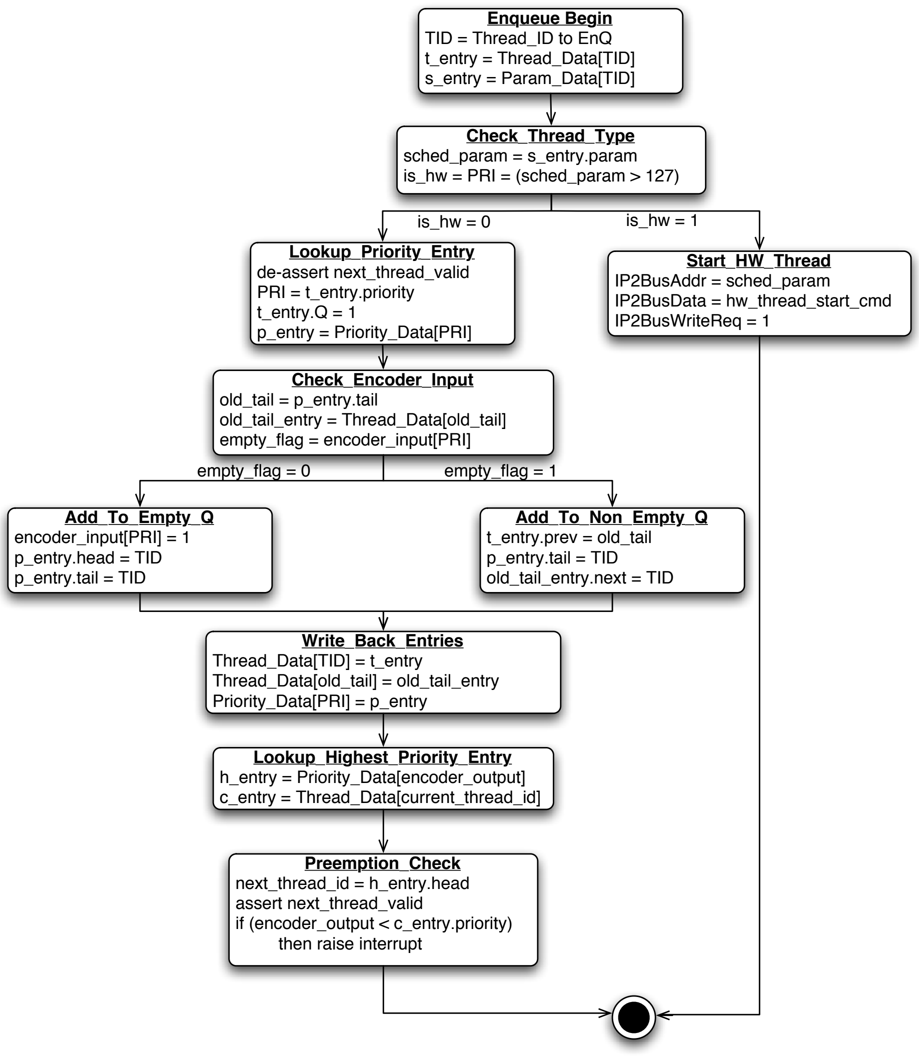

Enqueue Begin
TID = Thread_ID to EnQ
t_entry = Thread_Data[TID]
s_entry = Param_Data[TID]
Check Thread Type
sched_param = s_entry.param
is_hw = PRI = (sched_param > 127)
is_hw = 0
is_hw = 1
Lookup Priority Entry
de-assert next_thread_valid
PRI = t_entry.priority
t_entry.Q = 1
p_entry = Priority_Data[PRI]
Start HW Thread
IP2BusAddr = sched_param
IP2BusData = hw_thread_start_cmd
IP2BusWriteReq = 1
Check Encoder Input
old_tail = p_entry.tail
old_tail_entry = Thread_Data[old_tail]
empty_flag = encoder_input[PRI]
empty_flag = 0
empty_flag = 1
Add To Empty Q
encoder_input[PRI] = 1
p_entry.head = TID
p_entry.tail = TID
Add To Non Empty Q
t_entry.prev = old_tail
p_entry.tail = TID
old_tail_entry.next = TID
Write Back Entries
Thread_Data[TID] = t_entry
Thread_Data[old_tail] = old_tail_entry
Priority_Data[PRI] = p_entry
Lookup Highest Priority Entry
h_entry = Priority_Data[encoder_output]
c_entry = Thread_Data[current_thread_id]
Preemption Check
next_thread_id = h_entry.head
assert next_thread_valid
if (encoder_output < c_entry.priority)
then raise interrupt

Figure 5.13. State Diagram for Enqueue Operation, Build 3

The new hardware components used to schedule both SW and HW threads were fully integrated into the existing scheduler module. Results from simulation and synthesis tests to verify scheduler correct scheduler functionality of the new hybrid features along with O(1) ready-to-run queue structure can be found in chapter 6.

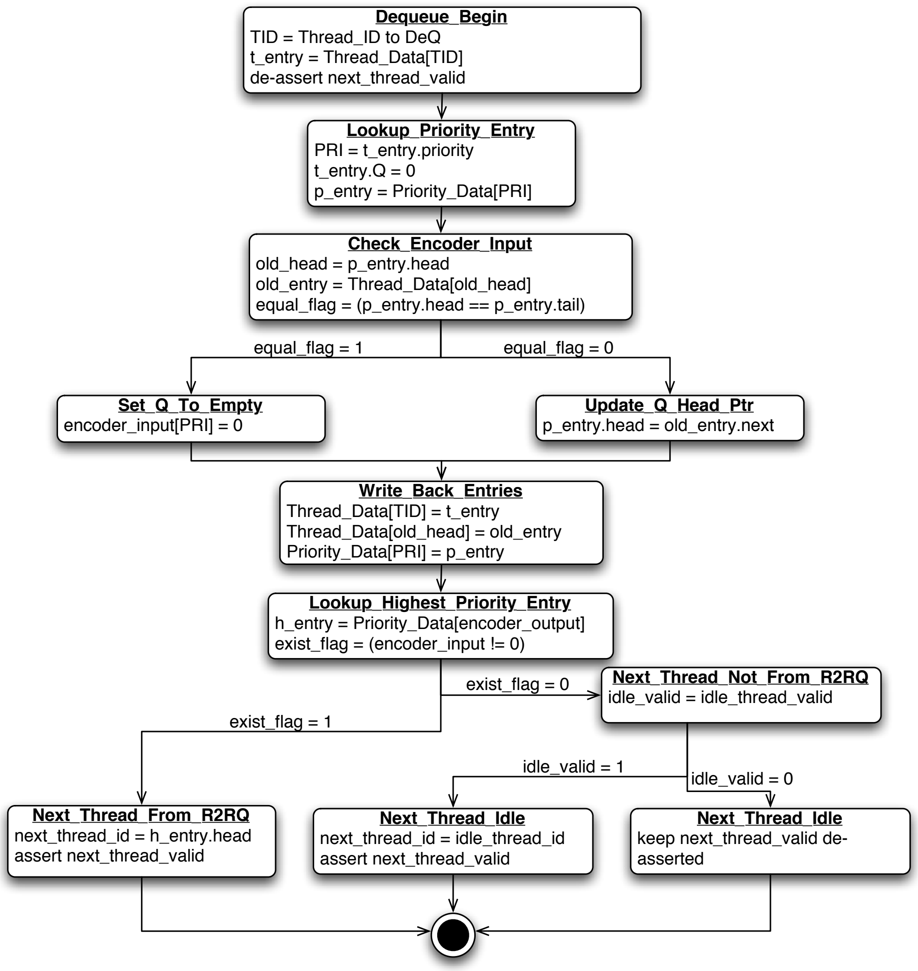

Dequeue_Begin
TID = Thread_ID to DeQ
t_entry = Thread_Data[TID]
de-assert next_thread_valid
Lookup_Priority_Entry
PRI = t_entry.priority
t_entry.Q = 0
p_entry = Priority_Data[PRI]
Check_Encoder_Input
old_head = p_entry.head
old_entry = Thread_Data[old_head]
equal_flag = (p_entry.head == p_entry.tail)
equal_flag = 1
equal_flag = 0
Set_Q_To_Empty
encoder_input[PRI] = 0
Update_Q_Head_Ptr
p_entry.head = old_entry.next
Write_Back_Entries
Thread_Data[TID] = t_entry
Thread_Data[old_head] = old_entry
Priority_Data[PRI] = p_entry
Lookup_Highest_Priority_Entry
h_entry = Priority_Data[encoder_output]
exist_flag = (encoder_input != 0)
exist_flag = 0
Next_Thread_Not_From_R2RQ
idle_valid = idle_thread_valid
exist_flag = 1
idle_valid = 1
idle_valid = 0
Next_Thread_From_R2RQ
next_thread_id = h_entry.head
assert next_thread_valid
Next_Thread_Idle
next_thread_id = idle_thread_id
assert next_thread_valid
Next_Thread_Idle
keep next_thread_valid de-asserted

Figure 5.14. State Diagram for Dequeue Operation, Build 3

Chapter 6

Implementation Results

The modules from each of the scheduler redesigns have undergone a series of tests in both simulated and synthesized forms. These tests help show the performance differences between the different ready-to-run queue structures in terms of scheduling overhead and jitter as well as the effects of the hybridization of the scheduler on the entire HybridThreads operating system.

6.1 Results of the First Redesign

The scheduler module was implemented on a Xilinx [27] ML310 development board which contains a Xilinx Virtex-II Pro 30 FPGA. Synthesis of the first redesign of the scheduler module yields the following FPGA resource statistics: 484 out of 13,696 slices, 573 out of 27,392 slice flip-flops, 873 out of 27,392 4-input LUTs, and 1 out of 136 BRAMs. The module has a maximum operating frequency of 166.7 MHz, which easily meets our goal of having a 100 MHz global system clock within the FPGA.

This version of the scheduler module has a O(n) ready-to-run queue structure. This structure requires a sequential queue traversal in order to calculate the next scheduling decision. Scheduling decisions are calculated during a dequeue operation, which is invoked when a thread begins running on the CPU and must be unlinked from the ready-to-run queue. The process of unlinking the newly running thread and calculating the next thread to run is handled by the dequeue operation of the scheduler module. Table 6.1 shows that it takes approximately 40 ns (4 clock cycles) per thread in the ready–to-run queue to perform the dequeue operation, with a small amount of setup cost due to the setup of the queue traversal. These results show that in the worst case scenario (250 threads in the ready-to-run queue) a dequeue operation will take about 10 μs to complete, running concurrently while the thread on the CPU is context switched to and begins to run. This variability in execution-time of the dequeue operation provided the impetus for redesigning the ready-to-run queue structure. The first redesign of the scheduler module provides low overhead scheduling operations that run in parallel with application execution, however there is still jitter evident in the process of making a scheduling decision. The goal of the second redesign of the scheduler module is to reduce or eliminate the jitter within the scheduler module.

Table 6.1. ModelSim Timing Results of Dequeue Operations, Build 1

No. of Threads in R2RQ	Time (ns)	Est. Time/Thread (ns)
250	10060	40.24
128	5140	40.16
64	2610	40.78
32	1330	41.56
16	690	43.13
2	130	65

To test the scheduler in synthesized form it must be fully incorporated into the HybridThreads operating system. This involves attaching the BUScom interface of the scheduler module to the HybridThreads system bus and the TMcom interface to the thread manager. The first performance measurement taken of the synthesized and integrated scheduler module is end-to-end scheduling delay. End-to-end scheduling delay in our system is defined as the time delay between when a timer interrupt fires to when the context switch to a new thread is about to complete (old context saved, new context is about to be switched to). The end-to-end scheduling delay was measured using a hardware counter that begins clocking immediately after a timer interrupt goes off and would stop timing as soon as the context switch was complete. The hardware counter measures this delay in an non-invasive way by monitoring the interrupt lines attached to the CPU. This allows for clock-cycle accurate measurements to be made without bias. Figure 6.1 shows a histogram of the end-to-end scheduling delay measurements for our system with 250 active threads, where each thread is in an infinite loop. The shaded region of figure 6.1 and figure 6.2 highlight the data range of the measurement from minimum to maximum delay. The end-to-end scheduling delay over a course of 150,000 events was approximately 2.0 μs with 250 active threads, with approximately 2.4 μs of jitter. Where the jitter is defined as being the difference between the maximum and mean delays. The results of both the average end-to-end scheduling delay and jitter are quite low when compared to the 1.3 ms scheduling delay of Linux [26] and the 40 μs scheduling delay of Malardalen University's RTU [24], and further tests have shown that the jitter is caused by cache misses during context switching as well as jitter and delay in the time it takes for the CPU to respond to the interrupt and jump into its ISR.

The second performance measurement taken of the system is raw interrupt delay. Raw interrupt delay is the time delay in our system from the time an interrupt signal fires to when the CPU actually enters its ISR. The raw interrupt delay was also measured using a hardware module that would begin timing immediately after a timer interrupt goes off and would stop timing when the CPU sent a signal to the hardware timer from its ISR. Figure 6.2 shows a histogram of the interrupt delay in our system with various numbers of active threads, where each thread is infinitely looping. This measurement shows that the average raw interrupt delay over a course of 2,500,000 events with 250 active threads is approximately 0.79 μs with approximately 0.73 μs of jitter. One can see that the raw interrupt delay makes up a significant portion of the end-to-end scheduling delay and jitter. The interrupt delay is relatively constant, and its jitter is primarily caused by the variable execution length of atomic instructions that prevent the CPU from acknowledging the interrupt immediately.

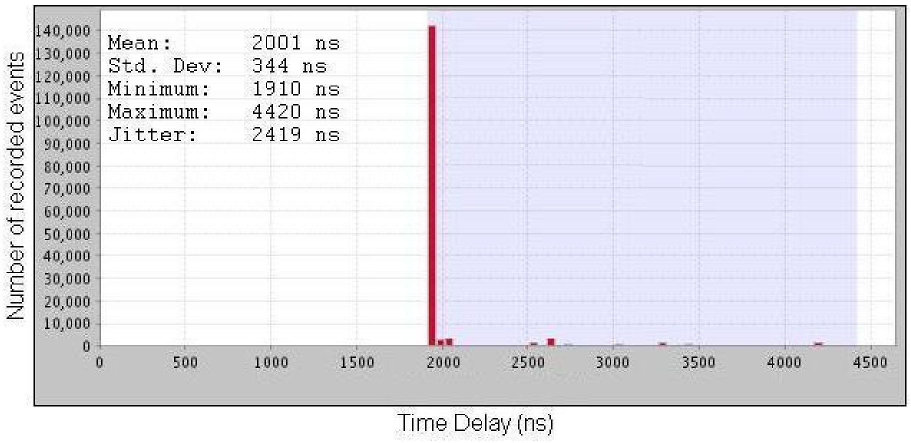


Figure 6.1. Histogram of Integrated End-To-End Scheduling Delay, Build 1 (250 Active SW Threads)

During a context switch, the CPU does not interact with the scheduler module, which gives the scheduler module the perfect opportunity to calculate the next scheduling decision. This scheduling decision is calculated in parallel with the

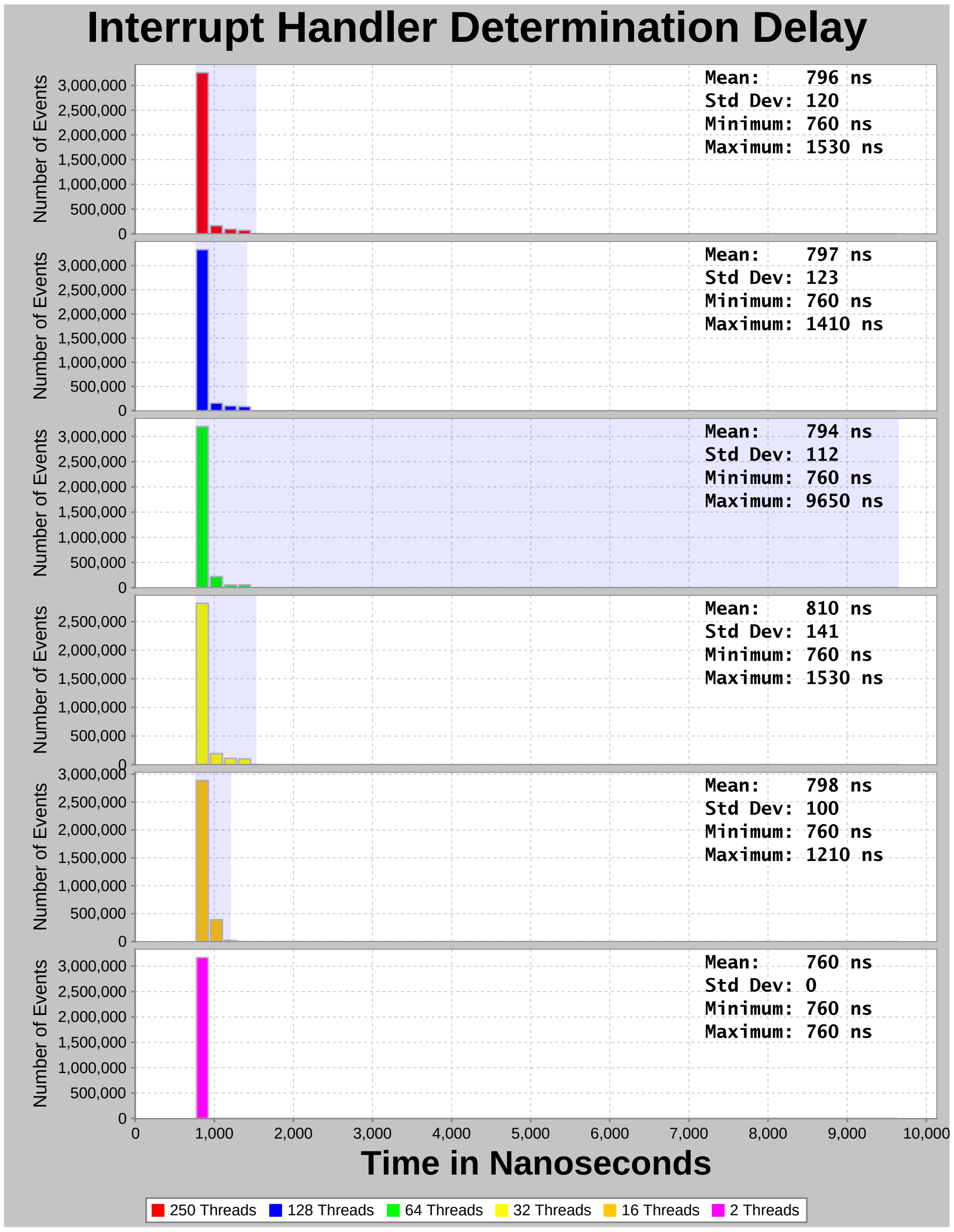


Figure 6.2. Histogram of Raw Interrupt Delay, Build 1

CPU as shown in the timeline shown in figure 1, thus eliminating much of the processing delays normally incurred by calculating a new scheduling decision. Also, the scheduling decision being calculated is for the *next* scheduling event so that when a timer interrupt goes off, the next thread to run has already been calculated. This pre-calculation of the scheduling decision allows the system to react very quickly to scheduling events because when a scheduling event occurs, the OS simply reads the next thread to run out of a register and performs a context switch, and then during this context switch, the register is refreshed with the thread that should run at the next scheduling event by the hardware scheduler module. The overhead of making a scheduling decision is hidden because it occurs during a context switch. However, if a scheduling event occurs very soon after a thread is initially context switched to, the system may have to wait for the scheduler to finish calculating the next scheduling decision resulting from the context switch. This is more likely to happen with more threads in the ready-to-run queue, due to increased execution-times of scheduling operations due to the O(n) nature of the queue itself. Although the amount of jitter is in the microsecond range, it can still be further reduced by restructuring the ready-to-run queue so that its functions are able to operate in constant amounts of time.

6.2 Results of the Second Redesign

Synthesis of the second redesign of the scheduler module targeting a Xilinx [27] Virtex-II Pro 30 yields the following FPGA resource statistics: 1,034 out of 13,696 slices, 522 out of 27,392 slice flip-flops, 1,900 out of 27,392 4-input LUTs, and 2 out of 136 BRAMs. The module has a maximum operating frequency of 143.8 MHz, which easily meets our goal of a 100 MHz clock frequency.

This version of the scheduler module has a O(1) ready-to-run queue structure. This structures uses a high-speed priority encoder to select the highest priority-level active in the system. The output of the priority encoder is used to access the first entry of the highest priority-levels queue. This type of data structure requires the use of more BRAM (2 versus 1), however no data structure traversals are required to make scheduling decisions. This makes for a ready-to-run queue structure that has fixed execution-times for making scheduling decisions as shown in table 6.2. This table shows that the time required to execute a dequeue operation is constant (240 ns) regardless of the length of the ready-to-run queue.

Table 6.2. ModelSim Timing Results of Dequeue Operations, Build 2

No. of Threads in R2RQ	Time (ns)
250	240
128	240
64	240
32	240
16	240
2	240

After being synthesized and fully integrated into the HybridThreads system, the second redesign of the scheduler has an average end-to-end scheduling delay of 1.9 μs with 1.4 μs of jitter with 250 active threads as shown in figure 6.3. The raw interrupt delay of this system with 250 active threads is shown in figure 6.4. This figure agrees with the results in figure 6.2, thus showing that raw interrupt delay remains constant in the system, regardless of any changes in the HybridThreads architecture itself. The O(1) ready-to-run queue structure is able to reduce the amount of end-to-end scheduling jitter in the system by about 1 μs. However, the scheduler module must still be modified further in order to uniformly handle both SW and HW threads within the HybridThreads system.

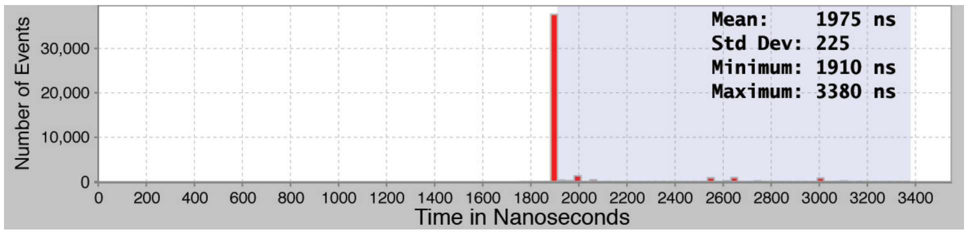


Figure 6.3. Histogram of Integrated End-To-End Scheduling Delay, Builds 2 & 3 (250 Active SW Threads)

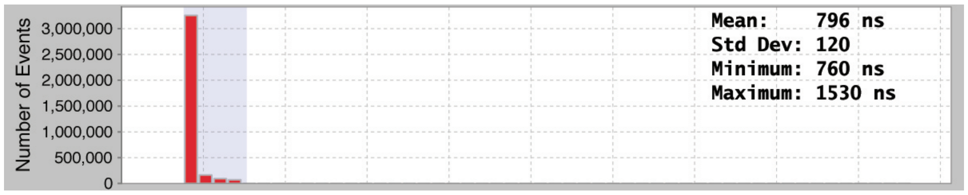


Figure 6.4. Histogram of Raw Interrupt Delay, Builds 2 & 3 (250 Active SW Threads)

6.3 Results of the Third Redesign

Synthesis of the third redesign of the scheduler module targeting a Xilinx [27] Virtex-II Pro 30 yields the following FPGA resource statistics: 1,455 out of 13,696 slices, 973 out of 27,392 slice flip-flops, 2,425 out of 27,392 4-input LUTs, and 3 out of 136 BRAMs. The module has a maximum operating frequency of 119.6 MHz, which easily meets our goal of a 100 MHz clock frequency.

This version of the scheduler module retains the O(1) ready-to-run queue structure, however it incorporates functionality to support the scheduling of both software and hardware threads. This scheduler module requires storage of a larger scheduling parameter to distinguish between hardware and software threads, thus requiring an additional BRAM (3 for this redesign versus 2 for the second redesign). Additionally, the scheduler module must be able to master the HybridThreads system bus in order to send commands to hardware threads. This requires the addition of a bus master interface attachment (Master IPIF) to the scheduler which causes a slight increase in the amount of logic resources needed to implement the module. These additions to the scheduler module result in fixed execution-time scheduling operations that are uniformly accessible to both software and hardware threads.

The primary scheduler operations have been simulated with ModelSim to characterize their execution times. Worst-case timing results for the scheduler's operations are shown in table 6.3. These results were verified by comparison against the expected results based on the number of states and transitions in the FSMs that implement the operations.

Table 6.3. ModelSim Timing Results of Scheduling Operations, Build 3

Operation	Time (clock cycles)
Enqueue(SWthread)	28
Enqueue(HWthread)	20 + (1 Bus Transaction)
Dequeue	24
Get_Entry	10
Is_Queued	10
Is_Empty	10
Set_Idle_Thread	10
Get_Sched_Param	10
Check_Sched_Param	10
Set_Sched_Param(NotQueued)	10
Set_Sched_Param(Queued)	50

From table 6.3, one can see that all of the scheduling operations execute in 500 ns (50 clock cycles) or less. This is extremely fast, especially when considering how much processing is being done, and that the FPGA and the scheduler module are only being clocked at 100 MHz. These measurements do not vary with the number of active (queued) threads, which makes for jitter-free scheduling operations at the base hardware level, and thus more predictable and precise scheduling of threads within the OS.

In synthesized form, the third redesign of the scheduler has the same performance as that of the second redesign of the scheduler because all modifications to the scheduler module involve adding functionality to support both hardware and software thread scheduling operations. Therefore all scheduling operations that involve only software threads will have the same system performance when using the scheduler module from build 2 or 3. The end-to-end scheduling delay and raw interrupt delay for the third scheduler redesign are the same as for the second scheduler redesign and are thus shown in figures 6.3 and 6.4 (found in section 6.2) respectively. These figures show that system with 250 active threads in it has a mean end-to-end scheduling delay of 1.9 μs with 1.4 μs of jitter, and with raw interrupt delay of of 0.79 μs with approximately 0.73 μs of jitter. These results are the same for both the second and third scheduler builds because the tests do not involve any hardware threads, and the mechanisms for handling software threads remained the same in both of the redesigns. Overall, this redesign resulted in a system capable of uniform OS services for both SW and HW threads with significant reductions in scheduling overhead and jitter by combining the hybrid scheduling mechanism together with the O(1) ready-to-run queue structure from the second scheduler redesign.

Chapter 7

Conclusion

In this paper we have presented the design of our scheduler module implemented in programmable logic. This design takes advantage of current FPGA technology to provide important operating system services that operate concurrently with the processor core(s). The scheduler module supports FIFO, round-robin, and priority-based scheduling algorithms. The system currently supports up to 256 active threads, with up to 128 different priority levels. The scheduler module provides constant time scheduling operations that execute in under 50 clock cycles (500 ns) or less at the base hardware level. From the system level, the hardware scheduler module provides very fast scheduling operations with an end-to-end scheduling delay of 1.9 μs with 1.4 μs of jitter with 250 active threads running on a Xilinx [27] Virtex-II Pro FPGA. The integrated system level tests have shown that the migration of scheduling services into the fabric of the FPGA have drastically reduced the amount of system overhead and jitter related to the scheduling of threads, thus allowing for more precise and predictable scheduling services. Migration of scheduling and management services has also allowed for uniform operating system services to be provided for both SW and HW threads.

This effectively raises the abstraction level of the hardware into the domain of threads and OS services. SW and HW computations have been encapsulated as threads which allows the computations to be treated as ”equals” as they are now able to communicate and synchronize with all threads in the system through the use of uniform, high-level APIs commonly seen in the world of multithreaded software programming. Overall, this thesis work has enabled uniform operating system support for both software and hardware based computations. Additionally, system overhead and jitter have been greatly reduced by designing and implementing a scheduling module in hardware whose operations all have fixed execution-times.

Immediate future work is now focused on creating a more flexible organization that will allow the user to reconfigure the scheduler module to support an arbitrary number of threads and priority levels, as well as user-defined scheduling algorithms. We are also working on optimizing the scheduler module in terms of FPGA resource utilization. The goal of this work is to try to shrink the size of the hardware implementation of the scheduler module as much as possible in order to free up additional FPGA resources for use by additional hardware modules.

The capability of migrating traditional operating system services into hardware allows system designers to improve concurrency, and reduce system overhead and jitter, which has the possibility of making real-time and embedded systems more precise and accurate. Complete systems can be built by integrating these HW/SW co-designed modules into an FPGA so as to provide fast and precise operating systems services to the embedded domain while still adhering to standard programming models and APIs. Additionally, the final redesign of the scheduler enables full OS support across the hardware/software boundary. Allowing all computations in the system, whether implemented in software or within the FPGA, to communicate and synchronize through the use of high-level APIs compatible with the POSIX thread standard. The APIs used in the HybridThreads system provide accessibility to the services and computations within the FPGA at level of abstraction familiar to that of software programmers without the need for such programmers to knowledge of hardware design principles.

New research projects have spawned from the development methodologies of the HybridThreads project [29], which furthers the future impacts of the project even more. More detailed descriptions of the hybrid threads project can be found in [13].

Acknowledgment

The work in this article is partially sponsored by National Science Foundation EHS contract CCR-0311599. The opinions expressed are those of the authors and not necessarily those of the foundation.

References

[1] J. Adomat, J. Furuns, L. Lindh, and J. Starner. Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, 1996.

[2] J. Agron, D. Andrews, M. Finley, E. Komp, and W. Peck. FPGA Implementation of a Priority Scheduler Module. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium, Works in Progress Session (RTSS WIP)*, Lisbon, Portugal, December 2004.

[3] D. Andrews, D. Niehaus, and P. Ashenden. Programming Models for Hybrid FPGA/CPU Computational Components. *IEEE Computer*, 37(1):118–120, Jan. 2004.

[4] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbee, J. Ortiz, E. Komp, and P. Ashenden. Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link. *IEEE Micro*, 24(4):42–53, 2004.

[5] 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*, 24(4):42–53, July/August 2004.

[6] 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 (ETFA), Catania, Sicily, September 2005.

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

[8] J. Furunas. Benchmarking of a Real-Time System that Utilises a Booster. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, Las Vegas, Nevada, June 2000.

[9] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers. Optimized Generation of Data-Path from C Codes. In *Proceedings of the ACM/IEEE Design Automation and Test (DATE)*, March 2005.

[10] V. J. M. III and D. M. Blough. A Hardware-Software Real-Time Operating System Framework for SOCs. *IEEE Design & Test*, 19(6):44–51, Nov/Dec 2002.

[11] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[12] J. Kreuzinger, A. Schulz, M. Pfeffer, T. Ungerer, U. Brinkschulte, and C. Krakowski. Real-time Scheduling on Multithreaded Processors. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 155–159, Cheju Island, South Korea, Dec. 2000.

[13] KU HybridThreads. Project Wiki. http://wiki.ittc.ku.edu/hybridthread/Main_Page. Last accessed May 1, 2006.

[14] P. Kuacharoen, M. Shalan, and V. M. III. A Configurable Hardware Scheduler for Real-Time Systems. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96–101, June 2003.

[15] J. Lee, V. Mooney, K. Instrom, A. Daleby, T. Klevin, and L. Lindh. A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 683–688, Kitaskyushu International Conference Center, Japan, Jan. 2003.

[16] S. W. Moon, J. Rexford, and K. G. Shin. Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches. *IEEE Transactions on Computers*, 49(11):1215–1227, 2000.

[17] M. D. Natale and J. A. Stankovic. Scheduling Distributed Real-Time Tasks with Minimum Jitter. *IEEE Transactions on Computers*, 49(4):303–316, 2000.

[18] D. Niehaus and D. Andrews. Using the Multi-Threaded Computation Model as a Unifying Framework for Hardware-Software Co-Design and Implementation. In *Proceedings of the 9th Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, pages 318–326, Isle of Capri, Italy, Sept. 2003.

[19] D. A. Patterson and J. L. Hennessy. *Computer architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[20] W. Peck, J. Agron, D. Andrews, M. Finley, and E. Komp. Hardware/Software Co-Design of Operating Systems for Thread Management and Scheduling. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium, Works in Progress Session (RTSS WIP)*, Lisbon, Portugal, December 2004.

[21] 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.

[22] Realfast. Realfast. http://www.realfast.se/. Last accessed May 1, 2006.

[23] S. Saez, J. Vila, A. Crespo, and A. Garcia. A Hardware Scheduler for Complex Real-Time Systems.

[24] T. Samuelsson, M. Akerholm, P. Nygren, J. Starner, and L. Lindh. A Comparison of Multiprocessor RTOS Implemented in Hardware and Software. In *Proceedings of the 15th Euromicro Workshop on Real-Time Systems*, 2003.

[25] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 6th Edition.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[26] C. Williams. Linux Scheduler Latency. Red Hat Inc. Technical Paper. http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf.

[27] Xilinx. Programmable logic devices. http://www.xilinx.com/. Last accessed May 1, 2006.

[28] V. Yodaiken. The RTLinux Manifesto. In *Proceedings of The 5th Linux Expo, Raleigh, NC*, Mar. 1999.

[29] 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)*, pages 788–792, 2005.