Run-Time Services for Hybrid CPU/FPGA Systems on Chip

Jason Agron, Wesley Peck, Erik Anderson, David Andrews, Ed Komp, Ron Sass, Fabrice Baijot, Jim Ste
Information and Telecommunication Technology Center - University of Kansas
2335 Irving Hill Road, Lawrence, KS
{jagron,peckw,eanderso,dandrews,komp,rsass,bricefab,jstevens}@ittc.ku.edu

Abstract

Modern FPGA devices, which include (multiple) processor core(s) as diffused IP on the silicon die, provide an excellent platform for developing custom multiprocessor systems-on-programmable chip (MPSoPC) architectures. As researchers are investigating new methods for migrating portions of applications into custom hardware circuits, it is also critical to develop new run-time service frameworks to support these capabilities. Hthreads (HybridThreads) is a multithreaded RTOS kernel for hybrid FPGA/CPU systems designed to meet this new growing need. A key capability of hthreads is the migration of thread management, synchronization primitives, and run-time scheduling services for both hardware and software threads into hardware. This paper describes the hthreads scheduler, a key component for controlling both software-resident threads (SW threads) and threads implemented in programmable logic (HW threads). Run-time analysis shows that the hthreads scheduler module helps in reducing unwanted system overhead and jitter when compared to historical software schedulers, while fielding scheduling requests from both hardware and software threads in parallel with application execution. Run time analysis shows the scheduler achieves constant time scheduling 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.

1 Introduction

1.1 Uniformity of Services in a Hybrid System

FPGAs provide a "blank" slate of computational components. Standard interfaces and services have not yet been established [9, 27], thus forcing hybrid system designers to literally build systems from "scratch". Conversely, abstract interfaces and services have been established in the world of software, and these have been standardized in programming models implemented in part with middleware libraries and operating systems (OSes). Of critical importance is the operating system, which forms the basis for common services to the user in a convenient and efficient manner [21]. Pragmatically, operating systems work as the intermediary to manage the lower-level platform-specific details of the system hardware. An OS provides services that enforce how programs, or more abstractly, computations, execute and communicate. Real time operating systems typically provide the task abstractions, 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 silicon structure to support tasks in either hardware or software. To exploit this structure 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 interrupting the software tasks to field OS calls from hardware tasks. This is a very inefficient method of organizing OS services that should 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 could be resident within a central computational component (i.e. CPU). This implies that an additional CPU could be used as a system 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. Alternatively, a specialized set of hardware based OS services can be developed. This approach allows the ISA of the OS copro-

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06/$20.00 © 2006 IEEE
Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1014, Page 000001
IPR2025-00478 (Advanced Micro Devices, Inc., Pensando Systems, Inc. v. Concurrent Ventures, LLC, XtreamEdge, Inc.)

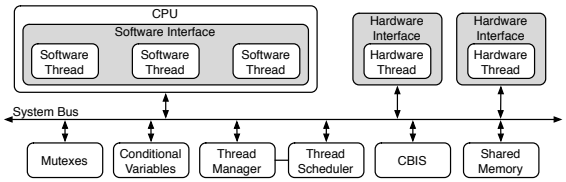

Figure 1. hthread System Block Diagram

cessor 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. The parallel nature of FPGAs also allows 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, improving overall system performance by reducing the overhead and jitter normally incurred by executing these OS services on a sequential CPU [16]. 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.

1.2 hthreads: A Distributed Run-Time Kernel

Figure 1 shows the hthreads run-time system components implemented in hardware. Hthreads migrates a Thread Manager, Scheduler, Mutex Manager, and a new CPU Bypass Interrupt Scheduler (CBIS) into the reconfigurable fabric on an FPGA. Migrating these services into hardware brings significant performance benefits to software threads through more efficient invocation and processing mechanisms [17, 2]. First, invocation mechanisms for accessing the system services are no longer based on inefficient traversal of hierarchical software protocol stacks, but instead are achieved through lightweight, atomic load and store operations. Second, speculative and variable length execution paths performed within key system services, such as the scheduler, are eliminated.

As an operational example, the mutex_unlock() operation illustrated in Figure 2 shows the processing steps the hthreads system performs to release a mutex, make a scheduling decision, and resume the execution of a thread. In a traditional operating system, steps A through E are

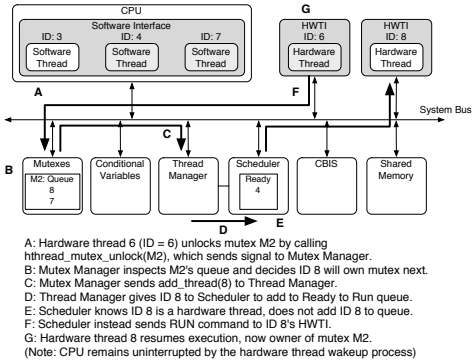

Figure 2. hthread Mutex Unlock Sequence

performed completely in software executing on the CPU. These steps would require a context switch from the application thread to the system services, and must be performed before the scheduler considers if a new scheduling decision is required based on the queuing of a blocked thread. In hthreads, steps A through G are performed in hardware, allowing application threads to continue executing on the CPU without interruption. For systems with both hardware and software threads, migrating this processing off the CPU is critical, as significant overhead and jitter can be introduced if the CPU must be preempted in order to process OS requests for hardware threads being unblocked. In contrast, [23, 24] reports a multithreaded capability that supports the creation and control of both hardware and software threads through Linux running on the CPU. This approach was taken to allow hardware threads to access data through Linux's existing virtual memory address space. Although convenient, this approach requires additional complexity within the hardware thread to maintain virtual address translation tables, and requires the use of external interrupts in order to invoke the memory manager, running on the CPU, for page swapping; thus demonstrating how unwanted overhead and jitter can be introduced when OS processing for both software and hardware threads must be executed on a shared computational resource such as a CPU.

2 Achieving Precise Scheduler Services

It has been well understood that system overhead and jitter can be dramatically reduced by careful redesign and migration of portions of the RTOS into concurrent co-designed hardware modules [22, 13, 18, 8]. Migrating these services off the CPU also helps in eliminating the hidden overhead of context switch times associated with entering and exiting the RTOS. Perhaps the biggest benefit of this approach is the ability to seriously reduce the overhead and jitter associated with processing variable entry points into the RTOS

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06/$20.00 © 2006 IEEE
Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.
IEEE COMPUTER SOCIETY

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 hthreads system [3, 4, 15, 5] 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 hthreads project for other RTOS services including semaphores, thread management, and hardware thread control mechanisms.

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, 18, 13]. Iterative control loops, used for searching through priority lists, can be decomposed into combinational circuits controlled by finite state machines (FSMs) that perform parallel sorting 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, [8] 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) [10]. 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. Due to the fact that the scheduling decision has been made (or is being made in cases of higher frequency timer interrupts) 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 and be serviced later 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 hthreads 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 pre-processing 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 seman-

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06/$20.00 © 2006 IEEE
Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.
IEEE COMPUTER SOCIETY

tics of allowing asynchronous invocations to interrupt the CPU, and incurring the overhead of a context switch to an interrupt service routine.

3 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, 12, 19]. 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 [14]. 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 3, 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 hthreads, 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 [12], 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 hthreads 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 implementation of the scheduler for the hthreads system, described in sections 4 and 5 only requires 1,455 logic slices and 973 flip-flops for a queue of size 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 hthreads 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 hthreads scheduler to incorporate new properties that distinguish between hardware and software threads, as well as different mechanisms that handle

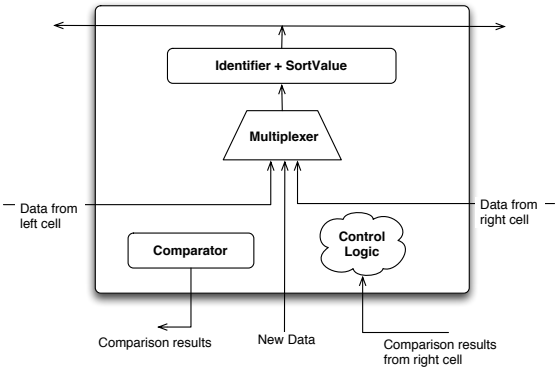

Figure 3. Structure of Typical Systolic Cell

the scheduling of each type of thread.

Additionally, the entire hthreads system is built around APIs that are fully compatible with the POSIX thread (pthread) standard [6]. Both Malardalen University's Real-Time Unit (RTU) and Georgia Tech's configurable hardware scheduler [12] use their own custom APIs for interacting with OS services. Using POSIX thread compatible APIs has enabled the hthreads system to be extremely accessible to those familiar with the POSIX thread standard. Additionally, simple library wrappers can be used to port applications written for use on the hthreads system to a POSIX thread application capable of running on a desktop computer equipped with POSIX thread libraries. This allows for rapid prototypes of hthreads applications to be debugged on a typical desktop machine, and then later ported for use on the hthreads architecture at no cost to the system programmer.

4 Scheduler Module Design

The Hybrid Threads system requires the capability to manage and schedule both SW and HW threads, therefore the thread manager and scheduler modules must be aware of all threads, whether they are CPU-bound or not. By examining the policy 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 deals 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 operations deal 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

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06/$20.00 © 2006 IEEE
Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.
IEEE COMPUTER SOCIETY

threads in the same way, and it will be the job of the scheduler 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 scheduling. Thus, a HW thread can be in either of two states: running or not-running (stopped or blocked). Because a HW thread has dedicated computational resources, it is 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 CPU) to execute it. This means that a SW thread can be ready to begin running, but has not ran yet, so it can be in either 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 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 room in memory somewhere, and this group of instructions are scheduled to be run 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 sends the HW thread a start command to begin its execution. A HW thread is not queued during an add_thread command because the HW thread itself is not a shared resource. The HW thread's execution is started when the scheduler module writes a run message to the address of its command register which is encoded in the thread's scheduling parameter. 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 it's priority level; a scheduling parameter greater than 127 denotes a HW thread with the parameter being the address of it's command register. This method of distinguishing between SW and HW threads allows for all thread management and scheduling APIs to remain uniform 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.

Both the thread manager (TM) and the scheduler are implemented in the programmable logic of an FPGA. The system-level architecture is shown in figure 4. Both components communicate directly and are attached to a peripheral bus (IBM CoreConnect Bus) that allows the CPU to communicate with the modules. The purpose of the TM is to control all thread management operations, while the scheduler controls all thread scheduling operations. The dedicated hardware interface between the scheduler 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). Four of these interface signals are writable by the TM and readable by the scheduler (`TM2SCH`), and are used for signaling the scheduler to perform certain operations on behalf of the TM. The remaining four signals are writable by the scheduler 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 to query thread management information in order to perform error-checking on certain scheduling operations.

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 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.

The `SCH2TM_busy` control signal is used to signify that the scheduler 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 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 1.

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06/$20.00 © 2006 IEEE
Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.
IEEE COMPUTER SOCIETY

Table 1. Scheduler Command Set

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	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

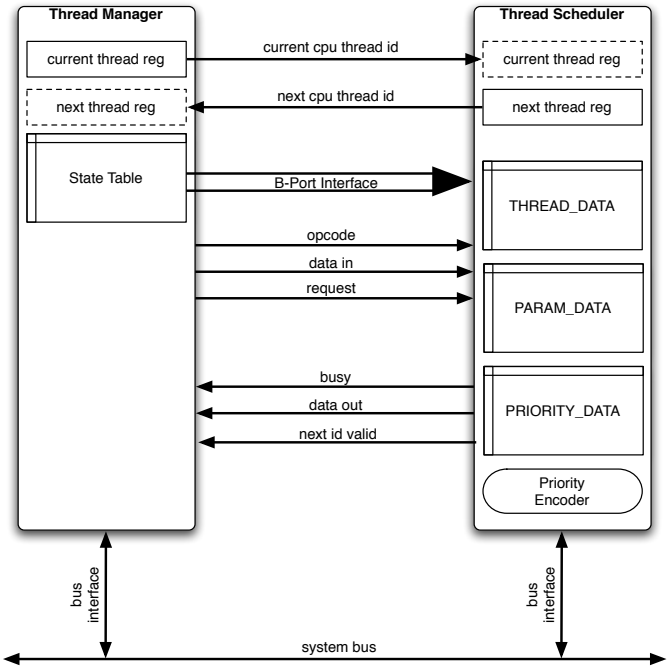

Figure 4. Scheduler Block Diagram

The ready-to-run queue within the scheduler was first implemented as a single linked-list kept in FIFO order [2]. 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 scheduler operations vary directly with the number of active threads. The most recent implementation uses a partitioned ready-to-run queue with a parallel priority encoder in order to provide constant time scheduler operations 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 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. The `Param_Data` BRAM is used to store the 32-bit scheduling parameter which is an overloaded field used to distinguish between software or hardware resident threads. 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 partitioned ready-to-run queue along with the priority encoder eliminate the need to traverse the scheduler'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` and `Param_Data` BRAMs.

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

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06/$20.00 © 2006 IEEE
Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.
IEEE COMPUTER SOCIETY

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

Param_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
s0	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15	s16	s17	s18	s19	s20	s21	s22	s23	s24	s25	s26	s27	s28	s29	s30	s31

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

Figure 5. Scheduler BRAM Data Structures

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 semantics of the enqueue operation can be seen in figure 6. 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 semantics of the dequeue operation can be seen in figure 7. 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 6 and figure 7, 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.

Timing results, which can be seen in table 2, show that the entire dequeue operation completes before the time it takes to complete a context switch, which validates the use of the partitioned queues in FIFO order. If a priority queue (sorted) was used in this situation, then the enqueue operation would take a longer amount of time because it would involve a sorted insert (BRAM traversal). A slow enqueue operation is detrimental to system performance, because enqueue operations are often performed by user-programmed threads which wait for the return status of the enqueue operation. A priority queue implementation would also take additional logic to constantly keep the list in sorted order, so it was decided to design the scheduler module to use a data organization that did not require sorting or BRAM traversal.

5 Results

The scheduler module was implemented on a Xilinx [26] ML310 development board which contains a Xilinx Virtex-II Pro 30 FPGA. Synthesis of the scheduler module yields

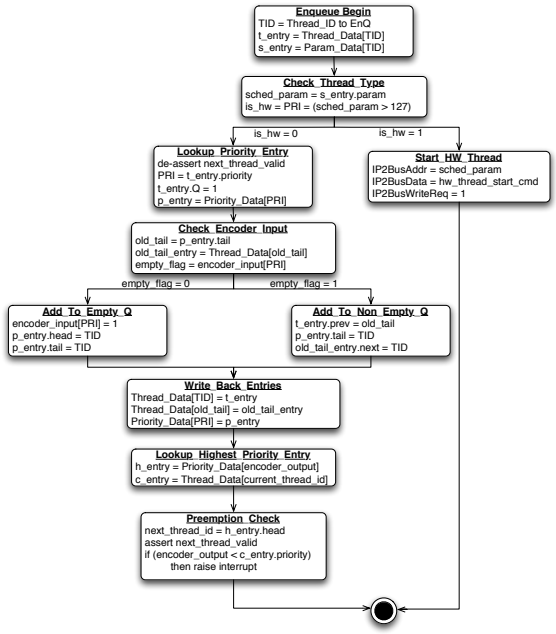

Figure 6. State Diagram for Enqueue Operation

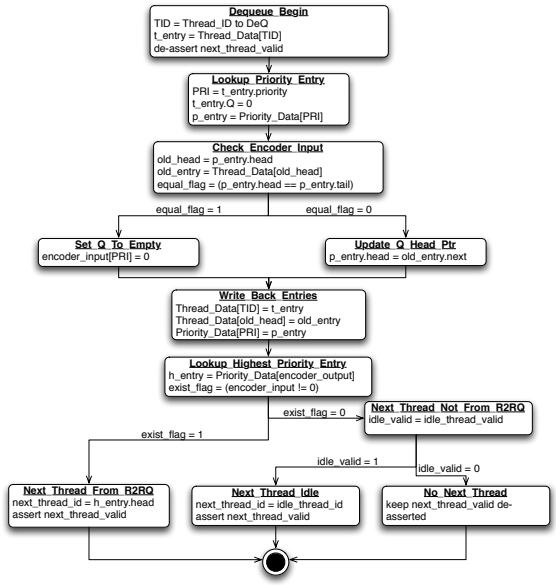

Figure 7. State Diagram for Dequeue Operation

Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.
IEEE COMPUTER SOCIETY
Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1014, Page 000007

Table 2. ModelSim Timing Results of Scheduling Operations

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

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 being able to support a clock frequency within the FPGA of 100 MHz.

The primary scheduler operations have been simulated with ModelSim to characterize their execution times. The worst-case timing results are shown in table 2. 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. From table 2, 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 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, and thus more predictable and precise scheduling of threads within the RTOS.

To test the scheduler in action, it has been synthesized and integrated into the hthreads OS. The first performance measurement taken of the scheduler is end-to-end scheduling delay. End-to-end scheduling delay, or scheduling latency, in the 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 [25, 7]. The end-to-end scheduling delay is measured using a hardware timer that begins clocking immediately after a timer interrupt goes off and would stop timing as soon as the context switch is 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 8 shows a histogram of the end-to-end scheduling delay measurements for our

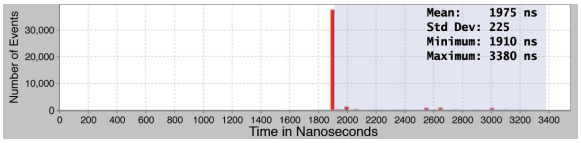

Figure 8. Histogram of Integrated End-To-End Scheduling Delay (250 Active SW Threads)

system with 250 active threads, where each thread is in an infinite loop. The shaded region of figure 8 as well as figure 9 highlight the data range of the measurement from minimum to maximum delay. The end-to-end scheduling delay over a course of approximately 40,000 events is 1.9 μs with 250 active threads, with approximately 1.4 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 [25] and the 40 μs scheduling delay of Malardalen University's RTU [20]. When compared to a commercial RTOS such as RTLinux, end-to-end scheduling delays are typically found to be in the 2 μs range, and with worst case delays in the range of 13 to 100 μs, depending on processor speed [7]. Additionally, further tests have shown that the jitter in end-to-end scheduling delay of the hthreads scheduling system is solely 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. When the hthreads system is run with data cache turned off the jitter in end-to-end scheduling delay is reduced to the nanosecond range.

The second performance measurement taken of the system is interrupt delay. Interrupt delay is the time delay in our system from when an interrupt signal goes off to when the CPU actually enters its ISR. The interrupt delay is 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 9 shows a histogram of the interrupt delay in our system with 250 active threads, where each thread is infinitely looping. This measurement shows that average interrupt delay over a course of 2,500,000 events is approximately 0.79 μs with approximately 0.73 μs of jitter. One can see that the variable 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 variable-length atomic instructions that prevent the interrupt from being acknowledged immediately (CPU dependent).

During a context switch, the CPU does not interact with the scheduler module, which gives the scheduler module

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06/$20.00 © 2006 IEEE
Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.
IEEE COMPUTER SOCIETY

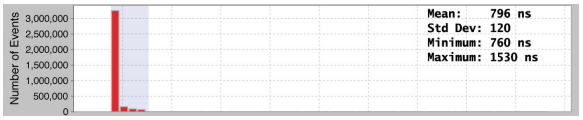

Figure 9. Histogram of Raw Interrupt Delay (250 Active SW Threads)

the perfect opportunity to calculate the next scheduling decision. This scheduling decision is calculated in parallel with the application execution on the CPU, 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 `SCH2TM_next_cpu_tid` register and performs a context switch, and then during this context switch, the `SCH2TM_next_cpu_tid` register is refreshed with the thread that should run at the next scheduling event by the hardware scheduler module.

6 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 [26] 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 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-based OS modules as well as user-defined hardware threads. Additionally, we have been testing a modular design flow for partial reconfiguration in order to dynamically alter the functionality of the IP cores (OS cores and HW threads) resident on the OPB bus.

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 hthreads 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 hthreads project [29], which furthers the future impacts of the project even more. More detailed descriptions of the hthreads OS can be found in [11].

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.

Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1014, Page 000009

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, 2004.

[4] 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.

[5] 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.

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

[7] FSMLabs. RTLinux Performance. FSMLabs Data Sheets on RTLinux. http://www.fsmlabs.com/literature.html.

[8] 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.

[9] A. A. Jerraya and W. Wolf. Hardware/Software Interface Co-Design for Embedded Systems, February 2005.

[10] 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, 2000.

[11] KU HybridThreads. Project Wiki. http://wiki.ittc.ku.edu/hybridthread/Main_Page. Last accessed September 22, 2006.

[12] 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, 2003.

[13] 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, 2003.

[14] 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.

[15] 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, 2003.

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

[17] 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.

[18] Realfast. Realfast. http://www.realfast.se/. Last accessed September 22, 2006.

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

[20] 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.

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

[22] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, 1991.

[23] M. Vuletic, L. Possi, and P. Ienne. Virtual Memory Window for Application-Specifc Reconfigurable Coprocessors. In *Proceeding of the 41st Annual Conference on Design Automation, ACM Press*, pages 948–953, 2004.

[24] M. Vuletic, L. Pozzi, and P. Ienne. Seamless Hardware Software Integration in Reconfigurable Computing Systems. *IEEE Design and Test of Computers*, pages 102–113, 2005.

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

[26] Xilinx. Programmable logic devices. http://www.xilinx.com/. Last accessed September 22, 2006.

[27] T.-Y. Yen and W. Wolf. Communication synthesis for distributed embedded systems, 1995.

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

Authorized licensed use limited to: UNIV OF SOUTH ALABAMA. Downloaded on November 25,2024 at 15:17:54 UTC from IEEE Xplore. Restrictions apply.