hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel

David Andrews, Wesley Peck, Jason Agron, Keith Preston, Ed Komp, Mike Finley, Ron Sass
EECS Department University of Kansas
Information and Telecommunications Technology Center (ITTC)
{dandrews, peckw, jagron, preston, komp, rsass}@ittc.ku.edu

Abstract

This paper describes the hardware/software co-design of a multithreaded RTOS kernel on a new Xilinx Virtex II Pro FPGA. Our multithreaded RTOS kernel is an integral part of our hybrid thread programming model being developed for hybrid systems which are comprised of both software resident and hardware resident concurrently executing threads. Additionally, we provide new interrupt semantics by migrating uncontrollable asynchronous interrupt invocations into controllable, priority based thread scheduling requests. Performance tests verify our hardware/software codesign approach provides significantly tighter bounds on scheduling precision and significant jitter reduction when compared to traditionally implemented RTOS kernels. It also eliminates the jitter associated with asynchronous interrupt invocations.

1 Introduction

This paper describes the hardware/software co-design of a multithreaded RTOS kernel on a new Xilinx Virtex II Pro [16] FPGA. Our multithreaded RTOS kernel is an integral part of our hybrid thread programming model being developed for hybrid systems, such as the new FPGA systems being developed by Xilinx [16] and Altera [1] which integrate general purpose CPUs into the FPGA fabric. The hybrid thread programming model is being developed to provide a uniform high level programming environment in which programmers are able to access the capabilities of the FPGA from within a comfortable and familiar software programming model. A description of our hybrid thread model is given in [2] [3] [11] [6] [4].

The multithreaded RTOS kernel presented in this paper provides a shared memory programming model similar to POSIX Threads [15] for developers of real-time applications. Currently our system supports up to 256 active software threads, 256 active hardware threads, 64 blocking semaphores, 64 binary spin lock semaphores, preemptive priority, round robin, and FIFO scheduling algorithms. Applications access the hardware based operating system components through familiar APIs such as create, join, and exit. However, these operations have now been efficiently migrated into state machines which operate in parallel within the FPGA. These operations cause changes in the state of an application thread and may lead to making new scheduling decisions. These scheduling decisions are performed in hardware, similar to [7] and [12], allowing the overhead of a scheduling decision to be reduced into a single memory read operation. Thus, a deterministic context switch is the only major source of overhead on the CPU. Migrating the scheduler into hardware has also allowed us to address the major source of jitter within real-time and embedded systems: asynchronous interrupt invocations. Our RTOS kernel now contains a new CPU Bypass Interrupt Controller (CBIS) that transforms the semantics of asynchronous interrupts into that of synchronous and controllable thread scheduling requests.

2 hthreads: A Multithreaded HW/SW RTOS Kernel

All operating system services introduce some general level of overhead that results in a reduction of execution capacity for application programs on the CPU. A simple hardware/software co-design of existing system services can reduce this overhead and therefore provide more CPU capacity for application programs [9] [12] [14]. Although a necessary step, simply migrating overhead functionality into the hardware is not sufficient for addressing the more critical issues that challenge developers of real-time systems. Specifically, developing an operating system with minimal jitter, deterministic behavior, and fine scheduling granularity have by far been the most challenging goals for real-time operating systems designers. To achieve the best and tightest bounds on application program scheduling, migration of both the scheduler processing as well as the management of state information must be moved off of the CPU and into hardware components. If the scheduler has access to all necessary state information then the scheduler can perform look-ahead processing concurrently with the application program without interrupting or requiring processing time on the CPU. Providing the scheduler with the necessary state informa-
tion requires redirecting all scheduler entry invocations to the hardware, including API calls that cause state changes in threads, blocking operations such as semaphores, and more importantly, interrupts. This requires a complete re-partitioning, and in some cases, re-interpretation of traditional operating system semantics.

Figure 1 shows a block diagram of our co-designed components. This co-designed real-time kernel consists of

1. a hardware resident thread manager
2. a hardware resident thread scheduler
3. hardware resident semaphores
4. a CPU Bypass Interrupt Scheduler (CBIS)
5. software API’s, context switching, and kernel code

The semaphore components shown in Figure 1 are outlined in [6] [4] and are not discussed further in this paper.

2.1 Software Thread Manager (SWTM)

Figure 2 shows the block diagram of our Software Thread Manager (SWTM) whose main function is to store and modify thread state and serve as the interface for all state change requests. State change requests can be caused by an application thread, by hardware resident application threads, by the system jiffy timer, by our semaphore components, or by the CBIS. In order to allow efficient execution of the thread management and scheduling operations, the SWTM and scheduler component have a dedicated interface which is used to communicate thread scheduling requests from the SWTM to the scheduler. The state of all threads are also maintained in the hardware by using state tables stored in the Virtex II Pro’s Block RAM (BRAM).

All thread management functions are invoked by reading or writing one register in the set of memory mapped registers shown in Figure 2. A write or read operation on a register invokes a state machine that performs the associated processing and updates the state table. As an example, Figure 3 shows the pseudocode for our co-designed thread create API with the pseudocode executed on the CPU appearing on the left of Figure 3, and the resulting actions performed by the hardware resident state machine shown on the right.

Each register shown in Figure 2 is not a physical register but instead is a virtual register which can have a variable depth. This helps save precious CLB resources inside of the FPGA. The depth of a given register specifies the number of successive, 32 bit address locations that comprise the register. Registers having a depth greater than one are utilizing the least significant address lines to encode parameters, such as a thread identifier, to be passed to the register. In most instances, the 32 bit address is encoded as:

```
31 8 7 0
```

where the lower 8 bits contains the identifier of the thread which is requesting the operation. This encoding allows the user of the API to both send information with a request and receive results from that request in a single, atomic bus transaction. This approach was taken for two reasons. First, implementing physical registers within an FPGA requires CLB resources so their use should be minimized. Second and more importantly, implementing a request more traditionally as a store instruction would require an additional load instruction to read back returned values. The combination of store and load instructions would then need to be protected by a critical region. This is a fundamental concern when partitioning functionality between distinct software and hardware components that can be interrupted. Although critical regions can be protected by semaphores, this introduces additional performance delays and semaphore resource requirements. For these reasons we chose to use “virtual” registers where
each virtual register corresponds to a single command whose parameters are encoded in its memory address. By doing so, we create truly atomic operations implemented as single instruction read requests. This approach trades off additional address space in place of physical resources. Additionally, the approach simplifies to coding of APIs and provides an inherent consistency within the system. The atomicity of an operation is guaranteed by the SWTM that operates as a slave device on the bus and controls the termination of the bus transaction. Once the register base address and thread identifier is decoded, the SWTM invokes the appropriate state machine and places any return data directly onto the data bus. For operations where no data is returned, the SWTM performs an immediate bus acknowledgement when the address has been decoded. The state machine can then run concurrently with the application program on the CPU. For operations that require return status, the SWTM holds the bus until the state machine can determine what return status to place on the data bus. All operations within the SWTM are performed by state machines that transition at the system clock rate and complete within tens of clock cycles. This approach ensures that all SWTM operations are atomic, deterministic, and have minimal overhead.

2.2 Scheduler Component

A block diagram of the scheduler component is shown in Figure 4. This component currently implements FIFO, round robin, and priority based scheduling algorithms. The scheduler was initially developed as an integrated component within the SWTM, but shortly afterwards it was separated out as a standalone component to promote portability for system designers and to allow implementation of custom scheduling algorithms.

The primary functions of the scheduler are to determine the thread identifier of the next thread in the system to run, and to determine if a context switch interrupt should be generated to the CPU. A context switch interrupt is only generated when a thread has the highest priority of any thread in the ready-to-run queue and its priority is higher than that of the currently running thread. This type of interrupt semantics allows the CPU to remain undisturbed until the scheduler decides that a context switch is necessary.

The scheduler must maintain its own ready-to-run queue data structure within the FPGA in order to provide fast access to each thread’s scheduler attributes (priority, next thread in queue, previous thread in queue, etc.). The scheduler module uses a partitioned ready-to-run queue with a parallel priority encoder that allows all scheduler operations to execute in constant time. The partitioned ready-to-run queue has been implemented using two BRAMs: the Priority Data BRAM and the Thread Data BRAM. The Priority Data BRAM is indexed by priority level and contains the head and tail pointers of the ready-to-run queue for that priority level. The Thread Data BRAM is indexed by thread identifiers and contains the thread attribute information including priority level, queued/not-queued flag, a pointer to next thread in queue, and a pointer to the previous thread in queue. Additionally, a parallel priority encoder continually calculates the highest priority in the system using a 128-bit register field that represents which priority levels have active (queued) threads. The parallel priority encoder calculates the highest priority in the system only when a change occurs in its 128-bit input register in a quick and constant four clock cycles. The parallel priority encoder along with the partitioned ready-to-run queue eliminates the need to traverse the scheduler’s ready-to-run queue as individual priority queues can be accessed using the Priority Data BRAM and individual thread attributes can be accessed
using the Thread_Data BRAM.

State change requests to the scheduler are initiated by

1. the software thread manager submitting a new thread identifier to be queued, or

2. the CPU reading the Next_Thread_ID register during a context switch.

During and an add thread operation the SWTM will first perform error checking and then update its state information before providing the thread identifier of the new thread to add to the scheduler’s ready-to-run queue. All communication is done over the dedicated hardware interface between the SWTM and the scheduler. For this type of queuing operation the scheduler performs the following sequence of actions. First, the thread identifier is linked into the bottom of the ready-to-run queue corresponding to its priority level. Second, the parallel priority encoder’s 128-bit input register is updated as necessary for the current enqueue operation. Third, the output of the parallel priority encoder is read and is used to access the head pointer of the highest priority level’s ready-to-run queue out of the Priority_Data BRAM. Fourth, the head pointer is placed in the Next_Thread_ID register. Finally, a preemption check is carried out by accessing the priority level of the next thread to run and the priority level of the currently executing thread. The priority levels are compared to one another and if the priority level of the next thread to run is better than that of the currently running thread then a context switch (preemption) interrupt is generated to the CPU.

It is important to note that when a new thread identifier is queued, the scheduler does not search through the ready-to-run queue or perform any type of priority sorting. Instead, the following three operations are performed: an enqueue of the thread to the tail of a list, a lookup of the head pointer of the highest priority queue in the system, and a comparison of the priority level of the next thread to run and the priority level of the currently running thread. This approach is not only very fast (13 clock cycles) but, more importantly, executes in constant time which eliminates the jitter introduced in traditional software schedulers that must sort or search through a single ready-to-run queue.
A new scheduling decision is initiated immediately upon the CPU reading the Next_Thread_ID register during a context switch. When the Next_Thread_ID register is read the scheduler must remove the thread identifier found in the Next_Thread_ID register from the ready-to-run queue and calculate the new next thread to run. The new next thread can be calculated in constant time by reading the output of the parallel priority encoder and using it to lookup the head pointer of the highest priority level’s queue. This thread identifier is then placed in the Next_Thread_ID register and the next scheduling decision has been completed without any linked list searching or traversal. This process happens in constant time, regardless of the number of queued threads. It also executes without introducing any jitter into the system. The state machine which implements this operation is shown in Figure 6. Figure 5 shows that the next scheduling decision is always pre-calculated before a context switch completes.

The scheduler component implemented on a Xilinx MI310 development board requires 1034 out of 13696 slices (7.5%), 522 out of 27392 slice flip-flops (1.9%), 1900 out of 27392 4-input LUTs (6.9%), and 2 BRAMs (1.5%). The module has a maximum operating frequency of 143.8 MHz, which easily meets our goal of a 100 MHz clock frequency.

### 2.3 Precise Interrupt Control

By far the largest sources of jitter within real-time and embedded systems are interrupts. As the integration of embedded and real-time systems continues to increase, it is becoming even more critical to address the jitter injected from the processing of external interrupts. Software approaches such as those employed by RTLinux [5], RTAI [13] and KURT [8] minimize the effects of this jitter using mark and delay processing techniques. Although these approaches can delay the major portion of interrupt processing overhead, they still incur the initial overhead costs of context switching and executing the equivalent of “top halves” of interrupt service routines (ISRs) to mark the pending interrupt. The jitter of context switching and ISR processing cannot be eliminated by software techniques that must first field the interrupt to determine if it should be processed. To completely eliminate this jitter requires evaluation of the request within a computational component external to the application processor.

To minimize this jitter and provide precise control over external interrupts we direct interrupt requests to our hardware-based thread scheduler through a new hardware component that binds interrupt requests to thread identifiers. From a systems perspective this has the positive effect of unifying the priorities of application threads and external interrupts under a single scheduling framework. Within our approach, the semantics of interrupt handlers are modified so that they are standard, schedulable threads. Additionally, routing interrupt processing requests directly to the hardware scheduler component allows the continued or resumed execution of the thread to be considered in relation to the state of all other system threads. This provides programmers with the ability to dynamically modify the priority of an interrupt depending on its request rate, processing time, and criticality. Figure 7 shows a block diagram of our CPU Bypass Interrupt Scheduler (CBIS) which takes the place of the traditional priority interrupt controller (PIC). The CBIS allows simultaneous interrupts to request processing by storing the requests into the pending latches similar to the operation of existing priority interrupt controllers. However, the semantics of the processing sequence for pending interrupts within the CBIS vary from existing PICs. Each interrupt channel (interrupt number) exists within one of three states shown in the state machine for a single interrupt shown in Figure 8.

In the default state, no interrupts are pending, and no thread has registered for the interrupt. Two scenarios can occur from this state:

1. an interrupt request from a device is asserted before any thread has requested that the interrupt be associated with its thread identifier, or
2. a thread requests to be associated with the interrupt before the interrupt has been asserted

In the first scenario, the CBIS state machine marks the interrupt as pending, but does not issue a scheduling request.
When a thread issues an associate request for this pending interrupt, the CBIS sets the valid bit, representing a pending request, and returns the pending status but does not transfer the thread identifier to the scheduler as the thread is already running on the CPU. Thus, no context switching overhead is incurred and the exception handler thread may continue to run if it is the highest priority thread. In the second scenario, when the thread first associates with no pending interrupt, the CBIS writes the thread identifier into the request table, sets the valid bit, and returns a no pending status to the software kernel.

When a thread associates its thread identifier with an interrupt the CBIS will return a status code indicating either that an interrupt is already pending so that the thread should continue running or that there is no interrupt pending so the thread state should be changed from running to blocked. If the thread is blocked because there is no interrupt pending then the CBIS will place that thread’s identifier into the ready-to-run queue once that interrupt occurs.

### 3 Performance

Performance results are shown for

1. basic scheduling operations of threads with no external interrupts
2. the effect of modifying the semantics of interrupts into priority-based thread scheduling requests

We have analyzed our thread APIs with no external interrupts to closely study the actual scheduling delays and jitter of our hardware/software co-designed multithreaded kernel. All interrupts were disabled during these tests to eliminate external jitter. More recently, we have begun to quantify the effects of modifying the semantics of the interrupts into thread scheduling requests. We present our initial testing of a set of simple, periodic, schedulable tasks and measured performance gains of our approach versus standard interrupt semantics.

#### 3.1 Thread Scheduling

The following timing results were generated on a MEMEC [10] development board with a Virtex II Pro 7 series hybrid chip using the traditional Xilinx priority interrupt controller. The CPU clock speed was 300 MHz, and the PLB, OPB, and FPGA clock speeds were 100 MHz. The scheduling delay times shown in Figures 9, 10, and 11 represent total end-to-end delay times including the time taken by the CPU to acknowledge the interrupt, context switch into the non-critical interrupt service routine, handshake the PIC, enter the thread context switch code, save the old threads context and stops immediately...
before control returns back to the newly scheduled application thread. The measurements were obtained by creating an additional counter register in the timer hardware that resets to zero on the clock cycle when a scheduling interrupt is asserted to the PIC, and then continues to increment at the 10 nsec clock cycle time. The free running counter was then read at the very end of the context switch code, immediately before the new thread runs.

Figures 9, 10, and 11 show mean scheduling delays of 1.97 \( \mu \)secs, 1.97 \( \mu \)secs, and 1.75 \( \mu \)secs and maximum scheduling times of 3.3 \( \mu \)sec, 3.1 \( \mu \)secs, and 2.1 \( \mu \)secs for 250, 128, and 2 active threads respectively. Although low, there is still small jitter (1.4 \( \mu \)secs, 1.2 \( \mu \)secs, and 0.3 \( \mu \)secs) in the system. To identify the source of the jitter, we ran the same tests with the data cache, instruction cache, and both data and instruction caches turned off. With both caches off, the maximum jitter for 2 through 250 threads was around 10 clock cycles (100 nsecs). This is a strong indication that the jitter observed in the scheduling times is a result of data cache misses that occur during the saving and restoring of the interrupt routine and thread context switches and should not be attributed to the hardware or software implementations.

Figure 12 shows the 0.8 \( \mu \)sec delay time from the time that the interrupt request is generated from the hardware based scheduler, through the CPU acknowledgment, context switching to the ISR, handshaking the interrupt controller and determining that the kernel code should run. This RAW ISR delay time represents over 30% of the total scheduling time delay.
3.2 Interrupt Control

Figure 13 shows a graphical representation of interrupt processing control through simple changing of priorities. For our experimental setup, we set a complete scheduling period of 100,000 clock cycles, or 1 msec and ran 1,000 periods per test run. Our low overhead, low jitter multithreaded programming model allows us to support such a fast scheduling cycle because of our fine-grained and lightweight threads. We generated a periodic interrupt, a fast scheduling cycle because of our fine-grained and threaded programming model allows us to support such a fast scheduling cycle because of our fine-grained and lightweight threads. We generated a periodic interrupt, with a period of 625 µsec that continued throughout the complete test run. In order to measure the overhead due to context switching only, our interrupt service routine contained no processing and simply returned. This allowed us to isolate the overhead and jitter due just to the invocation effects of the ISR. Figure 14 provides worst case finish times in clock cycles for our four threads, and Figure 15 shows the corresponding finish time jitter. The first rows in both figures show our ideal case with interrupts disabled. Thread C1 has a worst case finish time of 311 µsec, C2 494 µsec, NC1 570 µsec, and NC2 601 µsec. The corresponding jitter for C1 is 50 nsec, C2 100 nsec, NC1 120 nsec, and NC2 160 nsec. The second row of both figures (labeled Interrupt), shows the corresponding worst case finish times and jitter for normal asynchronous interrupt requests. As seen in Figure 15, substantial jitter is introduced in performing traditional interrupt context switching, even for a low frequency (1.6 interrupts per scheduling period) interrupt.

The next two rows show the effects of routing the interrupt requests through the CBIS. In these cases, the interrupt request does not cause an ISR invocation. Instead, the interrupt is translated to a thread scheduling request and routed directly to our hardware resident scheduler. In the third row, the priority of the interrupt routine is set between the priorities of the critical and non-critical threads. The jitter effects are substantially eliminated from the critical threads and migrated into the non-critical threads. The last row shows the effects of lowering the interrupt service routines priority to below the non-critical tasks. In this case, the finish times and jitter are within a few clock cycles of the no interrupt base case. This is intuitive as we now delay the processing of any interrupt during the scheduling period until after the threads have run.

4 Conclusion

In this paper we have presented the hardware/software co-design of our hthreads real-time multithreaded operating system kernel. Our co-designed operating system takes advantage of new hybrid CPU/FPGA chips as an enabling technology to provide a level of scheduling precision not obtainable from traditional software based operating systems. Our current system supports standard multithreaded operations through portable APIs, allowing 256 simultaneous threads with 128 priority levels, and 64 semaphores.

References