Symmetric Multiprocesser Design for Hybrid CPU/FPGA SoCs

Shane R. Santner

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. Arvin Agah

Date Defended

© 2007 Shane R. Santner

The Thesis Committee for Shane R. Santner certifies
That this is the approved version of the following thesis:

Symmetric Multiprocesser Design for Hybrid CPU/FPGA SoCs

Committee:

____
Chairperson

____

____

____
Date Approved

Abstract

This thesis presents the design of a Symmetric Multiprocessor (SMP) hybridthreads (hthreads) system that allows multiple threads to execute in parallel across multiple processors controlled by a single hardware scheduler. This approach increases the performance of software at a minimal cost to hardware. The issues that must be addressed for extending a uniprocessor kernel include system initialization, processor identification, context switching and concurrency control. As a proof of concept this thesis shows how hthreads, an existing hardware/software co-designed kernel can be extended to control multiple processors from a single, centralized hardware scheduler. Analysis results from executing on hardware reveal that for computationally intensive programs the typical speedup is in the range of 1.65x. This shows improvement in system performance while also illustrating issues associated with bus arbitration and memory access times.

Contents

Acceptance Page **i**

Abstract **ii**

1 Introduction **1**

2 Statement of Problem **4**

2.1 SMP Systems, Coherency and Concurrency 4

2.2 Thesis Contributions 7

3 Background **9**

4 Related Work **13**

5 System Initialization **16**

5.1 Hardware Setup 17

5.2 Software Setup 21

6 Design & Architecture **26**

6.1 SMP Hardware Design 26

6.2 SMP Software Design 39

7 Implementation Results **44**

8 Conclusion **49**

9 Future Work **51**

References **55**

List of Figures

5.1 High-level overview of the SMP hthreads system 18

6.1 Illustration of the check_preempt function implemented in the hardware scheduler 31
6.2 Illustration of the SYSCALL_LOCK implemented in the hardware scheduler 39
6.3 hthread data structure 40
6.4 Entry and Exit paths for acquiring system call locks 41

7.1 simpletest.c results 46
7.2 simple_buffer.c results 47
7.3 dual_test.c results 48

List of Tables

6.1 Command Set Additions and Modifications to the Hardware Scheduler Module 32
6.2 Command Set Additions and Modifications to the Thread Manager Module 36

Chapter 1

Introduction

Multiple processors embedded on the same die are becoming common in today's reconfigurable computing world. Advances in multicore scheduling support [3] provide increased performance on software based systems. The hybridthreads project at the University of Kansas [10] extended thread scheduling support to a hardware/software co-design. The work of this thesis builds upon the previous hthreads design, expanding it to provide support for multiple processors.

SMP architectures consist of two or more identical processors connected to a single shared main memory. This shared access allows multiple threads of execution to run on several different processors concurrently. One of the features of an SMP architecture is the concept of processor affinity which enforces processor specific execution of threads. This feature was developed because many applications rely heavily on cache utilization [11], however the functional SMP hthreads system does not use data caching and therefore the use of processor affinity is outside the scope of this thesis. Another feature of SMP architectures is load balancing. Typically the scheduler will assume the role of attempting to balance the workload for each processor in the system. This implementation in the SMP hthreads sys-

tem occurs through the check_preempt function which will be discussed in greater detail in section 6.1. Using the hthreads system as the foundation for the design, the next logical step is to migrate the existing hardware/software co-design to account for both PowerPC Processors embedded in the Virtex-II Pro FPGA.

Modifying the hardware scheduler and thread manager to account for the additional processor is required. Additional registers are also needed to track this new information that must be stored. The finite state machine of the hardware scheduler will also require significant modifications to many of the internal states. Traditional scheduling operations such as ENQUEUE and DEQUEUE will now have to determine which processor is requesting the operation. Also, with the addition of another valid processor into the system the hardware design will have to be modified so that interrupts are sent to the appropriate processor.

As complex as the changes will be to the hardware portion of the design, the software changes will require an even greater amount of modification. SMP system initialization will be the first obstacle to overcome. This involves requesting a processor id from the thread manager, creating a separate stack for each processor to use during initialization, setting up mutexes to perform thread-safe memory allocation and deallocation, creating a separate stdout for each processor if requested, creating the initial main and idle threads and finally enabling preemption. Interleaved with SMP initialization issues are run-time issues such as memory allocation and deallocation, cache coherency and concurrency control. These potential problems will need to be addressed and resolved to achieve a functionally correct and stable system.

The remainder of the paper is partitioned as follows: The problem statement of this thesis work is developed in chapter 2 with a list of contributions shown in

section 2.2. A brief background of the uniprocessor hthreads design is discussed in chapter 3. Related work including the latest SMP application note [30] by Xilinx is covered in chapter 4. System initialization relating to the hardware setup and software initialization (including processor identification) are discussed in chapter 5, sections 5.1 and 5.2. The hardware and software design is covered in chapter 6, with a detailed look at the hardware portion in section 6.1 and the software portion in section 6.2. Performance results of the various versions of the design are shown in chapter 7. Finally, a summary will be provided in chapter 8 with future work being discussed in chapter 9.

Chapter 2

Statement of Problem

2.1 SMP Systems, Coherency and Concurrency

Why migrate the current design of hthreads to be SMP compatible? Quite simply, there exists an additional resource with tremendous computational power that is sitting idle on the FPGA. If this resource could be used then two threads of execution could be run in parallel with the potential to linearly increase the overall system performance. This would be useful for a wide range of applications where performance and real-time execution are critical factors.

In general, hardware/software co-design is very difficult. Unforeseen bugs in the hardware can cause bugs in the software with the opposite also holding true. Furthermore, predicting the interaction of the hardware portion of the design with the software portion can also be extremely difficult, leaving the developer frustrated. Fortunately, much of this frustration was handled during the initial hthreads design. The hthreads project has matured to a point where much of the interaction between the two contrasting sides of development is stable. Although this is helpful, debugging real-time system-level software issues that might also

be related to hardware issues can be nearly impossible. Often the majority of the time spent debugging is to determine the problem, rather than correcting it. This is mainly due to the lack of debugging tools available for SMP systems, and more specifically for System On Chip (SoC) SMP systems. Without these critical tools available to isolate software and hardware issues, finding bugs (if they are found at all) can take a significant amount of time and effort.

Hardware limitations can also prove to be insurmountable obstacles when attempting to design something unique to the field of study. This will later be shown to be a profound impediment to system performance. The first hardware limitation encountered was the arbitration scheme used for the Processor Local Bus (PLB). Both PowerPC's connect to the PLB, and Xilinx's PLB arbitration scheme allows one processor to always be granted access to the bus in the case of a tie [28]. Under heavy PLB load, this equates to one processor sitting idle while the other processor gets exclusive access to the bus. The worst case scenario would nearly reduce the SMP system to a uniprocessor design. The second limitation to the hardware is the lack of cache coherency. Xilinx does not bring out the necessary hardware lines from the PowerPC's to implement a standard snoopy cache protocol [27], therefore when data is shared among resources the possibility exists for that data to be incoherent. Typically this can be avoided completely by turning off data cache for the entire system which is the approach taken by the uniprocessor hthreads design. This is also the approach taken for the SMP hthreads design which will be discussed further in section 6.2.

Finally, intrinsic to all SMP systems is the concept of concurrency control. This refers to the idea of multiple threads of execution requesting the same resources at the same time. Consequently, which resource is granted to which thread(s)

and what process is used to ensure that race conditions are avoided needs to be determined. To further complicate the implementation of concurrency is the hthreads design itself. Because hthreads uses soft hardware cores to perform many traditional operating system services, the typical methods to ensure concurrency control may not be available to use, or if they are available then the scope of what is useful to the SMP hthreads system might be limited. For example, in the uniprocessor hthreads design a system call will typically contain a combination of software routines with soft hardware core requests. This means that critical thread information is stored in hardware registers specific to the soft hardware cores and in system software data structures. To execute a system call correctly in an SMP hthreads system it may be necessary to lock around the entire system call, or just the hardware or software portions of the call. This is also highly dependent on which system call is requested and the data structures that are modified during the system call. To implement concurrency efficiently in any system the goal is to maximize the amount of parallelism that is present in the system, yet still ensuring correct functional operation. Therefore, it would be inefficient to provide a single heavy-weight lock around all system calls because it may not always be necessary to lock down every system call available to the user. There could be dozens of pairs of system calls that would not require concurrency control, or that might only require a small amount of concurrency control and therefore locking around all system calls could as much as double the amount of time it would take for both processors to complete their requests. This difficulty and others closely associated to SMP concurrency control stem from the fact that in the initial hthreads design only a minimal effort was put into making it SMP compatible for future revisions. This meant that only two of the three standard

types of concurrency were considered, thread and interrupt - physical concurrency was not a priority at the time. Consequently, system-level software was not written to take into account the architectural concepts of SMP systems, which means that modifying the system-level code will be extremely challenging.

2.2 Thesis Contributions

- Modified hardware scheduler and thread manager to account for multiple processors
  - Designed the check_preempt hardware function which is used to schedule multiple threads of execution on multiple processors
  - Added multiple states to the finite state machines in the hardware scheduler and thread manager
- Developed an initialization routine to synchronize the hardware and software
  - Processor Identification Assignment
  - Unique stack creation for each processor
  - Idle thread creation
  - Interrupt initialization
- Implemented multiple hardware locks in the hardware scheduler for concurrency control and system stability
  - Processor lock used to protect system-level software
  - Thread specific lock which is used by __malloc_lock() and __malloc_unlock() for thread safety

- Created custom hthreads kernel software
  - System initialization software
  - Concurrency control software used in conjunction with the hardware scheduler locks

Chapter 3

Background

The hthreads project was developed to allow pthreads applications to be migrated to a hybrid FPGA/CPU SoC to increase the performance of multi-threaded applications. To achieve this goal the hthreads design uses a hardware/software co-design to implement system services that are available to the user application. These services are part of the hthreads kernel which provides the majority of the standard pthreads system calls to the application. Many of these services are controlled by software, however to increase performance some operations were moved into hardware. These services include operations involving mutexes, thread management (such as create, join, exit, etc.) and scheduling. Moving this functionality into hardware increases system performance, however the difficulty of achieving a functionally correct design becomes challenging.

The first system software service to be migrated into hardware involved mutex operations. It was observed that mutex operations required a significant portion of processor clock cycles to satisfy requests. Soon after, the mutex manager was created which is a soft hardware core used to grant and release mutexes. Because the pthreads standard requires the implementation of pthread_mutex_lock

and pthread_mutex_unlock [12] for correct thread locking/unlocking operation, the mutex manager created blocking, non-blocking, recursive and spin-lock type mutexes to enforce this standard.

The thread manager and hardware scheduler [1] are closely related soft hardware cores due to the nature of the operations they perform. Originally they were combined into one hardware core - only later in the design was the decision made to separate their combined functionality. They are so tightly intertwined that the decision was made to implement a dedicated communication channel between them. This would eliminate OPB transactions when the thread manager requests an operation from the hardware scheduler. Because of the recurring nature of these operations, this design decision improves overall system performance.

The thread manager was created to perform a managerial type of role within the design. Examples include allocating and clearing thread id's and tracking thread specific information in hardware registers. The thread manager handles many typical pthreads operations, many times in conjunction with the scheduler. Common thread specific requests that are serviced by the thread manager includes hthread_create, hthread_join and hthread_exit. The thread manager implements this managerial type of role by providing an interface among the processing components in the system and the hardware scheduler. This interface among resources tracks the thread-specific data as it passes through. This thread-specific information is then used to perform hthread system calls and ensure correct functional output.

The hardware scheduler core is used to implement the ready-to-run queue for the hthreads system. Using information stored in local BRAM's the hardware scheduler makes scheduling decisions about which threads on the ready-to-run

queue should be executed next. Offloading these features from the kernel code has been shown to reduce the jitter [1] in the system and consequently increase the overall performance of user applications running on the hthreads system. The hardware scheduler implements the ready-to-run queue as a fixed-priority FIFO queue to perform scheduling operations. The queue contains 128 priority levels with each level containing a pointer to the thread id for the head and the tail of the FIFO. The scheduler reduces the jitter in the system by eliminating unnecessary interrupts to the processor. Consequently the scheduler will only interrupt the processor if a thread with a better priority is ready to run. This is in direct contrast to software schedulers which repeatedly interrupt the processor to check if there exists a better thread id to run. In the case where the thread with the best priority is executing then the interrupt sent to the processor results in a significant waste of processor clock cycles. Again, this scenario is eliminated by using the hardware scheduler.

The glue that holds the hthreads hardware/software design together is the hthreads RTOS kernel [4]. This system-level software seamlessly provides hthreads services to the user application. The kernel handles system calls, interrupts and I/O for the user application. This is done through a combination of memory-mapped reads to the soft hardware cores and system-software data structures which also track thread-specific information. The existing kernel design approach is to limit the amount of time required to perform system services while carrying out the pthreads POSIX standard. Migrating this portion of the design to become SMP compatible will be the most challenging of all the hthreads system components.

Recently, the hardware thread interface (HWTI) [2] has been added as an

additional component into the hthreads system. This is a custom hardware core that is wrapped around hardware threads and is used to perform services such as reading from and writing to memory. This interface also allows hardware threads to communicate with other threads in the system. This is a key component of the hthreads system because it eases the amount of work required for a developer to transition a software routine into hardware. This transfer from software to hardware also holds the greatest opportunity for speedup, which is why the HWTI is so critical.

In summary, hthreads will allow a typical pthreads application to spawn multiple threads of execution which can run concurrently in software and hardware. Furthermore, the operating system has reduced overhead because tasks that are typically serviced by the kernel have been moved into hardware where they can run in parallel with the operating system. These vital components of the hthreads system allow significant speedups which directly benefit the converted pthreads user application.

The hthreads design has matured significantly over the life of the design, and the next logical progression is to use both PowerPC processors that are embedded in the Xilinx FPGA. This has the potential to increase the performance of the system without substantially increasing the amount of space required to implement the design in the FPGA. The second processor is valuable computational resource that has been idle, and the opportunity to modify the design to take advantage of this idle processor is promising. The modifications necessary will be substantial, requiring significant changes to the thread manager and hardware scheduler to account for the additional processor. Furthermore, the software will require substantial modifications to ensure that the system is stable and yet efficient.

Chapter 4

Related Work

The earliest related work found was done by Dr. John K. Bennett at the University of Colorado at Boulder [6]. He was able to construct an architecture that could use both PowerPC's embedded in the Xilinx Virtex-II Pro FPGA. His approach employed two PLB's, one for each PowerPC. He also placed the DDR memory on the OPB bus which requires a bridge from the PLB to OPB in order for the PowerPC's to gain access to main memory. Also, he used local block RAM's (BRAM's) for each PowerPC to store the application program. Finally, he considered the UART to be a shared resource between both processors and therefore used PowerPC specific instructions to create atomic operations to institute locks around gaining access the UART. The lock was placed in a shared BRAM (shared by both PowerPC's) section of memory and a simple test was executed to demonstrate both PowerPC's running in parallel. Although this design illustrates the possibility of using both PowerPC's concurrently on the Virtex-II Pro FPGA, it does not perform any computationally intensive tasks. The design itself is targeted as a basic introduction into multiprocessing using the embedded PowerPC's on the Virtex-II Pro FPGA. Also, this design does not use an operat-

ing system and therefore does not implement multiple threads of execution which can be queued, blocked or transferred between processors in the system.

A Research Accelerator for Multiple Processors (RAMP) [13] is a relatively new project from the University of California, Berkeley in which they attempt to provide an architecture that can support hundreds of processors executing concurrently. While the scope of this project is much larger than the scope of SMP hthreads, some interesting similarities exist. RAMP White is a subset of the RAMP project with the focus on coherently sharing memory among multiple processors [5]. In their research they have identified several possible solutions to the cache coherency issue with timelines for completion. Depending on the outcome of this work, it could certainly be ported to the SMP hthreads design in the future to allow data caching within the system.

May 10th, 2007 Xilinx released xapp996 citexapp996 which is their dual processor reference design suite. In this application note they provide three different scenarios for running dual processors. The first example illustrates running two Microblaze processors in parallel. Next they show two PowerPC's running concurrently, and finally they illustrate how to run one Microblaze and one PowerPC concurrently. Initially the dual PowerPC example looks very promising, however they use a different linker script for the same application to ensure that each program is stored in a separate location in main memory. In direct contrast to this approach, the SMP hthreads design seeks to execute software by both PowerPC's stored in one location in main memory. Also, the application note provides four different software applications which test the functionality of both processors executing concurrently. These examples do illustrate concurrent software execution, however they are not multi-threaded examples and do not require a kernel for ex-

ecution, therefore like the previous work they are somewhat trivial in comparison to the capabilities of an SMP hthreads system. These differences aside, the hardware setup included in the microprocessor hardware specification (mhs) file was useful to check the implementation of the hardware design for the SMP hthreads system. Although Xilinx has yet to develop a true SMP system using their line of Virtex FPGA's, progress is being made.

Chapter 5

System Initialization

SMP systems require a substantial amount of setup before any custom hardware/software co-design can be carried out. Several issues have to be considered, such as setting up the initial hardware to function correctly. In a broad sense this includes processor initialization, bus connections, memory configuration and custom hardware core setup. Once the hardware setup is complete, the focus then turns to the software setup which includes modifying the software kernel to account for multiple processors and editing configuration files to account for the new SMP modifications during the software build cycle. Once the hardware and software have been correctly setup and initialized the next step is to acquire a processor identification number (PIN). This allows the kernel to request services from the various hardware cores based on which processor is requesting the service. Finally, idle threads will be defined and issues involving multiple processors and idle threads will be addressed.

5.1 Hardware Setup

The Xilinx University Program (XUP) development board was used to setup and test the hthreads SMP system. The hardware contains a Virtex-II Pro FPGA with two embedded PowerPC's, a DIMM slot for DDR SDRAM memory, ps2 ports for a mouse and keyboard, a VGA connector for communicating with an external monitor, an RS232 and ethernet connector for serial I/O, a USB connector for downloading bitstreams, a microphone input and two audio connectors. This development board from Xilinx is targeted to the undergraduate engineering student who can purchase this board as part of their curriculum and use it throughout their academic career. That being said, much of what is actually on the board is irrelevant for the hthreads system to function properly, therefore a significant portion of the hardware is not used. The interesting portion of the hardware is the on-chip setup - the intricacies of what is involved with setting up the system to function properly when using both PowerPC's embedded within the FPGA. The following sections will cover the hardware configuration of the SMP hthreads system with the necessary modifications to allow utilization of both processors.

The hardware configuration for the SMP hthreads system is a combination of processors, bus architectures, memory and hardware cores. The performance of the processors embedded in the FPGA will be severely crippled if they cannot access shared resources such as memory and the custom hardware cores. This is the job of the buses within the system, to connect these vital components together. Figure 5.1 shows a block diagram view of the SMP version of hthreads.

The first bus that is used is the processor local bus (PLB) [28]. This bus (single PLB in the SMP hthreads system) connects directly to both PowerPC's, main memory, and the plb2opb [24] and opb2plb [23] bridges which, as the name

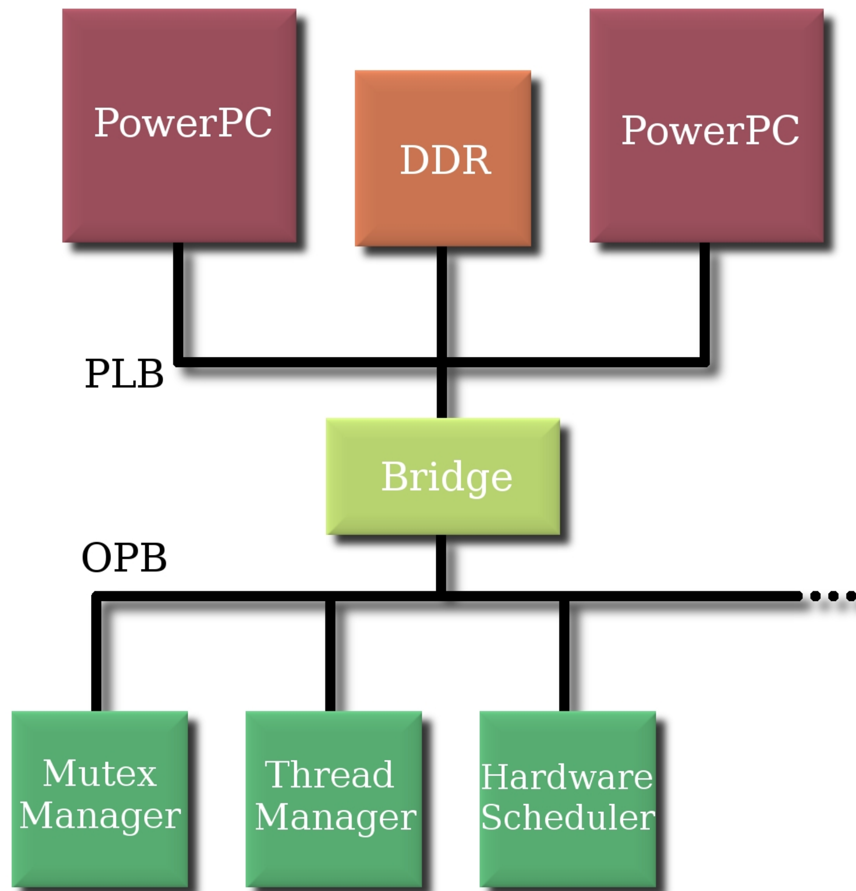

Figure 5.1. High-level overview of the SMP hthreads system

infers provides a 'bridge' between the PLB and OPB [22](On-Chip Peripheral Bus) buses in the design. The decision to use one PLB for both processors in the system was rooted in the design decision to allow fast access to main memory for both processors, which requires connecting main memory directly to the PLB. If the decision instead would have been to use two PLB's, one for each processor, then main memory would need to be connected to the OPB bus which would have introduced additional latencies when accessing main memory from the PowerPC's

because of the plb2opb bridge.

The OPB is a bus that is used to offload slower and less time-critical cores, keeping the PLB free for memory transactions. Deciding how to attach the hardware cores to the different bus architectures can be difficult to determine. Finding the most efficient balance is the ultimate goal. If too many cores are added to the PLB then the PLB becomes overloaded with transactions and the system as a whole suffers. However if the only devices attached to the PLB are both PowerPC's then this too can be detrimental to the entire system because every access to outside resources would require a transaction over the plb2opb bridge which adds additional latencies. In the SMP hthreads design, determining the balance was based on the fact that access times to memory are critical, therefore other than the PowerPC's, main memory is the only core (besides the shared BRAM) that is on the PLB.

Concerning memory, the SMP hthreads system contains three layers. The largest and correspondingly slowest memory is the DDR SDRAM memory, also called main memory or global memory. This memory is configurable in size because it plugs directly into the DIMM connector provided with the board. Again, this memory has been placed directly on the PLB in an attempt to reduce the latency when requesting read and write operations from main memory.

The second level of memory in the hthreads system is the on-chip BRAM's (Block RAM) which are synthesized into the FPGA. These memories can be application specific or they can be used as OCM's (On-Chip Memory) for the PowerPC's. When used as OCM's for the PowerPC's the BRAM's act as an L2 cache [20]. These access times are significantly faster than main memory, however because of the clock difference between the system clock and the PLB clock,

they are still slower than accessing the L1 caches. This type of memory is used in conjunction with many of the custom hardware cores, however because data coherency is an issue in the multiple processor design, this memory can not be used as OCM's for the PowerPC's. This memory architecture can also be used directly on the PLB to store user application programs. However because many applications are too large to fit entirely into the PLB BRAM, only small sections or more typically just the .boot0 section is stored in the PLB BRAM and is then used to jump to the main application in DDR SDRAM.

Local PowerPC caches are the final level of memory in the system. The PowerPC's each have two L1 caches, one for instructions and the other for data. Both L1 caches are 16kB in size [18]. This is also the most interesting level regarding the multiprocessor design because in the Virtex-II Pro FPGA's there is no cache coherency between processors, therefore to implement data caching in this setup would require a snoopy cache protocol which is a way to ensure that when data has been updated outside one of the local processor caches that a section of that cache becomes invalidated. It would also be required that if a value in the data cache was changed, then that value would be written out to main memory and that any other resource which held this value in its cache would invalidate this data as well. The implementation of this protocol has not taken place in the SMP hthreads system and in previous designs the work-around was simple - disable data caching. This is the necessary approach dictated by current limitations and is also the approach that will be taken for the SMP hthreads design.

Finally, the custom hardware cores in the hthreads system can be classified as either operating system specific or hardware threads. The operating system specific cores include the thread manager, scheduler and mutex manager. These

cores were designed to reduce the load of a typical operating system so that the processor can spend more time executing applications and less time managing them. For example, in the previous design the scheduler was designed to only interrupt the processor if a thread with a better priority was ready to run. This eliminated a substantial amount of jitter in the system compared to a software only implementation of the kernel. To take advantage of these performance improvements for multiple processors the thread manager and scheduler will both require modifications. This will be covered extensively in section 6.1.

5.2 Software Setup

Using the hardware support built into the system, the first problem to solve is how to initialize a multi-processor software system. Many variables become a factor when trying to consistently bring the system up into a known state. For instance, which processor reads the processor identification register first? Which processor performs system component initialization? What ensures that each processor during initialization is given a unique stack? Which processor should run the main thread, and which one should execute the idle thread (enter an infinite while loop and wait for an event to occur)? These are all issues that must be addressed to ensure proper operation.

Two source files are used to perform the majority of the initialization, crt0-smp.S and init.c. The first file is an assembly language file which reads the PIN from the thread manager and depending on the value returned will branch to create the appropriate stack for the current processor. It is important that each processor is given a unique stack, otherwise both processors would be using the same stack which would lead to the corruption of data and ultimately system fail-

ure. Once the stacks have been initialized, the global variables used for reentrancy are stored in user special purpose register zero (usprg0). This information is critical for the implementation of malloc in a thread-safe fashion and will be discussed further shortly. Once the reentrancy variables have been stored locally to the processors registers, the next step is to initialize the libc locking and unlocking code used for malloc. This completes the major portion of system initialization specified in crt0-smp.

Next, the software branches to the init.c file, and more specifically the hthread_init function call. At this point the processor running has a unique PIN and based on that identification performs differing tasks. The processor which was assigned PIN zero is used to perform most of the system initialization. This includes sending a reset to the soft hardware cores in the system, initializing the thread manager and setting the stack pointer for each thread in the system to NULL. Following the system initialization, this processor will then request a thread id from the thread manager and set this id as the idle thread for this processor. To complete the setup the processor will enable interrupts, enter user mode of operation and branch to the idle thread function. The processor which was assigned PIN one receives two thread id's from the thread manager. Thread id one is used as the main system thread and thread id two is used as the idle thread for this processor. Similar to processor zero, processor one finishes the initialization by enabling interrupts, entering user mode and running the main thread.

Assigning PIN's is an important task because it allows each processor in the system to be uniquely identified. The PIN is used throughout the software initialization and is important for ensuring correct functional output from the application. This task is challenging because in many ways it is similar to the 'chicken

or the egg' metaphor. For instance, when assigning PIN's to their corresponding processors what assurance is there that you have assigned the correct PIN to the corresponding physical processor on the die.

Due to the limitations of the PowerPC's embedded in the Virtex-II Pro it is extremely difficult to distinguish between the physical processors on the die of the chip. The embedded PowerPC's in the Virtex-4 FPGA's have a PVR (Processor Version Register) register dedicated for identifying processors, however this register is not available for the PowerPC's in the Virtex-II Pro's [26]. Due to these hardware limitations, a custom solution had to be developed. To achieve this, the thread manager was chosen to be responsible for distributing PIN's to the requesting processor. Since requests to all hardware cores are memory mapped, the address 0x60007C00 was designated as a request to receive a PIN from the thread manager. The value of the PIN is returned on the two least significant bits of the data lines on the OPB and it is stored in Special Purpose Register 7 (sprg7) within the PowerPC so that all future lookups will be local. Because the number of processors in the hybridthreads system has been limited to four, the thread manager simply increments a two bit register after a request has been received and processed. This will cause the register to overflow back to zero after the fourth processor has received the PIN value of three. This ensures that the highest PIN that can be granted from the thread manager is a value of three.

This accounts for PIN assignment, however what confidence is there that the correct physical processor will be mapped to the correct PIN, and is this even necessary? In a general sense, it is not. However the interrupt controller provided by Xilinx is not flexible enough to support arbitrary PIN assignment. The solution to this problem is to force sequential operation during system initialization. This

will force processor zero to execute first, ensuring that the next processor does not get an opportunity to request a PIN from the thread manager until the first processor completes initialization. Therefore the first physical processor on the die (the right PowerPC) [25] will execute first, securing a PIN value of zero. Next, the second physical processor on the die (the left PowerPC) [25] will secure a PIN value of one.

Setting up interrupts is the next problem to address in this multiple processor design and is tightly coupled to arbitrary PIN assignment and the limitations of the Xilinx interrupt controller. Initially, the interrupts were hard-coded into the .mhs file which is the hardware description file used in the Xilinx build process. However this limits the design by forcing only one interrupt to be sent to each processor. This means that if the sequence of acquiring the PIN from the thread manager is out of order, then the scheduler could be sending an interrupt to the wrong processor. In an attempt to solve this problem, the .mhs file was modified to send both interrupts to both Processor Interrupt Controller's (PIC) for each PowerPC, and then depending on the PIN for that particular processor one of the interrupts becomes masked, ensuring that the correct processor is servicing the interrupt. This solution seemed promising at first, however because the design uses two PIC's, one for each processor, then each processor has no way to tell which PIC it is assigned (this address is hard-coded). This could lead to situations where PowerPC zero is reading from or writing to the incorrect PIC. Clearly this is not functionally correct. In section 9 a possible solution to arbitrary PIN assignment and interrupt masking will be explored.

The next topic regarding initialization is how to initialize the stack for the software initialization portion of the code. Although only one main thread is

used, both processors execute very similar portions of the initialization code and therefore require a stack. The issue arises when they both use the same stack, and the second processor wipes out data that was setup during the initialization phase of the first processor. To solve this issue, the first statement executed in software is to acquire a PIN from the thread manager. This allows each processor to identify itself, and then branch to the correct section of the crt0-smp file for stack initialization. Based on this identifier, each processor then create a unique stack and therefore eliminates the possibility of data corruption through a shared stack.

Finally, in hthreads there exists the concept of an idle thread. This thread is issued a priority of 127, the worst priority assignment possible. This ensures that all other threads will preempt the idle thread if they are in the ready-to-run state. Functionally, the idle thread is simply an infinite while loop that executes when there is nothing else for the processor to run. Inside this infinite while loop is a call to hthread_sched. hthread_sched is a system call used to run the scheduler. This gives the idle thread the opportunity to find something better to execute. In the context of initialization, the first processor branches to the idle thread after completing initialization. It then 'waits' for the second processor to begin executing the main thread where new threads will be created and executed on the both processors.

Chapter 6

Design & Architecture

To achieve correct operation using multiple processors in the hthreads system, significant architectural hardware changes were required for the thread manager and hardware scheduler. Similarly, to mirror the changes that occurred in hardware, the software also needed to incorporate these updates to guarantee correct functional operation. This translates into very low-level changes on how operations are performed within the hthreads system. The scope of these changes ranged from modifying memory-mapped commands that are sent over the bus to changing the way I/O is serviced by each PowerPC in the system.

6.1 SMP Hardware Design

The initial step in the design was to become familiar with the original uniprocessor hthreads system to gain a better understanding of what issues needed to be addressed to expand the design to include multiple processors. This involved learning the directory structure, where files are stored and which files require modifications. This is not exclusive to hthreads, however what is unique to hthreads is

the originality of the design and the locations of files that required modifications. Also, learning the hthreads build process and how it interacts with the Xilinx toolsuite is critical to the overall system design. Although this sounds trivial compared to the actual design, it was a significant undertaking because the learning curve for the Xilinx toolsuite is substantial and requires much time and effort to become competent. Once an understanding of the basic toolsuite was acquired, a more in-depth look at the hardware cores in the hthreads system was the next step in the process. Specifically, understanding in great detail the scheduler and thread manager cores that are used in conjunction with one another to perform thread management and thread scheduling tasks for the hthreads system was an important step in the SMP hthreads system design. Each of these modules implements a large number of states in their respective state machines with dozens of registers and signals to control in each core.

After overcoming the initial learning curve associated with this design, the first modification was to the scheduler. To implement a multi-processor scheduler required adding additional current thread registers to the scheduler and thread manager cores. The purpose of the current thread register is to identify which threads are currently executing on which processor. Since the design now includes more than one processor the number of current thread registers were increased to equal the number of processors in the system. The current thread register tracks processor specific information, therefore the implementation of these registers are in the thread manager which directly interacts with all the processors in the system. This means that these registers are writable by the thread manager and readable by the scheduler.

Similar in concept to the current thread register is the next thread register

which contains the thread id of the highest priority thread on the ready-to-run queue that is scheduled to execute next. For this design the decision was made to continue to use only one next thread register. If the decision had been to equate the number of next thread registers to the number of processors in the system, then deterministic scheduling that was achieved in the previous scheduler design would have been much more difficult to implement. The reason for this is that instead of making one scheduling decision based on the highest priority thread in the system, the scheduler would now be responsible for N scheduling decisions where N is the number of processors capable of executing threads in the system. This would lead to additional overhead and would increase the amount of time required to make a scheduling decision. Although these are negative aspects of using multiple next thread registers, there are also several positive aspects that should not be ignored. First, with multiple next thread registers the scheduler has more control over which threads run on which processors. Currently, the scheduler makes a decision and can preempt the processor which is executing the lowest priority thread, however this does not guarantee that the thread will actually run on the intended processor due to the interrupt latency of the PowerPC. For example, if the scheduler interrupts the first processor and the second processor then requests the next thread available to run then it is possible that the second processor could 'beat' the first processor to the DEQUEUE request from the thread manager/scheduler. Fortunately this will only cause the first processor to respond eventually to the interrupt and check that the thread that is executing is the best thread to be running - this adds overhead into the system but is functionally correct. Also, multiple next thread registers would allow more flexibility in scheduling threads through additional SMP scheduling constructs such

as processor affinity. This would allow the user to force certain threads to only execute on the specified processor. However, because the SMP hthreads system does not use the data cache for the PowerPC's, the implementation of processor affinity would not be beneficial to the performance of user applications. This type of control would only be necessary in future versions of the hthreads system when cache coherency is no longer an issue.

An integral part of the new scheduler design was to develop a method for determining the next thread to execute from the pool of available threads on the ready-to-run queue. Several factors must be considered when making a scheduling decision. First, the scheduler needs to know the priorities of all currently running threads. This required creating a new register solely responsible for tracking this information (current_priority_reg). Secondly, the scheduler needs to ensure that if multiple processors could be preempted, that only one processor is interrupted and furthermore that the processor that is interrupted is the processor currently executing the lowest priority thread. For example, if the first processor is executing a thread of priority equal to ten and the second processor is currently running a thread of priority equal to seven, and the highest priority thread on the ready-to-run queue has a priority equal to five, then for proper operation the scheduler must ensure that the first processor is interrupted and not the second one (zero is the highest priority in the hthreads system). This is because the first processor has the lowest priority thread, and if the second processor is preempted instead then when the thread with a priority of seven is enqueued back onto the ready-to-run queue the scheduler would then immediately preempt the first processor and cause an unnecessary preemption. This would negatively impact system performance because of the unnecessary preemption which would waste

processor clock cycles. To ensure that this does not happen the check_preempt function was created to handle these scenarios. This function takes the highest priority thread on the ready-to-run queue and the priority register as parameters, and determines which processor if any should be preempted. The method used to make this determination is to compare priorities, and if the highest priority thread on the ready-to-run queue is better than the currently executing thread then the function tracks the difference between these priorities. In the end the highest difference corresponds to the processor that should be preempted. Figure 6.1 helps to illustrate this point.

If there does not exist a higher priority thread in the ready-to-run queue to execute then C_NUM_CPUS is returned which is an indication to the caller that preemption should not occur. This is because C_NUM_CPUS will always be one value higher than the highest PIN assignment in the system.

The scheduler redesign required significant changes to the original design. Table 6.1 highlights the states that were altered or added to the hardware scheduler for this design.

The first modification required making changes to the ENQUEUE operation. To account for multiple processors, the first addition is to read the processor id from the bus, encoded into the request itself. Due to bit mapping definitions from the previous design the mapping of the processor id bits into the request was limited to bits 14 and 15 of all memory mapped reads, regardless if the operation is requested from the hardware scheduler or thread manager. Extracting this information from the bus has become a common procedure in the SMP version of the design. The second modification to the ENQUEUE operation is to determine if the thread being added to the ready-to-run queue is an idle thread. This check

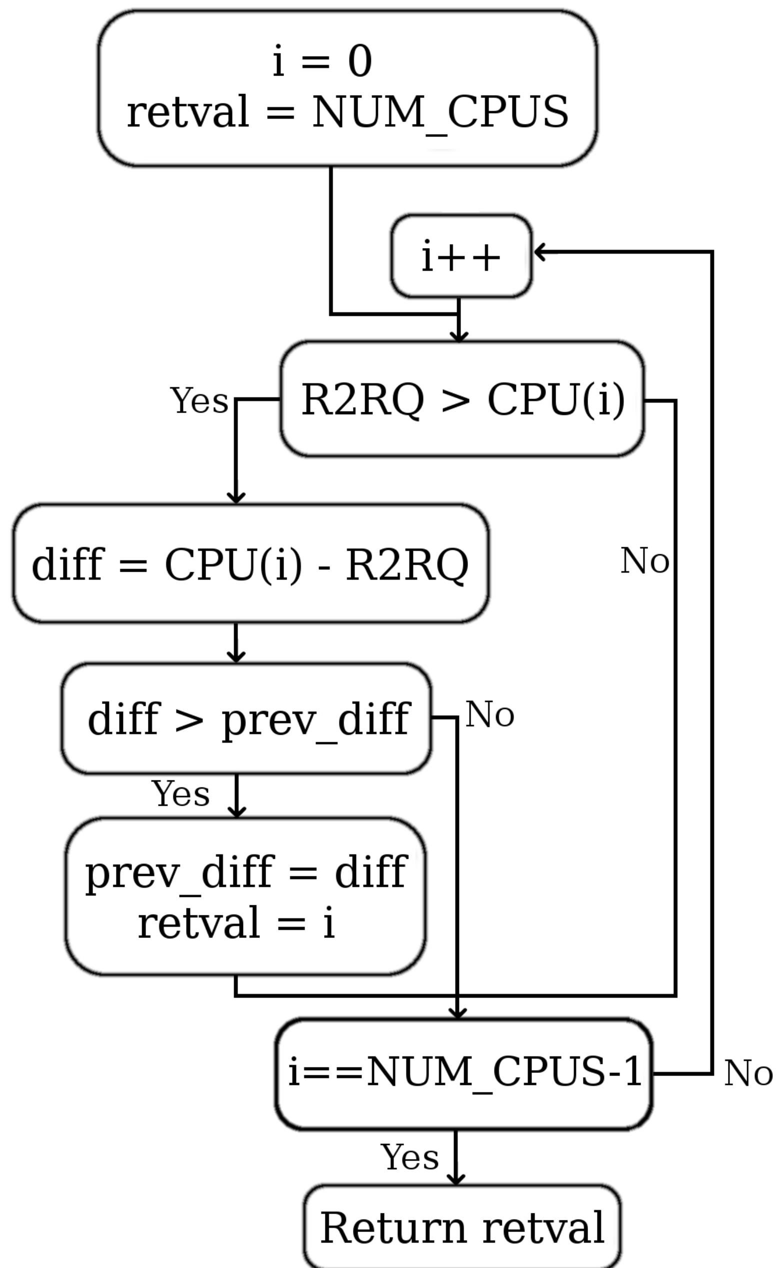

Figure 6.1. Illustration of the check_preempt function implemented in the hardware scheduler

Type	Name	Actions
TMcom	Enqueue	Adds a thread to the ready-to-run queue
TMcom	Dequeue	Removes a thread from the ready-to-run queue
TMcom	Idle_Request_ID	Requests the processor specific idle thread
BUScom	Set_Sched_Param	Alters a threads scheduling parameters
BUScom	Set_Preemption	Enable/disable preemption
BUScom	Get_Idle_Thread	Return the processor specific idle thread id
BUScom	Set_Idle_Thread	Set the processor specific idle thread id
BUScom	Syscall_Lock	Request (acquire or release) the system call lock
BUScom	Malloc_Lock	Request (acquire or release) the lock used by libc malloc (malloc_lock() and malloc_unlock())

Table 6.1. Command Set Additions and Modifications to the Hardware Scheduler Module

is done because idle threads cannot exist on the ready-to-run queue. The reason for this limitation is that if idle threads were allowed on the queue then it would be possible for an idle thread to be put into the next thread register which means that it could be dequeued by the wrong processor. If this check returns true, then the hardware simply returns without adding the thread to the queue. The final modification required to the ENQUEUE request was to determine which processor needs to be preempted when a thread with a higher priority becomes available to execute. This is handled by the check_preempt function as previously described. Again, the check_preempt function takes two parameters as its arguments. The first parameter is the priority of the highest priority thread on the ready-to-run queue. The second parameter passed to check_preempt is a std_logic_vector that tracks the priorities of the threads which are currently executing on all processors in the system. Using this information, the check_preempt function then compares the two parameters and can take two paths to completion. The first path determines that the highest priority thread on the ready-to-run queue should not

preempt any of the processors in the system. The second path to completion is that one or more processors could be preempted, and here the check_preempt function determines which processor out of all the available processors should actually be preempted by returning the PIN value to the requesting state.

The second modification to the hardware scheduler involved the DEQUEUE operation. Like the ENQUEUE operation, the first modification is to again read the processor id from the bus. This information is necessary because the scheduler needs to track the priorities of the threads that are currently executing on all the processors in the system. This is one of two places where this information gets updated, the other being inside the IDLE_ID_DEQ operation where an idle thread is chosen to execute on a particular processor because there are no threads in the ready-to-run state. The other change to the DEQUEUE operation occurs when the ready-to-run queue is empty. When this happens, the next_thread is invalidated. Previously the idle thread was set up as the next_thread to run, however due to the nature of SMP systems and how idle threads are handled, this was changed because again if an idle thread is placed into the next_thread register it is possible that the wrong processor could attempt to dequeue and execute the idle thread which was set up for the other processor.

SET_SCHED_PARAM begins as the previous two operations, by acquiring the processor id from the bus. The PIN is used to help determine whether or not this is a valid operation for the requested thread. This is an artifact of the previous design. The second major change to this set of states within the finite state machine of the scheduler is to skip the preemption check if the thread is not on the ready-to-run queue. This is important because a thread could have its priority elevated above a currently executing thread, however this thread that is

being updated might not be in the ready-to-run state. Therefore this would cause a processor to respond to an interrupt and attempt to execute a thread that is not on the ready-to-run queue. Conversely, if the thread being modified is on the ready-to-run queue then the check_preempt function is called to determine if a processor should be preempted in light of the updated scheduling parameters for this particular thread. This operation and the ENQUEUE operation are the only places in the hardware scheduler where preemption can occur.

Because idle threads are handled differently in the SMP hthreads system, there must now be a way to 'dequeue' idle threads when a better thread is not available to execute. To accomplish this the IDLE_REQUEST_ID state was added to the state machine. This is a request from the thread manager when either a YIELD or DEQUEUE system call has been requested. The thread manager checks to see if the ready-to-run queue is empty, and if so it instead requests the idle thread for the requesting PIN. This functionality allows each processor to have its own unique idle thread to execute.

SYSCALL_LOCK is the most critical addition to the hardware scheduler with regards to system stability. However, because the scope of this call encompasses all system calls from start to completion, it is also a hindrance to system performance because of the large restriction it places on the amount of concurrency in the system. This call uses bit 13 of the address to encode the type of operation into the request. If bit 13 is set high, then the request is to acquire the lock. If it is set low then the request is to release the lock. This lock is processor specific and works by locking the bus if the lock is not already taken. However, if the lock has already been acquired then it is not granted. The lock is also not released unless the requesting processor owns the lock. This prevents the competing processor

from releasing the lock held by the other processor. System concurrency issues will be discussed in greater detail shortly.

MALLOC_LOCK is also critical for system stability in SMP systems. This is because malloc is not thread-safe by default. The libc implementation of malloc calls two functions to lock around critical regions of software - malloc_lock and malloc_unlock. However, by default these calls do nothing unless the system designer implements these calls. This means that without modification it would be possible for malloc to allocate the same region of memory for multiple threads. It would also be possible for the libc function free to attempt to deallocate memory that is still in use. This scenario only worsens when caching is enabled (because of the lack of a snoopy cache implementation). Malloc also requires the locks to be recursive, therefore a completely new type of lock had to be added in the hardware scheduler, however most of the basic concepts from the SYSCALL_LOCK could be carried into the MALLOC_LOCK operation. Bit 13 is still used to encode the type of operation into the request. However this time instead of locking on the PIN, the lock is based on the thread id. This allows the malloc call to complete before another thread can call malloc. Also, a new signal was added to the state machine called lock_count which tracks the depth of the recursion. Only when this signal returns to zero can the lock be released, and of course only when the correct thread id is requesting the release. This ensures the correct functional operation of the recursive lock required by malloc.

For the remaining operations (SET_PREEMPTION, GET_IDLE_THREAD and SET_IDLE_THREAD) the only modification made was to retrieve the PIN from the bus to perform processor specific operations. For instance, when setting the idle thread it is now required to know which processor is requesting the

operation.

Similar to the hardware scheduler redesign, the thread manager also required modifications to the original design. Table 6.2 highlights the states that were altered or added to the thread manager for this design.

Type	**Name**	**Actions**
BUScom	Assign_CPU	Assigns a unique PIN to the requesting processor
BUScom	Next_Thread	Updates thread specific data/registers, sends DEQUEUE request to the scheduler

Table 6.2. Command Set Additions and Modifications to the Thread Manager Module

The first addition to the thread manager was to add the ASSIGN_CPU state which returns the PIN to the requesting processor. This is a memory mapped read from the thread manager and returns the PIN on the two least significant bits of the bus. The logic is simple for this state - a counter tracks which PIN should be returned on the bus. Each time this operation is requested the counter increments. The counter is a two-bit register which allows for a maximum of four processors in the system. When the counter returns PIN three it overflows to zero.

The only other state requiring modification was the NEXT_THREAD state which is used to dequeue the thread stored in the next_thread register by the scheduler. The modification to this state involves dequeueing the idle thread from the scheduler. This state initially checks to see if the next_thread is valid. Previously if the next_thread was invalid then the thread manager would wait for the scheduler and timeout if the request took too long. This is now no longer valid because if the next_thread is invalid it could be because the ready-to-run queue is empty. Previously the next_thread_valid would have been true with

the idle_thread being put into the next_thread register. However, because the design now contains multiple idle_threads this solution was not feasible. Now the NEXT_THREAD state will check to see if the idle_thread is setup for the requesting processor and if so will request a pseudo dequeue of the idle_thread for that processor. This ensures correct operation of the idle_threads with multiple processors.

One final note on the hardware design involves concurrency. From operating system theory, the three standard types of concurrency are thread, interrupt and physical concurrency [14]. In the original design of the hthreads system, the first two types were considered and measures were taken to ensure that concurrency could be maximized while keeping the design functionally correct, however physical concurrency was not of critical importance because more pressing tasks were being addressed at that point. Many safeguards that are typically used to protect against the SMP-type issues were not included in the uniprocessor design. To counter this, the SMP version of hthreads uses spinlocks to enforce correct operation of the system. This permits all three types of concurrency to exist in the SMP hthreads design.

System calls were one area that was identified as a critical region in the design. These regions of code modify critical shared data which are used to store information about threads in the system. If both processors have access to this region of code at the same time then race conditions can occur with the outcome of the data being read and/or written being incorrect. To protect this region of code a shared lock between both processors has been implemented (SYSCALL_LOCK). In the first design of the hthreads system, the user would acquire a lock by performing a system call, however system calls are the region of code that require protection.

This presents a substantial obstacle because now a completely new technique for locking system calls between processors will need to be designed into the system.

Adding this functionality into the system required modifying the hardware scheduler core to include this critical lock around system calls. This is a memory-mapped read operation from the kernel with the type of request being encoded into the address. Two types of requests exist around the call to lock system calls, a request to lock the critical region of code and a request to release the lock around the critical region of the code. The scheduler implements this functionality by creating two new registers, syscall_mutex and syscall_mutex_holder. The call to syscall_mutex is used to track whether or not the mutex is locked and the syscall_mutex_holder keeps track of which processor currently owns the lock. When a request is made to acquire the lock the scheduler checks to see if the mutex is currently in use, and if so it returns a zero on the bus and returns to the idle state. However, if the mutex is not in use then syscall_mutex is updated to a value of one and the value of the requesting processor is stored in the syscall_mutex_holder. This is important because it allows the scheduler to only release a lock if it is owned by the processor requesting the release. This safeguard is used to ensure that only the processor which owns the lock is allowed to release the lock. In this scenario, the scheduler will update both registers and return the value of one on the bus to show that the release operation was successful. Conversely, the scheduler will return a zero on the bus when a request to release the lock has been denied. Figure 6.2 illustrates the SYSCALL_LOCK scheduler implementation.

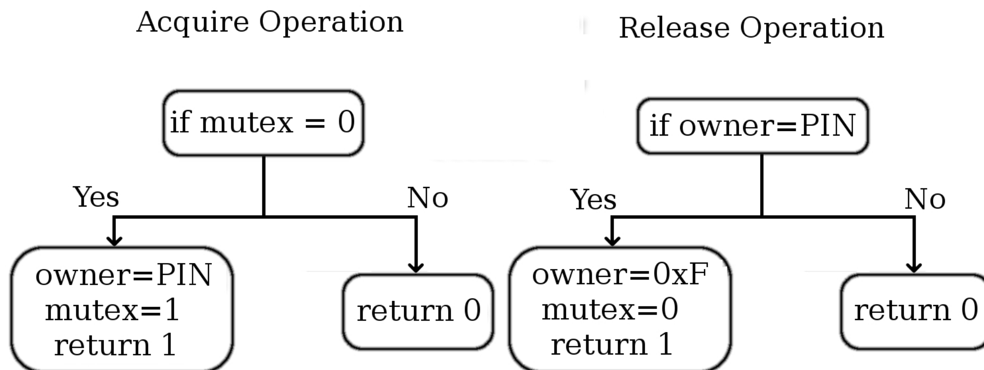

Figure 6.2. Illustration of the SYSCALL_LOCK implemented in the hardware scheduler

6.2 SMP Software Design

Transforming non-physically concurrent system-level software into SMP compatible code is challenging. Several problems exist either from hardware limitations or from troubleshooting concurrency issues. The following sections will describe the natural progression of the software as the project matured, along with the difficulties and successes that accompanied this progress.

Due to limitations of the hardware design provided by Xilinx, cache coherency is extremely difficult to achieve. Xilinx does not implement a snoopy cache protocol which severely limits the use of cache in SMP systems. Even worse, the implementation of a custom snoopy cache protocol is even more difficult because Xilinx does not extend the necessary hardware cache lines to perform this type of an operation. Initially, the SMP hthreads design attempted to incorporate data caching into the system. To work around this issue of incoherency the cached section of data was limited to the first 128 MB of main memory. Conversely, the non-cached section occupied the remaining memory. Through modifications to the linkerscript and special gcc directives, critical system variables that are

shared between processors were specified to be stored in the non-cached region of memory. This does not prevent race conditions, however it does keep the data that is shared cache coherent. The data that is being stored in this region of memory is an array of hthread_thread_t struct's. The definition of the hthread_thread_t struct is shown in Figure 6.3. This data is analogous to the Task Control Block (TCB) in linux. It is critical that this information is stored in the non-cached region of memory to eliminate erroneous errors that occur from reading the incorrect context of the thread or using the incorrect stack to access data.

```
typedef struct
{
    Hbool    newthread;
    void*    retval;
    void*    context;
    void*    stack;
    Huint    stacksize;
    Huint    hardware;
} hthread_thread_t;
```

Figure 6.3. hthread data structure

After many attempts to enable data caching for limited portions of the software, it became obvious that system stability was being compromised and that it was hindering the overall goal of porting hthreads to an SMP architecture. Due to this limitation, data caching was completely removed from the system. Also, because a snoopy cache implementation is extremely difficult due to the previously mentioned hardware issues, the data cache will stay disabled for the foreseeable future.

System stability is critical to the success of this project. To ensure stability in the developmental stages of this project significant portions of concurrency have been removed from the system. To accomplish this spin locks have been inserted

around system calls in the system-level software as shown in figure 6.4.

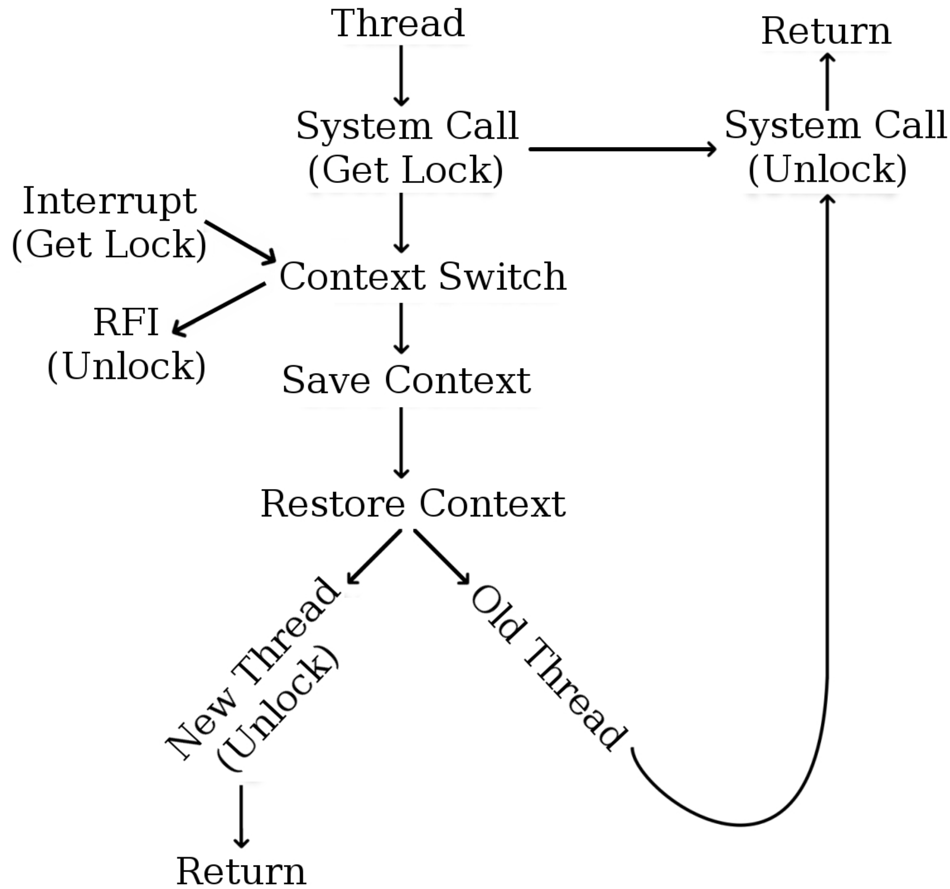

Figure 6.4. Entry and Exit paths for acquiring system call locks

This will ensure that only one processor has access to shared system data at a time. The initial call to acquire the lock is immediately before the system call and inside the non-critical exception handling code. Entry into kernel-space occurs at these two entry points exclusively. Unlock calls are placed at the end of the system call, immediately before running a newly created thread and when returning from a non-critical interrupt. Exiting from kernel-space occurs at these three exit points exclusively.

After examining the possible scenarios for acquiring and releasing the system

call lock in more detail it can be shown that in the hthreads system where a thread 'comes out' of a context switch depends on if the thread is a new thread or has already been executed and blocked. If the thread is new then it is 'bootstrapped' into execution and it is critical that the system call lock is released by this processor before the new thread begins execution. The other option for 'coming out' of a context switch is that the context of a previously run thread has been restored which will then release the system call lock when returning from the system call handler. Also, interrupts acquire and eventually will release the system call lock inside the non-critical exception handling code. It is also important to remember that this lock is PIN specific, therefore different threads can acquire and release the lock as long as they both execute on the same processor.

Although this lock is extremely valuable because it stabilizes the system, it is also a hindrance to concurrency because it locks portions of software that could safely execute concurrently. This can be detrimental to system performance and will require a significant amount of analysis to determine the best possible solution for balancing performance and concurrency. Future approaches will allow a finer level of control over concurrency and identify smaller portions of software that need protection while still maintaining the level of stability currently achieved.

Finally, several issues involving memory allocation and deallocation have been solved. First, in the SMP hthreads system multiple calls to malloc are necessary to allocate memory. This is not necessary in the single processor version. This could be related to the PLB, where the processor with the highest priority is constantly gaining access to the bus and preventing the other processor from successfully allocating the needed memory. Further analysis will be needed to verify this issue and possible solutions. However, the problem of not successfully

allocating memory is temporarily solved by continuously calling malloc until the pointer returned in not equal to NULL. This typically only takes one additional call to malloc and ensures that the memory is allocated properly.

Another malloc issue occurred when attempting to use data caching. When a thread exits and the parent thread has joined on it the system attempts to free the memory used by the exited thread by calling the libc free routine. The problem was that the free routine was attempting to deallocate memory that was still valid - the pointer to the region of memory to be freed was actually pointing to a valid region of memory for another thread. This was the result of cache incoherency and the fact that malloc is not thread-safe by default.

Making malloc thread-safe was briefly described in section 6.1 with a more detailed explanation to follow. In order to have a thread-safe implementation of malloc, it is imperative that the system design implements the stub function calls to __malloc_lock() and __malloc_unlock(). These are required by the libc implementation of malloc [15] to ensure that race conditions do not occur when attempting to allocate and deallocate memory. Furthermore, the locks which are used inside __malloc_lock() and __malloc_unlock() must be recursive locks [15] to ensure proper operation. The implementation of this lock is in the hardware scheduler state machine and is an atomic operation because the bus is locked until the scheduler returns the result of the request on the bus.

Chapter 7

Implementation Results

Synthesis of the SMP hardware scheduler module targeting a Xilinx [29] Virtex-II Pro 30 (speed grade -7) FPGA yields the following resource statistics: 1,553 out of 13,696 slices, 1104 out of 27,392 slice flip-flops, 2,860 out of 27,392 4-input LUTs, and 3 out of 136 BRAMs. The module has a maximum operating frequency of 150.991 MHz, which easily meets the hthreads goal of a 100 MHz system clock frequency. This version of the scheduler module retains the O(1) ready-to-run queue structure developed in the previous version of the design, however it incorporates functionality to support SMP scheduling of software threads while maintaining the backward compatibility to execute hardware threads concurrently.

Results of the SMP thread manager synthesis using the same setup as before reveals the following FPGA resource statistics: 700 out of 13,696 slices, 594 out of 27,392 slice flip-flops, 1,317 out of 27,392 4-input LUTs, and 1 out of 136 BRAMs. The module has a maximum operating frequency of 182.465 MHz, which again surpasses the goal of a 100 MHz system clock frequency. This version of the thread manager, like the hardware scheduler was developed to support SMP architectures.

Three software test cases were developed to stress the system and measure

the capabilities of the SMP hthreads system. These tests were designed to operate at both extremes of speedup. The first test performed numerous system calls in which little performance improvement would be expected due to the limited amount of concurrency available when performing a system call. The second test is a typical producer consumer problem in which the buffer size varies. For the single processor system if the buffer size is small then this would equate to nearly constant context switching between the producer and consumer threads. However for the SMP system no context switching is required, therefore giving a performance improvement which should be close to double the single processor system. The final application is a computationally intensive test that was developed to test the performance under more typical circumstances. This should reveal a more reasonable approximation for speedup that would be expected from the SMP hthreads system.

The first software test case is simpletest.c. This program loops through a series of hthread_create system calls followed immediately by hthread_join system calls. Remembering that all system calls are locked according to the processor that is executing the system call, it is not surprising that the performance improvement as shown in Figure 7.1 is almost zero. This is because the program is not computationally intensive, but instead spends the majority of its execution cycles performing system calls. This forces the SMP hthreads system into an almost uniprocessor system once again. The output from the program verifies this as well, showing a 'ping-pong' effect of execution between both PowerPC's. This makes sense as well. The first processor will sit idle, waiting for an event. The second processor then runs the main thread and executes an hthread_create system call. This generates an interrupt to the first processor which cannot get

the system call lock until the second processor exits the hthread_create system call. As soon as the second processor exits this system call, the first processor acquires the lock and switches context to the newly created thread - releasing the lock before bootstrapping to this new thread. This frees the lock for the second processor to run hthread_join. However, inside hthread_join there is nothing to join on because the first processor is still executing the newly created thread from the hthread_create system call, and it therefore releases the lock and starts running its own idle thread. Meanwhile, the first processor finishes running the new thread, exits and starts running the thread in the next_thread register which is the main thread (the thread with the highest priority on the ready-to-run queue). At this point the cycle has come full circle, and the process repeats with the roles of each processor reversed. This is what creates the 'ping-pong' effect and limits the performance of the system. There is little thread-level parallelism in this application.

Iterations	Single	Dual	Δ	Speedup
1	140715	235058	-94343	0.6
10	1324747	1300603	24144	1.02
100	13122225	12837658	284567	1.02
1000	131137366	128228828	2908538	1.02

Figure 7.1. simpletest.c results

The second test is the producer/consumer problem which has the opposite effect. This test is setup to show the best possible configuration and the corresponding results. A near 2x speedup executing this application in the SMP hthreads system is illustrated in figure 7.2. This stems from the fact that the buffer size is one, causing the uniprocessor system to context switch almost constantly. This buffer size however has no effect on the SMP version because each

thread can execute exclusively on one of the PowerPC's, therefore eliminating virtually all context switching. It is also interesting to note the performance results as the buffer size increases, which relieves the amount of context switching required for the uniprocessor system. The data illustrates that as the buffer size increases, the performance improvement of the SMP system plateaus at around 1.19x. This is because parallelism is being removed from the application and the effect of context switching is being masked out by the increasing size of the buffer.

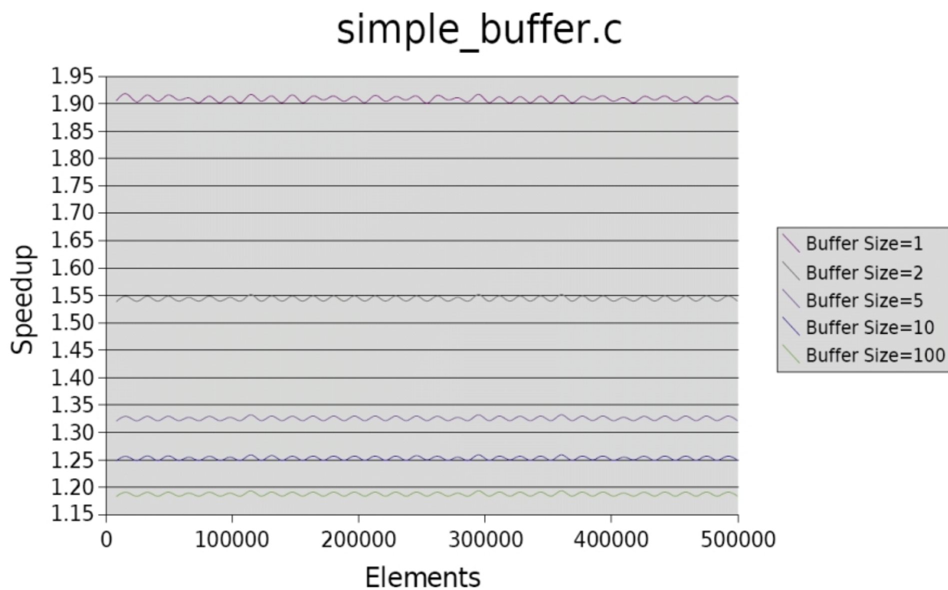

Figure 7.2. simple_buffer.c results

Similar to simple_buffer.c, dual_test.c is also computationally intensive. This application divides an array of numbers in half, allocating each half to a separate thread. Each thread will then perform a summation operation on the data, and the final value is the difference of the two summation results once both of the threads have completed. One important note about this application is that the data is not shared until the final difference operation. This frees up the overhead of communicating between processors and should provide a greater amount of

speedup compared to the producer/consumer test with a large buffer size. Figure 7.3 illustrates this accurately, showing a speedup of approximately 1.65x compared to the single processor implementation for large amounts of data.

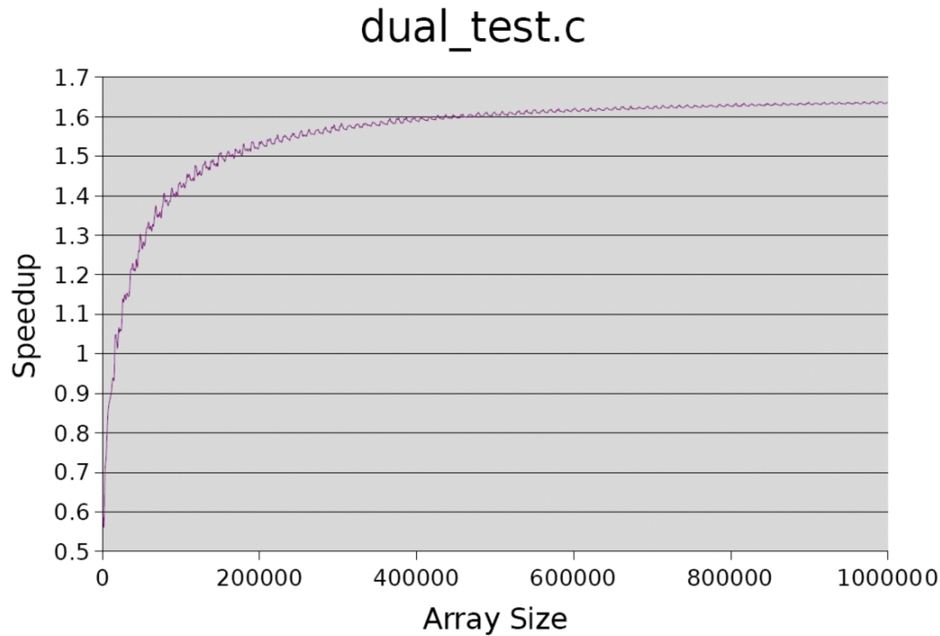

Figure 7.3. dual_test.c results

It is also interesting to note that the speedup does not plateau at 2x for this application. This is because of bus contention on the PLB between the two PowerPC's. Both processors are reading large amounts of data from main memory which results in bus arbitration. When bus arbitration occurs, this restricts main memory access to only one processor. This issue could be greatly improved by turning on the data cache for both processors which would relieve the amount of stress on the PLB.

Chapter 8

Conclusion

Xilinx has stated that attempting to design an SMP system with their dual PowerPC line of FPGA's is not feasible [19]. However, the work of the SMP hthreads project has shown otherwise. Not only is the project feasible, but performance improvements are readily available for computationally intensive applications. This is ground breaking work because it allows performance improvements from an existing resource that was previously unavailable to use. This resource, the second PowerPC, has been sitting idle and taking up valuable space on the FPGA. With the SMP hthreads design this previously unused resource now has a tremendous impact on the performance of computationally intensive applications.

To achieve this success, several modifications were required to the existing hthreads uniprocessor design. First, changes to the soft hardware cores were necessary to account for the additional processor. This meant changes to critical states within the state machines for the hardware scheduler and the thread manager. Issues such as acquiring processor identification numbers and how to execute idle threads were solved. Furthermore, a new function (check_preempt) was required to make scheduling decisions for multiple processors. This function

allows the thread(s) with the highest priorities in the system to be executing at all times. Second, major system-level software changes were required to control the concurrency in the system and ensure stability. This required providing locks around system calls. To implement these locks required a combination of system-level software and custom hardware logic. In additional to locking around system calls, malloc also requires locking to ensure thread safety. These locks are similar to system call locks, however they must be recursive and lock on the thread id rather than the processor id.

Finally, three software tests were developed to show the performance of the new SMP hthreads system. The first test, simpletest.c successively calls hthread_create and hthread_join. Because this application is not computationally intensive and instead spends most its execution time performing system calls, the performance improvement was negligible. The next test illustrated the opposite boundary condition. Using a producer/consumer model with a buffer size of one, simple_buffer.c showed a performance improvement of 1.95x. This is explained by the almost constant context switching of the uniprocessor system between producer and consumer threads, contrasted with the SMP design where each thread occupies a PowerPC for exclusive execution. The third example, dual_test.c revealed more of a typical performance improvement for computationally intensive applications with a speedup of 1.65x when operating on large amounts of data.

In summary, hthreads has been successfully ported to an SMP architecture. This design takes advantage of the otherwise idle second PowerPC by making changes to the hardware scheduler and thread manager. Furthermore, critical software changes were also made to account for concurrency issues.

Chapter 9

Future Work

SMP hthreads is an enormous project, and although it is now functionally correct and stable there are several improvements that if carried out would improve the system performance substantially. First, arbitrary PIN assignment would allow the system to initialize using either PowerPC. Currently, sequential operation is forced which ensures that the right PowerPC on the FPGA die always receives PIN zero and the left PowerPC always receives PIN one. This is important because of the limitations of using one PIC (opb_intc) for both PowerPC's. To enable arbitrary PIN assignment a hardware wrapper core would be inserted around the PIC with two registers, one for each PowerPC. These registers would mirror the enable register on the PIC, however these registers would be PIN specific. This would allow both PowerPC's to reference the same PIC, however the PIC would contain unique registers for each processor. This allows specific interrupts to be masked. The order of execution would require first that each processor acquires a PIN from the thread manager followed by a request to this new hardware wrapper core by each processor to mask interrupts that should only be serviced by the other processor. This functionality is not critical, however the implementation

would be relatively simple.

The next improvement needed concerns adding concurrency back into the system. Initially the focus was on stabilizing the system which involved removing concurrency from the design. However, now that the design is stable, performance could be improved by increasing the amount of concurrency. The major portion of this improvement would come from reducing the amount of code that is locked during a system call. Currently, the entire system is locked down when a system call is performed. This is inefficient because much of the code does not require a lock and therefore there exist many opportunities where both processors could be executing in parallel and therefore improving the performance of the system. This would have the greatest impact on applications that use system calls frequently. Successfully adding concurrency back into the system requires an in-depth analysis of the system software to determine where it is safe to remove locks and which shared variables are critical and require locks.

Another improvement that would have a substantial effect on improving system performance is cache coherency. This is not unique to the SMP hthreads system, however due to hardware limitations it may not be feasible to attempt to implement this functionality. Others have tried [5] [7] [8] [16] [17], however the result is either unsatisfactory or would not merge well with the existing hthreads design. Xilinx will be releasing the FX (PowerPC) version of the Virtex-5 near the end of 2007, so it is possible that some of the hardware limitations that currently exist would be fixed.

PLB arbitration is another issue that is affecting performance on the SMP hthreads design. The current implementation of the PLB arbiter uses a fixed priority scheme to allow access to the bus [28]. Even worse, if a tie exists among

multiple PLB masters then the tie breaker also uses a fixed priority scheme [28] to resolve the tie. This means that one PowerPC will always be granted access to the PLB over the other PowerPC. The worst case scenario for this type of arbitration is that the higher priority processor is constantly using the bus, virtually eliminating the second process from the system. Similar to the cache coherency issue, this also is a hardware limitation. Two possible solutions exist which could solve this issue. First, use of the PLB4 citeplb4 bus provides two types of arbitration: fixed and round-robin. Clearly using the round-robin arbitration would allow fair access to the PLB for both PowerPC's, however documentation on Xilinx's implementation of this newer bus architecture is obscure, if it exists at all. Second, Xilinx has recently introduced the Multi-Port Memory Controller 2 (MPMC2) [21] which allows multiple connections to external memory. Even more important however is the arbitration scheme that is available when using the MPMC2. From the documentation, the MPMC2 allows fixed, round-robin and custom arbitration schemes to be used. This would clear the way for fair access to other components in the design.

The most exciting future work involves porting the SMP hthreads design to use the Microblaze architecture. With a Microblaze system several processors could simultaneously execute user applications. This allows for even greater speedups than what was shown when using both PowerPC's. Furthermore, it allows more stringent testing of the current design - stressing the system with congested bus traffic and data transfers. Potentially, the design could detect the amount of free space left on the FPGA and expand to add these processing blocks into the system for additional processing power.

Finally, upon successful completion of the Microblaze port, work could be

done to implement a heterogeneous processor system which uses both Microblaze and PowerPC architectures. This would be the most difficult to implement and could take several different paths. First, the approach could be taken to use separate memory banks for each architecture and execute two user applications concurrently. The second approach would be to allow threads to migrate between different architectures. This would provide a significant increase in performance, however the implementation would be extremely difficult to achieve.

For more information about the SMP hthreads design or the hthreads system in general please visit the main web-site [9] and/or the developer's web-site [10].

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. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens. Run-Time Services for Hybrid CPU/FPGA Systems on Chip. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, Rio De Janeiro, Brazil, December 2006.

[2] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. In *Proceedings of the The Fourteenth Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, December 2006.

[3] J. Anderson and J. Calandrino. Parallel Real-Time Task Scheduling on Multicore Platforms. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, Rio De Janeiro, Brazil, December 2006.

[4] 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.

[5] H. Angepat, D. Sunwoo, and D. Chiou. Ramp-white: An fpga-based coherent shared memory parallel computer emulator. `http://domino.research.ibm.com/acas/w3www_acas.nsf/images/projects_0607/$FILE/derekChiou.pdf`.

[6] J. K. Bennett. Shared memory multiprocessing using the virtex-ii ppc. Website. http://ecadw.colorado.edu/ecen5633-f05/SharedMemoryMultiprocessingwiththeVirtexIIPro.pdf.

[7] D. Brooks and M. Martonosi. Implementing Application-Specific Cache-Coherence Protocols in Configurable Hardware. http://www.eecs.harvard.edu/~dbrooks/canpc99.pdf.

[8] A. Hung. Cache Coherency for Symmetric Multiprocessor Systems on Programmable Chips. http://etd.uwaterloo.ca/etd/a2hung2004.pdf.

[9] K. Hybridthreads. hybridthreads - main page. Website. http://www.ittc.ku.edu/hybridthreads/.

[10] K. Hybridthreads. hybridthreads - wiki page. Website. http://wiki.ittc.ku.edu/hybridthread/Main_Page.

[11] IBM. Take charge of processor affinity. Website. http://www-128.ibm.com/developerworks/linux/library/l-affinity.html?ca=dgr-lnxw09Affinity.

[12] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly Media, Inc., Sebastopol, CA, USA, 1998.

[13] RAMP. Research accelerator for multiple processors. Website. http://ramp.eecs.berkeley.edu.

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

[15] Sourceware. sourceware.org - __malloc_lock/__malloc_unlock. Website. http://sourceware.org/newlib/libc.html.

[16] T. Suh. Integration and Evaluation of Cache Coherence Protocols for Multiprocessor SoC's. http://arch.ece.gatech.edu/pub/tsuh.pdf.

[17] T. Suh, S.-L. L. Lu, and H.-H. S. Lee. an FPGA Approach to Quantifying Coherence Traffic Efficiency on Multiprocessor Systems. In *17th International Conference on Field Programmable Logic and Applications*, Amsterdam, Netherlands, August 2007.

[18] Wikipedia. Powerpc 405. Website. http://en.wikipedia.org/wiki/PowerPC_400#PowerPC_405.

[19] Xilinx. Designing multiprocessor systems in platform studio. Website. http://direct.xilinx.com/bvdocs/whitepapers/wp262.pdf.

[20] Xilinx. Fpga embedded processors - revealing true system performance. Website. http://www.xilinx.com/products/design_resources/proc_central/resource/ETP-367paper.pdf.

[21] Xilinx. Multi-port memory controller 2. Website. http://www.xilinx.com/esp/wired/optical/xlnx_net/mpmc.htm.

[22] Xilinx. On-chip peripheral bus v2.0 with opb arbiter (v1.10c). Website. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_v20.pdf.

[23] Xilinx. Opb to plb bridge. Website. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb2plb_bridge.pdf.

[24] Xilinx. Plb to opb bridge. Website. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/plb2opb_bridge.pdf.

[25] Xilinx. Powerpc 405 clock macro for -7(c) and -6(i) speed grade dual-processor devices. Website. http://www.xilinx.com/bvdocs/appnotes/xapp755.pdf.

[26] Xilinx. Powerpc 405 processor block reference guide. Website. http://www.xilinx.com/bvdocs/userguides/ug018.pdf.

[27] Xilinx. Powerpc processor reference guide. Website. http://www.xilinx.com/bvdocs/userguides/ppc_ref_guide.pdf.

[28] Xilinx. Processor local bus (plb) v3.4. Website. http://www.xilinx.com/ipcenter/catalog/logicore/docs/plb_v34.pdf.

[29] Xilinx. Programmable logic devices. Website. http://www.xilinx.com.

[30] Xilinx. Xilinx xapp996. Website. http://direct.xilinx.com/bvdocs/appnotes/xapp996.pdf.