Abstracting the Hardware / Software Boundary through a Standard System Support Layer and Architecture

Erik Konrad Anderson

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 Doctor of Philosophy

Thesis Committee:

Dr. David Andrews: Chairperson

Dr. Perry Alexander

Dr. Douglass Niehaus

Dr. Ron Sass

Dr. Yang Zhang

Date Defended

The Thesis Committee for Erik Konrad Anderson certifies
That this is the approved version of the following thesis:

Abstracting the Hardware / Software Boundary through a Standard System Support Layer and Architecture

Committee:

____
Chairperson

____

____

____

____

____
Date Approved

Abstract

Reconfigurable computing is often lauded as having the potential to bridge the performance gap between computational needs and computational resources. Although numerous successes exist, difficulties persist bridging the CPU/FPGA boundary due to relegating hardware and software systems as separate and inconsistent computational models. Alternatively, the work presented in this thesis proposes using parallel programming models to abstract the CPU/FPGA boundary. Computational tasks would exist either as traditional CPU bound threads or as custom hardware threads running within the FPGA.

Achieving an abstract parallel programming model that spans the hardware/ software boundary depends on: extending an existing software parallel programming model into hardware, abstracting communication between hardware and software tasks, and providing equivalent high-level language constructs for hardware tasks. This thesis demonstrates that a shared memory multi-threaded programming model with high-level language semantics may be extended to hardware, consequently abstracting the hardware/software boundary.

The research for this thesis was conducted in three phases, and used the KU-developed Hybridthread Computational Model as its base operating system and platform. The first phase extended the semantics of a running thread to a new, independent and standard hardware system support layer known as the Hardware Thread Interface (HWTI). In this phase hardware threads are given a standard interface providing communication between the hardware thread and system. The second phase of the research extended and augmented the initial design to support a *meaningful* abstraction. Mechanisms in this phase include globally distributed local memory, a standard function call stack including support for recursion, high level language semantics, and a remote procedure call model. The third phase evaluated the HWTI by comparing the semantic and implementation differences between hardware and software threads, by using an adapted POSIX conformance and stress test-suite, and by demonstrating that well-known algorithms may be translated and ran as hardware threads using the HWTI.

This thesis concludes that parallel programming models can abstract the hardware/software boundary. Creating a meaningful abstraction depends both on migrating the communication and synchronization policies to hardware, but also providing high level language semantics to each computational tasks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reconfigurable computing has come a long way since Gerald Estrin presented his Fixed-Plus-Variable structure computer in 1960 [32, 33]. His idea was simple: combine a fixed processor with hardware that could be configured (and reconfigured) to run complex operations with speed and efficiency. Estrin's proposal was generations ahead of then-existing technology. It was not until 1993 that reconfigurable computing became a mainstream research endeavor when Athana proposed PRISM-I, using FPGAs to extend the instruction set of general purpose CPUs [11]. Today an entire commercial industry lead by manufacturers Xilinx [107] and Altera [4] is growing up around Field Programmable Gate Arrays (FPGA). Modern FPGAs are designed as a large array of Configurable Logic Blocks (CLB). Each CLB may be programmed, post-fabrication, to a small circuit. By connecting CLBs together, larger and more complex circuits may be produced.

Interestingly, the problem reconfigurable computing was originally intended to address has not been resolved. Estrin proposed his fixed-plus-variable structure in response to a challenge from John Pasta. At the time Pasta was the chairman of the Mathematics and Computer Science Research Advisory Committee to the Atomic Energy Commission. He wanted to look beyond Von Neumann architectures to raise the computation power of computers. Estrin proposed his fixed-plus-variable structure as a way to take advantage of hardware parallelism where Von Neumann sequential processors could not.

Nearly 50 years later, researchers make the same argument to promote the promise of reconfigurable computing. A number of recent successes validate the argument. In 2006 Morris used reconfigurable computing to gain a 1.3x speedup for mapping conjugate gradients [68], Kindratenko achieved a 3x speedup after porting NAMD, a molecular dynamics application [60], and Zhou reported a 2x speedup for LU decomposition [112]. However impressive their gains may have been, they are not without a cost. For each application the workgroup must invest considerable time in design-space exploration, determining which portion of the code can be translated, and how best to translate it. The process that each group went through during design-space exploration is presented in their respective publications.

The design-space exploration problem is one of the many issues keeping reconfigurable computing away from the mainstream. Ideally software programmers *without any hardware-specific knowledge* would be able to design, implement, and debug a hardware/software system in a known high-level language. The decision as to which portions of the applications get mapped into hardware cores and which remain as software binaries would be made after verification. Engineers could then profile their application using on-the-board results and determine which software and hardware combinations perform the best.

To achieve post-verification hardware/software segmentation, a reliable, efficient, and complete high level language to hardware descriptive language (HLL to HDL) translator is needed. A number of research groups have made impressive strides towards this goal. Most of these groups are focusing either on C or a derivative of C as the targeted high-level language. Handel-C is perhaps the most widely known commercial hardware descriptive language [22], Garp [19] and ROCCC [48] are two well known translators focusing on efficiently unrolling loops, and Altera's C2H is perhaps the most complete C to HDL translator [65]. Although existing research in C to HDL translation is impressive, no group has thus far been able to translate unaltered C to HDL for reconfigurable computing. Research groups either leave out important semantics within the language, such as pointers or function calls, or they add constructs to specify hardware details, such as bus connections or I/O ports.

What makes C to HDL so difficult? C is an abstraction for a general purpose processor, while an HDL is a layout language for hardware circuits. The two are solutions to different problems. More to the point, when C is compiled it targets an existing architecture, namely Von Neumann's. A HDL has no predefined target; all details of the circuit must be specified. Existing C to HDL tools overcome this difference by mapping concepts in C to equivalent circuit details in HDL. For example, variables become registers, case statements become multiplexers, and operations become arithmetic circuits. However, these types of translations leaves out many of C's intrinsic capabilities. For example, how do you translate a function call into HDL, how do you pass a software created data structure to a hardware task, or perhaps most importantly, how do you maintain C's "write once run anywhere" abstraction?

Parallel computing programming models that hide the architectural details of the underlying hardware offer a solution. Skillicorn defined CPU-based parallel processing computing models in [82,83]. Updating his definitions for FPGA/CPU systems, parallel computing programming models are highlighted by two points. First the model must be *easy to program*. A programming model implementation hides how the operating system and computational model achieve parallelism. The locations of tasks are irrelevant, either on one or more CPUs or as dedicated hardware cores within the FPGA. Second the model must be *architecture-independent*. A program written using a parallel programming model should be cross-platform and cross-CPU/FPGA compatible without modifications to the code. All architectural details of the underlying hardware are abstracted from the user by the programming model interface implementation.

This thesis demonstrates that a programming model and high level language constructs can be used to abstract the existing hardware/software boundary that currently exists between CPU and FPGA components. Using the shared memory multi-threaded programming model, the concept of a thread was extended to hardware. An abstract standard system interface, known as the Hardware Thread Interface (HWTI), was written to provide operating system functionality, programming model methods for synchronization and communication, and high level language semantics.

The shared memory multi-threaded programming model was chosen because it is a well-known existing abstraction for specifying task level parallelism [17,80]. To define the model clearly, shared memory means each thread has equal access to a global address space. Communication is accomplished by reading and writing to shared data structures within global memory. The location of data structures are passed through pointers within the semantics of the programming model. Multi-threaded is a reference that execution is divided between separately executing independent threads. Threads may have equal or varying priorities, and may either be running in pseudo-concurrency by sharing a computational unit, or running in true concurrency by using multiple computational units. "Computational units" typically refers to CPUs, however in this research their meaning is extended to include application specific hardware cores located within the FPGA fabric, referred to as hardware threads.

This research demonstrates that the shared memory multi-threaded programming model can be extended to hardware; as a consequence, parallelism is automatically enabled at the task level. The resulting system may best be described as a MIMD machine. As a note, this research does not specifically address instruction level parallelism as many other FPGA researchers do. Equally important to note is that this research does not prevent instruction level parallelism within the hardware thread.

The research for this thesis was conducted within the Kansas University developed Hybridthreads Computational Model (hthreads) [9,10,56,57,73,74]. Hthreads was initially conceived as a real time embedded operating system that could abstract the hardware/software boundary. Programmers specify their design using a familiar "Pthreads like" programming model. Their design is then compiled to run on a hybrid CPU/FPGA core. Hthreads uses a subset of the Pthread's API and is intentionally similar to it, supporting thread creations, joins, mutexes, and condition variables.

A block diagram of the hthread runtime system is shown in Figure 1.1. Hthreads efficiently implements many operating system functions by moving them to independent hardware cores located within the FPGA fabric (note that the hardware implemented system cores are different from hardware threads). For example, the

Scheduler core can make a scheduling decision a priori of a context switch [1, 2]. With hthreads, the CPU is only interrupted when a higher priority thread enters the scheduler and should be context switched with the currently running thread. This prevents periodic timer interrupts of the CPU to determine if a context switch should occur, minimizing jitter due to scheduling decisions. Other important operating system hardware cores include the Thread Manager to manage the creating, running, and joining of threads, the Mutex Manager to manage which threads may lock and own a mutex, the Condition Variable Manager to manage signals between threads, and a priority interrupt controller, known as the CPU Bypass Interrupt Scheduler (CBIS).

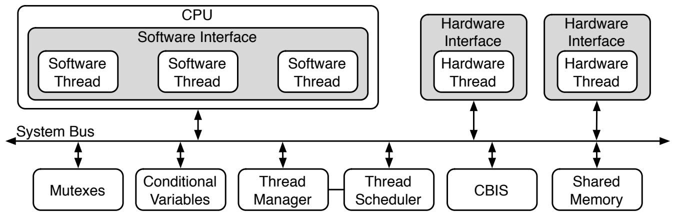


Figure 1.1. Hthread system block diagram

Moving operating system management to hardware cores provides efficient backend implementations of hthread policies. More importantly, because access to these cores is through memory mapped registers their functionality is available to both hardware and software interfaces. In fact, many of the hthread APIs that both the HWTI and software hthread libraries support are simply wrappers for encoded bus load and store operations. As a note, the development of the synchronization system cores, and the software libraries to call them, was completed prior to the research presented in this thesis.

A diagram of a hardware thread, showing the internal state machine and high-level communication between processes is in Figure 1.2. A hardware thread is composed of three entities: the vendor supplied IP bus interconnect (IPIF), the HWTI, and the user logic. The HWTI has two interfaces, system and user. The system interface is a set of memory mapped registers, whose protocol permits any hthread core, including CPUs, operating system cores, or other hardware threads, to interact with the hardware thread. On the other side, the user interface is a facade for synchronization, communication, and system services provided by the HWTI. The details of the interface protocols are in Appendix A.

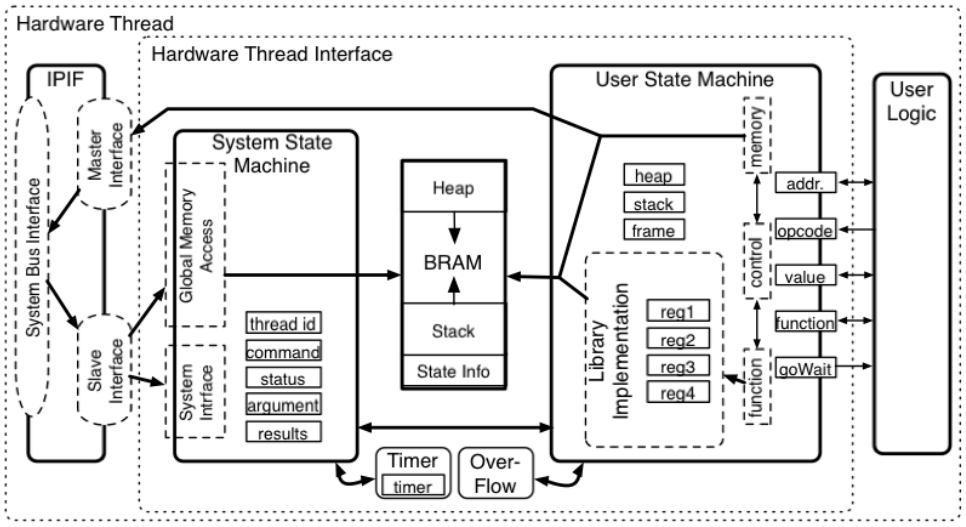


Figure 1.2. Hardware Thread Interface block diagram

Briefly, the system state machine manages the system interface and protocol and the user state machine manages the user interface and protocol. The user state machine also contains the functionality for the hthread API, the HWTI function call stack, and shared memory communication. Details of the state machine implementations are in Appendix B. The user logic requests services from the HWTI by driving the `opcode` register, along with appropriate values in the `address`, `function`, and `value` registers. The HWTI responds to requests only when the `goWait` register is a ‘1’ (a `GO`). A ‘0’ indicates the user logic should stall while the HWTI finishes a previous request. Table 1.1 lists the supported opcodes and their meaning. As an example, if the user logic wanted to read the value at memory location 0x32007840, the user logic would drive a `LOAD` to the `opcode` register and 0x32007840 to the `address` register. On the next clock cycle the HWTI would set the `goWait` register to `WAIT` and initiate the bus read. When the read completes the HWTI updates the `value` register with the value and returns `goWait` to `GO`.

Opcode	**Meaning**
`LOAD`	Read a value from memory
`STORE`	Write a value to memory
`POP`	Read a function parameter from the stack
`PUSH`	Push a function parameter to the stack
`CALL`	Call a function, system of user defined
`RETURN`	Return from a user defined function
`DECLARE`	Add space on the stack for local variables
`READ`	Read a variable from the stack
`WRITE`	Write a variable to the stack
`ADDRESSOF`	Retrieve a variable's global address

Table 1.1. User interface opcodes

Extending the hthread API to hardware, with only a few exceptions, is conceptually a straightforward process. Again this is because the synchronization services are implemented as independent hardware cores and accessed through encoded bus transactions. For example, neither the hthread software library or the HWTI has to "know" how to lock a mutex, they only have to know how to communicate with the Mutex Manager that will lock the mutex for them. Previous work on hardware threads presented in [6,56,57] demonstrate how the hthread API acts as a wrapper for the system services. The problem with this work is that the the HWTI's user interface is rigid, and does not inherently support expanding the hthread API or a more generalized shared memory programming model. As it turns out, extending the hthread API to hardware in a *meaningful* way is much more difficult. What was needed was a way to declare memory that is globally accessible and allow threads to call arbitrary functions with arbitrary parameter lists without making a distinction between user and system functions.

The enabling technology to provide a meaningful abstraction was "globally distributed local memory," or local memory for short. The local memory is instantiated into each HWTI using on-chip dual ported memory blocks. Local memory is accessible by both the user logic though existing load and store primitives and all other hardware cores through standard bus operations. Access time to the local memory for the user logic approaches "cache-like" speeds: 3 and 1 clock cycles for read and write operations respectively. Having a fast memory in turn enabled the HWTI to support a traditional function call stack. In essence, the HWTI manages a frame and stack pointer that point to its local memory compared to software stacks that point to traditional off-chip global memory. The HWTI supports push and pop operations to pass arbitrarily long parameter lists to the function, as well as the ability to declare space for user logic's local variables.

To complete the abstraction, programmers need a methodology to access the same set of libraries available to them while programming in a high-level language. Including support for C's stdio, string, or math libraries has largely been ignored by other FPGA developers. The HWTI uses one of two methods to add support for library functions. Specifically in the case of `malloc`, `calloc`, and `free`, the HWTI implements these functions in the same manner as other hthread system call functions, through its internal state machines. The memory the HWTI dynamically allocates is within its local memory; consequently access to it for the user logic is relatively fast. The second method of implementing libraries is to signal the CPU to perform the function. The HWTI signals a special CPU-bound Remote Procedural Call (RPC) system thread using existing support for mutexes and condition variables. Although relatively slow, this model provides hardware support for the majority of the existing library functions.

The context of a running thread in software is well known. Each software thread has a unique ID, a status (running, ready to run, exited, etc), its own stack, program counter, initial argument, a result, and so forth. A success of the HWTI is that it maps the context of a software resident thread into an equivalent context for a hardware resident thread. For example, a hardware thread's unique ID, status, initial argument, and result are encapsulated within memory mapped registers maintained by the HWTI. Each HWTI also manages a frame, stack, and heap pointer for the user logic (accessible through memory offset bus reads). The thread's starting address is encapsulated by the functionality of the thread, and the program counter is encapsulated by the current state of the user logic state machine. Encapsulating the same concepts of a running thread helps create and reinforce the abstraction between hardware and software.

Extending the concept of a thread to hardware is a relatively new concept. Previous work in this area [56, 96] has not explored the semantics of threads in hardware, especially the supporting API. Although meanings of many of the API calls are the same in hardware as they are in software, there are some important differences. These differences are due to hardware thread resources being statically allocated at synthesis time. For example, when a hardware thread blocks on a mutex, the FPGA resources go practically unused. In software if a thread has to block, the CPU context switches to a different thread. CPU time is not wasted but continues to perform useful work with a different thread. In the future though, context switching in hardware may have a meaning similar to software. If and when FPGAs can support partial runtime reconfiguration, conceptually the blocked hardware thread's resources may be swapped out for a thread that is not blocked, allowing the FPGA resources to continue to work.

To demonstrate that the HWTI provides an abstraction for communication and synchronization, all applicable test-cases from the POSIX test-suite [38] were adapted for hthreads, specifically the conformance and stress test-cases. A hardware and a software version was written for each test case. The stress cases were adopted to interact between hardware and software threads. The conformance tests demonstrated that both the hthread software API and the HWTI hardware API correctly produced the expected state after it was called. For example, one of the tests verified that after calling `hthread_mutex_lock`, the calling thread owns the mutex. Stress tests were designed to identify any long-term use errors, such as insufficient memory resources or insufficient thread IDs. As expected, the HWTI passed all conformance and stress tests. These tests were important since they demonstrate abstract communication and synchronization.

To demonstrate the HWTI's ability to support high level language semantics, a number of well known algorithms were translated to VHDL targeting the HWTI. The algorithms were quicksort, factorial, Huffman encoding, Haar discrete wavelet transform, and IDEA encryption. Collectively these algorithms demonstrated the HWTI correctly supports reading and writing to global memory using `LOAD` and `STORE` operations, a function call stack using `PUSH`, `POP`, and `CALL` operations, variable declaration using `DECLARE`, `READ`, and `WRITE` operations, and dynamic memory allocation using its light version of `malloc`. On the performance side, these results show that the HWTI's local memory provides hardware threads with cache-like performance. They also show that using local memory helps prevent bus contention, consequently improving overall system performance. Finally, the IDEA results were compared with the results from Vuletic's hardware accelerator threads and hand optimized VHDL code [96]. In short, these results show that Vuletic's threads achieved better performance by acting as a hardware accelerator, while hthread hardware threads achieved better performance through multi-tasking.

Both the test-cases adapted from the POSIX test-suite and the algorithms implemented in VHDL were hand written as user logic entities in VHDL. They were implemented without any knowledge of the underlying system, including: bus interconnect, memory hierarchy, hthread system cores, or location (hardware or software) of other threads. The only communication each user logic entity had was through the HWTI.

All experiments were conducted using either a Xilinx ML310 development board [103, 107] or a bus model simulation using ModelSim [47]. The ML310 contains a Virtex II Pro 30 [106] with 2 PowerPC 405 cores and roughly 30,000 equivalent logic gates. The only experiments conducted in simulation were the conformance and stress test cases for the test-suite. These were run in simulation due to lengthy synthesis time for the Virtex II Pro.

1.1 Contributions of this Thesis

- The hardware/software boundary may be abstracted using a parallel computing programming model.
- The context of a running thread may be extended to hardware through a

standard register set, function call stack, and a user logic state machine. The context of a hardware resident thread is *equivalent* to the context of a software resident thread but they are not *equal*.

- Abstract synchronization, within the multi-threaded model, is achieved in hardware by supporting key API calls for the programming model. System calls may be accessible to hardware threads by: redirecting the call to an independent system service hardware core, implementing the functionality within the thread, or using remote procedural calls to execute the system service on the CPU.

- Migrating system services to hardware cores give equal access to both software and hardware interfaces. The software and hardware implementation of these system calls becomes a wrapper for an encoded bus transaction to the system service hardware core.

- Abstract communication within the shared memory model is achieved in hardware by supporting load and store primitives.

- Hardware thread performance is improved while maintaining a shared memory model by providing abstract access to a dual ported on-chip memory. One port provides support for bus transactions; the second port provides support for the thread's memory requests. This memory is called globally distributed local memory.

- Function call stacks may be extended to hardware cores by leveraging globally distributed local memory. Hardware threads maintain their own function and stack pointer referencing their local memory.

- By supporting system calls, abstract communication, a standard function call mechanism, and local variables, the HWTI is a *meaningful* target for high-level language to hardware descriptive language translation.

- The primary difference between hardware resident threads and software resident threads is hardware threads persist physically within the FPGA fabric and are created at synthesis time, while software threads are non-persistent and are created virtually at runtime.

  - A subtle but important difference between creating a hardware thread and creating a software thread is that a hardware thread has dedicated resources that run in true concurrency, instead of the pseudo-concurrence prevalent with software threads. Hardware threads act like a software thread with a dedicated CPU running at the highest priority.

  - Context switches have no meaning in hardware since a hardware resident thread can not relinquish the CPU. To maintain a consistent behavior of a context switch, hardware threads block execution until the synchronization primitives permit the thread to resume.

  - FPGA resources within a hardware thread go unused while waiting to be created, blocking on a mutex, waiting for a condition variable signal, and after the thread has exited.

  - With partial runtime reconfiguration, the semantic meaning of a context switch for hardware resident threads may change to referring to reconfiguring the FPGA resources for a different user logic.

- The size of hardware supported system calls is proportional to the number of states the implementation uses. The performance of hardware supported system calls is proportional to the number of bus operations.

- To support the shared memory model, CPUs must run with data cache off. Any task level parrallelism improvements gained by using dedicated hardware threads must be significant enough to overcome the CPU's no-data cache penalty.
- Hardware threads have the same memory latency and bus contention problems as CPUs do. Long-term success of hardware threads will depend on finding inexpensive solutions to these problems.

Chapter 2

Background and Related Work

2.1 Field Programmable Gate Arrays

Field programmable gate array chips are growing in popularity in a diverse set of domains: software defined radio, image processing, encryption, networking, scientific applications, and even satellites. FPGAs are widely seen as the "middle-ground" between the relatively expensive but fast ASIC designs, and the inexpensive but sequential CPUs [26,41], this is especially true as the price to manufacturer FPGA chips has decreased. Although they have yet to be integrated into large numbers of commercial products FPGAs have proven their worth as prototypes for ASIC designs. Recently FPGAs have been integrated into supercomputers such as the Cray XD1 [29], as well as adapted to fit the HTX slot of the AMD Opteron processors [25]. It is not unreasonable to predict seeing FPGAs in desktop machines within 10 years.

There are three primary manufacturers of FPGAs: Xilinx [107], Altera [4], and Atmel [12], with Xilinx receiving the largest market share. Regardless of the manufacturer, the basic technology works the same. Each FPGA chip is composed of a two dimensional mesh of configurable logic blocks (CLB). Each CLB contains a small amount of memory that gets used as a look up table (LUT). The LUT determines what boolean logic circuit the CLB mimics. As a simplified example Figure 2.1 shows what a LUT would look like as a nand gate. Note that LUTs are a physical representation of a truth table. Using a interconnect network between CLBs, larger and more complicated circuits may be created. Within each CLB manufacturers also include hard resources to assist in common circuit designs such as multiplexers, shift registers, or adders. Mindful of the types of applications industry is targeting, manufacturers are also embedding memory, fixed point multipliers, and processors within the FPGA chip. As an example of the state of the art of FPGA technology, Xilinx has recently released its Virtex-5 family of chips [105]. The 330T model contains 51840 slices (2 slices per CLB, and each slice containing 4 6-input LUTs), 1458KB of embedded RAM, 192 embedded 25 * 18 multipliers, and off-chip connections for PCI Express, ethernet, and RocketIO. Virtex-5 family chips may run up to 550MHz and is fabricated using 65nm technology.

Input A	**Input B**	**Output**
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.1. Look up table example for a nand gate

2.2 The CPU/FPGA Boundary

In [99], the authors survey FPGAs use, specifically in high performance computing (HPC), and discuss the problems and differences of programming FPGA chips compared to software. They note that the flexibility of FPGAs is both a "blessing and a curse." Although FPGAs may be configured to application specific circuits, and once implemented they often have impressive speedups, it takes a specially trained hardware engineer to design the circuit. To program FPGAs engineers must design the circuit in a hardware descriptive language (HDL) such as VHDL or Verilog, and initially test the design in simulation. Where as it may only take a few minutes to compile a software application, it may take a few hours to synthesize a hardware application. Consequently the design-test cycle for hardware is potentially longer and more expensive. Finally, and perhaps most importantly, hardware engineers often have to deal with chip specific issues and constraints. In contrast software engineers are accustomed to programming in a platform independent manner. They sum up the problem best in the following paragraph.

> *As a prime example HPC software engineers expect, indeed demand, cross-platform standards. These abstract the scientific problem from the hardware, thus enabling the software engineer to design his or her solution independently of the hardware so enabling the software to be run in many different environments. Nowadays this is the case whether the software is serial or parallel, message passing or multithreaded: For a careful and experienced HPC software engineer code once, run anywhere is now a truly achievable goal. However for hardware this is, as yet, a very far distant goal.*

The hardware/software abstraction problem is restated in a number of other publications. In 1997, Mangione-Smith [85] states "The community must develop a set of application-programming interfaces (APIs) to support common interactions with any programming hardware." In 2003 Wolf [102] states "Research must also look at creating interfaces for both the FPGA fabric and CPU sides of the system." In 2004 Andrews [8] states "A mature high-level programming model would abstract the CPU/FPGA components, bus structure, memory, and low-level peripheral protocols into a transparent system platform." And in 2005 Jerraya [55] states "This heterogeneity (between hardware and software) complicates designing the interface and makes it time-consuming because it requires knowledge of both hardware and software and their interaction. Consequently, HW/SW interface codesign remains a largely unexplored noman's-land." It is clear that researchers believe abstracting the CPU/FPGA boundary remains a important problem to solve.

2.3 Computational Models

Computational models provide a slightly abstracted description of the primitive capabilities of a machine [3]. The basic attributes of a computational model were described by Brown [16] as a machine's primitive units, control and data mechanisms, communication capabilities, and synchronization mechanisms. Flynn created a taxonomy of computational models [50] differentiated by their instruction and data streams. FPGAs are unique in that they may be configured, post-fabrication, to conform to any desired computational model. By far, the most commonly accepted computational models for FPGAs within the reconfigurable computing community are based on the Single Instruction Stream Multiple Data Stream (SIMD), and Single Instruction Stream Single Data Stream (SISD) organizations. Under the SIMD model the FPGA operates on multiple data streams, typically extracted by unrolling loops, but sequenced by an instruction stream executing on the CPU. The SISD model for FPGAs has evolved from the lineage of complex instruction set computers (CISC), with the gates replacing traditional microcode. In the SISD model, sequences of general purpose instructions can be replaced with customized circuits executed as co-processor type instructions.

Both SIMD and SISD share the common attribute of a single instruction stream executing on the CPU that acts as a controller for an instruction-level *hardware-accelerator* embedded within the fabric of an FPGA. Figure 2.1 outlines this basic organization in which the FPGA can be viewed as a co-processor (slave) that implements a specific portion of an application. During application execution the CPU (master) must pipe input data into the co-processor module, and then it must poll the hardware accelerator until it is finished executing. This means the CPU is forced to remain idle while the FPGA-based co-processor is active, and only a single instruction stream can be active within the FPGA at any given point in time. To support the co-processor model, explicit interfaces are required for passing data between the distinct CPU and FPGA components. This should be concerning for the reconfigurable computing community, as lessons learned from historical parallel processing efforts clearly indicate the need to provide portable parallel programming models composed of unaltered high-level languages and middle-ware libraries [83].

On small scales this hardware acceleration model can be effective in exploiting instruction-level parallelism (ILP). On larger scales, the hardware-acceleration model has been used to replace ultra-critical sections of code within applications. This method, regardless of scale, is certainly not without its merits. In 2006 Morris used FPGAs to gain a 1.3x speedup for mapping conjugate gradients [68], Kindratenko achieved a 3x speedup after porting NAMD [60], a molecular dynamics

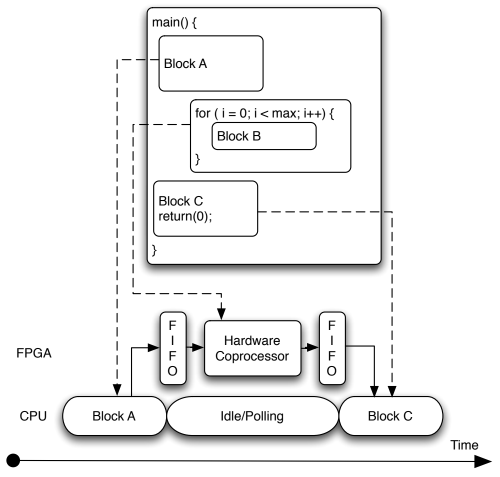


Figure 2.1. Traditional FPGA Co-Processor Model

application, to a SRC-6 CPU/FPGA supercomputer [53], and Zhou reported a 2x speedup for LU decomposition [112]. The problem with the hardware-acceleration model is that there is only a finite amount of parallelism to exploit within a single flow of execution [64,100]. These efforts effectively ignore the benefits of exploiting coarse grain parallelism historically seen in MIMD models.

In [50], Hennessy and Patterson describe the historical use of SIMD architectures starting from Unger [94] in 1958 and Slotnick [84] in 1962. They note that despite numerous attempts at achieving commercial success with an SIMD architecture, such as Thinking Machines and MasPar, they were all ultimately unsuccessful. Their reliance on data parallelism limited the applicability to a small set of problems (mostly signal and image processing applications). Furthermore, they failed to take advantage of more flexible and cost effective multiple microprocessor model. They conclude that with few exceptions all parallel computing machines today fit the MIMD model.

2.4 Programming Models for Parallel Computing

Although the distinction is somewhat ambiguous, a programming model is an a computation model abstraction. A parallel programming model allows a developer to specify multiple independent threads of control, the communication between threads, and synchronize thread execution. A parallel programming model abstracts the *mechanisms* of parallelism by providing *policies* of parallelism encapsulated within an API.

Skillicorn describes the requirements for a parallel computing programming model in [82,83], and are highlighted by two points. First the model must be *easy to program.* The software engineer should not have to know how the parallelism is achieved at the OS or hardware layer. To achieve this the policy of the parallelism must hide data communication, synchronization between tasks, and location of tasks (tasks located on which processing elements). Second the model must be *architecture-independent.* Programs written using a parallel programming model should be executable on all machines supporting the programming model without modifications.

Updating the parallel computing programming model definition to hybrid hardware/software systems requires extending the CPU computational model to FPGA tasks and abstracting the difference between CPU and FPGA resident tasks. Easy to program additionally means enabling the programmer to ignore the *location* of the task not only between CPUs but more importantly between hardware and software. This in turn means that the same communication and synchronization primitives available to software tasks must be extended to hardware tasks. Architecture-independent now implies compiling or translating tasks from software to hardware without the need to modify the source code prior to translation. This also implies HDL source code describing the functionality must be portable, without modifications, between different FPGA solutions.

There are two well known programming models for parallel computing. First is shared memory multi-threaded. POSIX [17] and Microsoft's .net framework [27] implement this model. Second is message passing, implemented in MPI [36]. The research presented in this thesis uses the POSIX Threads, or Pthreads, programming model implementation as its base.

2.5 C to Hardware Languages

A number of research groups have attempted to gain access to the FPGA through compiling a C [59] or C-like language into a hardware descriptive language. Creating hardware cores from C is compelling. There are far more skilled software engineers who can develop in C than there are hardware engineers that can develop in VHDL or Verilog. There is also a significant body of pre-written and pre-tested C code that could be leveraged within hardware. Although variations exist between C to HDL tools the general process is the same. The compiler creates and analyzes control and data flow graphs derived from the source code. The objective is to identify opportunities to parallelize operations. Most opportunities come from loop unrolling. Once identified the compiler derives, or relies on user pragmas to derive, hardware/software segmentation. The hardware identified code is translated to HDL, and a custom interface is created allowing data to be passed to and from the core. In most cases, the C to HDL compilers rely on vendor supplied tool chains to synthesize the HDL to netlists that may be downloaded onto FPGAs.

Unfortunately the translation of C (or any other high-level language) to a hardware descriptive language is not an easy task. Survey papers that cover this topic include Edwards [31] and De Micheli [66]. Unlike the pure software world where there is a known target, either a processor or a virtual machine, FPGAs provide only an "open slate." There is no notion of instructions, memory, stacks, heaps, or any other semantics common to high level languages. These semantic mechanisms must either be created by the compiler or inserted by the programmer. Furthermore there is the challenge of converting a language meant for a SISD machine to either a SIMD or MIMD machine. There remains a debate on how best to achieve this goal. The three schools of thought are to identify and use instruction level parallelism to create a SIMD machine (the most common approach), map sequential instructions into a systolic arrays to create a heavily pipelined SISD machine, or enable task level parallelism to create a MIMD machine. With the exception of [21], a yet unexplored fourth option is to combine task *and* instruction level parallelisms. Given these difficulties most C to hardware languages either require the user to insert hardware specific pragmas to take advantage of the underlying FPGA's computational model, or they leave out important capabilities within the language such as support for pointers or recursion. As impressive as some of these efforts have been, these limitations fail to completely abstract the hardware software boundary.

The earliest known C to hardware approach came from Cones [92] in 1988, and supported only a strict subset of C. Cones was also the first to capitalize on loop unrolling to create parallelism. Later work, including Garp [19, 20] and ROCCC [18, 48], depended more heavily on loop unrolling to achieve speedups. However, they discovered that data bandwidth is a major limiting factor regardless of the efficiency of their core. Streams C [37, 44, 45] with its predecessor NAPA

C [43] are perhaps the most well known examples of generating systolic arrays types of cores. Impulse C [52] and Handel C [22, 24] are commercial products that use a C like language as a driver for hardware synthesis. Altera's C2H (C to Hardware) [65] is unique in that it does not add pragmas to C and supports nearly the entire C language, recursion and floating point operations being the exception. Also noteworthy are PipeRench [34], Transmogrifier C [39], the SUIF compiler [90], DEFACTO [13, 86], C2Verilog [87, 88], SPARK [95], CASH [67], and SA-C [70]. Lastly, there is JHDL [15] and System C [93], these languages use objects in Java and C++ respectively to represent low level hardware details.

2.6 Abstracting the Hardware/Software Interface

High-level language to hardware descriptive language compilers attempt to abstract the hardware/software boundary by allowing the programmer to design his or her hardware core in a known programming language. The core of the problem though is more abstract, that is, how to enable and hide communication and synchronization between hardware and software portions of the application [102, 110]. Ideally, communication and synchronization is achieved through a standard policy that may be applied to a large set of problems. The research of a number of groups investigating this problem follows.

The Processor Architecture Laboratory from Ecole Polytechnique Federale de Lausanne proposes using an abstract virtualization layer between a software wrapper thread and the user logic "hardware accelerator" implemented in the FPGA [96–98]. Lead by Milian Vuletic, this idea has many desirable benefits. It satisfies the need for a standard interface into the FPGA and the need for platform independence. Furthermore, since it utilizes the thread model, multiple hardware accelerators may be running simultaneously, in parallel to the CPU, enabling thread level parallelism. This is advantageous since this would support the MIMD model instead of the SIMD model most C to hardware compilers use. However there are two major drawbacks to the virtualization layer approach. First, the hardware accelerator requires a separately written software wrapper thread for control. Second, their virtual memory manager requires interrupting the CPU whenever the hardware accelerator requires additional data from memory. This is largely a consequence of their adoption of Linux as their operating system, which requires hardware threads to access data through virtual addressing. A hardware thread must maintain a local version of a translation lookaside buffer (TLB), which is updated and kept consistent through the CPU's memory management unit (MMU). When a request for a virtual address not in the TLB is issued, an interrupt is generated to a custom patch within the Linux memory management functionality. Thus, the hardware thread remains a slave to the CPU, causing additional overhead and jitter for the applications running on the CPU, and must idle while the memory management code is ran on the CPU.

The System-Level Synthesis Group from Techniques of Informatics and Microelectronics for Computer Architecture has proposed a roadmap for developing seamless interfaces for hardware/software co-design [14, 54, 55]. They note that traditional hardware/software design is done by creating custom register sets for hardware that are interfaced by the CPU through custom software drivers for the programmer. This reliance on custom partitioning and design must be broken to solve the hardware software disconnect. They argue for designing system on a chip (SOC) applications using multiple layers of abstraction. Their abstractions layers, from lowest level to highest level, are "Explicit interfaces," "Data transfer," "Synchronization," "Communication," and "Partitioning." These abstraction layers are designed to give the engineers an easy methodology to build a system from the low level architecture details up to the application's algorithm. If done properly, they argue that SOC designs will take less time to develop and verify, be higher quality, enable post-development decisions on the chip's architecture, and ultimately be a better return on investment.

From Washington University, John Lockwood, who has specifically been researching FPGA solutions for Network Intrusion Detection Systems (NIDS), has proposed creating an abstraction to better encapsulate the parallelisms and stateful analysis aspects of network analysis algorithms [28, 72]. He notes that while Moore's Law is breaking down the need for network analysis continues to grow. He predicts that future performance advantages will either come from carefully crafted applications composed of multiple cores, or more likely application specific hardware cores. Lockwood goes on to state that handcrafting optimal hardware designs, both because of the differences in architectures as well as differences in various network analysis algorithms, is a "losing game." His solution is first to generate a high level language that can encapsulate the algorithms, second to generate a "transactor"abstraction from the language which encompasses a parallel hardware core, and finally to compile the transactors to a hardware specific implementation.

Lastly, and perhaps most importantly, is the original hybridthreads work investigated by Razali Jidin [56, 57]. His work created a prototype of the work presented in this thesis. Jidin extended the concept of a thread to hardware, developed mechanisms to support mutexes and condition variables, and supported communication between hardware and software threads. Still in use today, his work also demonstrated how to migrate operating system functionality into hardware, allowing both software and hardware threads equal access to them. The downside of his work was the separate treatment given to hardware and software threads. At the time there was not a unifying mechanism to treat threads, regardless of their location, equally. Lastly, his hardware thread interface still needed customization based on the thread's functionality.

Chapter 3

Extending the Shared Memory Multi-Threaded Programming Model to Hardware Resident Threads

In traditional Pthreads programming, a program consists of n threads running on m CPUs. A programmer may conceptualize each of his or her threads running on its own CPU, that is $n = m$, as depicted in Figure 3.1. Each thread has a sequential set of instructions to execute, accesses data from global memory, and synchronizes with the other threads using functionality provided in a middleware layer. The middleware layer is of course the Pthreads library that each thread links against.

An abstraction is created because the Pthreads library hides the fact that $n > m$ (except in rare cases), that is there are more threads than processors as

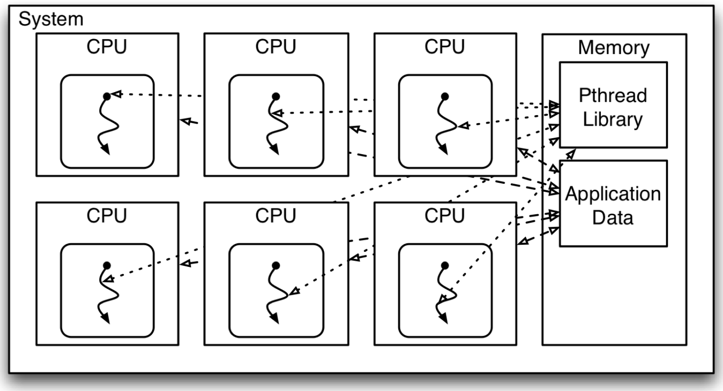


Figure 3.1. Conceptual Pthreads programming model

seen in Figure 3.2. In this case, threads will run in pseudo-concurrency, sharing time on the limited number of CPUs. A programmer does not care which CPU a thread runs on, and to some extent does not care when or how a thread runs. A programmer is only concerned that the overall sequence of operation runs according to the synchronization routines he or she inserted into the application.

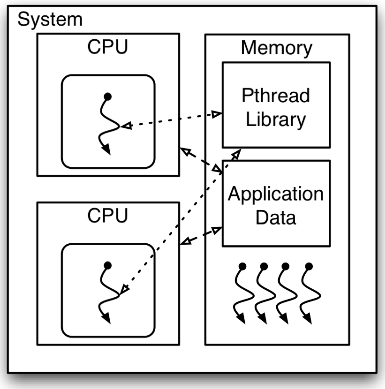


Figure 3.2. Implementation of the Pthreads programming model

With FPGAs, the algorithm performed by a thread may be conceptually implemented as a hardware circuit. This is advantageous since it provides threads with a "dedicated CPU" that is not time shared with other threads and will run in true concurrency. Further advantages may be found if hardware resident threads perform faster, with less power, or less jitter (for real time applications). Although we have hardware engineers who can write a thread's functionality as a hardware core, depicted in Figure 3.3, there remains three key challenges to creating the same synchronization and communication abstraction that Pthreads have.

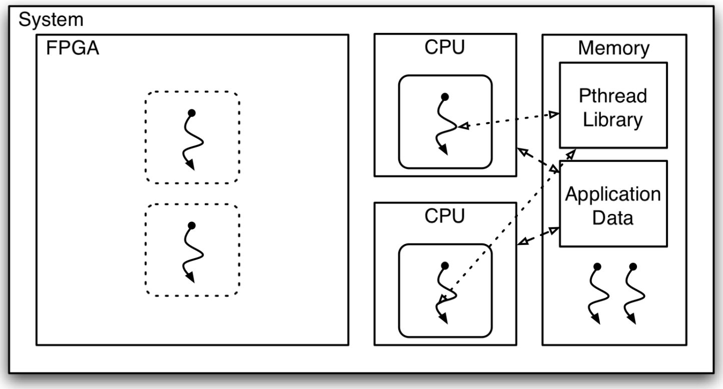


Figure 3.3. Migrating threads to hardware faces key abstraction challenges

- **Access to Synchronization Libraries** Hardware resident threads must be given *equal* access to the synchronization libraries that software threads have. This is a problem since the libraries are CPU instructions and not easily executable by custom, non-CPU, circuits.

- **Access to Shared Memory** Like the synchronization primitives, hardware threads must be given *equal* access to shared memory. With Pthreads, communication between threads is achieved by passing pointers to abstract data structures located in a global memory address space. The challenge for hardware threads is to receive and pass pointers within the semantics of the programming model, and have equal access to the data structure in memory.

- **Standard Register Set** No standard register sets exist within the FPGA fabric to encapsulate API operations. This includes register sets to abstract synchronization and communication operations from the hardware thread, and register sets to abstract creating, synchronizing, and joining on a hardware thread.

This research, in conjunction with the Hybridthreads project, solves these problems. These problems are solved by: creating a standard register set and interface protocol representing the semantics of the shared memory programming model, migrating key system synchronization services to hardware providing equal access to both hardware and software threads, and providing direct memory access for hardware threads. These services are encapsulated within a standard system service layer known as the Hardware Thread Interface (HWTI). The new system is depicted in Figure 3.4.

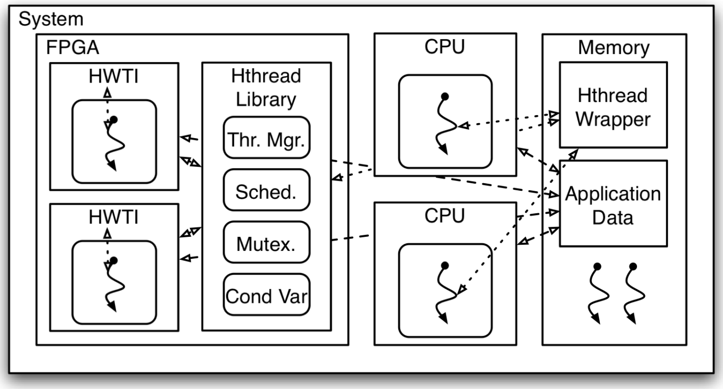


Figure 3.4. Hthreads system providing context for hardware threads and equal access for communication and synchronization

3.1 HWTI Standard Register Set

Migrating thread functionality to hardware is a solved problem; what may be represented in software may be implemented in hardware. However, there remains key questions as to the interface into and out of hardware threads. The HWTI meets this challenge by providing a standard register set to and from the hardware thread and system. Each hardware thread has its own copy of the HWTI. The HWTI remains constant regardless of the threads functionality. The register set is depicted in Figure 3.5, and consists of two interfaces, the "user interface" and the "system interface". The HWTI is a layer between the system's communication mechanism (shown in Figure 3.5 as a bus) and the hardware thread "user logic" entity. The user logic represents a thread's functionality and is written by the programmer.

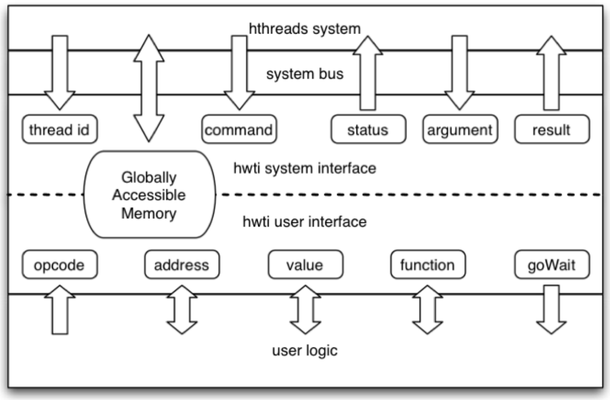


Figure 3.5. HWTI standard register set

The system interface captures the context implied within the system calls `hthread_create` and `hthread_join` as well as the hthread synchronization policies. This is important since it enables seamless creation and synchronization of threads regardless of their location. It is also important since this means the system interface is specific to the multi-thread shared memory programming model. If a different programming model was used the system interface may look very different.

The user interface acts as a facade [40], hiding system and implementation details from the user logic entity. The user logic requests services through the user interface that the HWTI performs on its behalf. The user logic is never concerned as to *how* the services are fulfilled. The user interface is important as it satisfies the *easy to program* and *architecture-independent* requirements of a programming model described in Section 2.4.

When porting between different communication mechanism neither the system interface nor the user interface would change, only the internal implementation of how the HWTI fulfills the communication mechanism protocol. For instance, porting the HWTI from a On-chip Peripheral Bus to a Processor Local Bus would require signal assignment changes to communicate with the new bus. This change would not effect the user logic implementation.

Details of the HWTI system and user interfaces are in Appendix A. A discussion of similarities and differences between the context of hardware and software threads is in Section 5.1. Also note, creating and joining on threads remains a software only capability. Migrating this service to hardware proved too difficult. This is discussed in detail in Section 5.2.3.

3.2 Abstract Synchronization

Hthreads migrates key system services to hardware. These are specifically mutex, condition variables [56], thread management [35, 73], and scheduling [1, 2]. Each of these cores are accessible through embedded bus transactions. By reading or writing specific addresses within each core's address range the calling thread will be invoking specific functionality. This hardware/software co-design approach is advantageous since access to synchronization functionality is through encoded bus transactions and not exclusively in the execution of sequential code on a CPU [7,9,10]. More importantly, since the functionality is centrally located and accessible through asynchronous bus transactions heterogeneous threads have equal access to their functionality.

Given this, an hthread abstract interface, be it in software or hardware, only has to know how to access the synchronization services within the FPGA. More precisely, an abstract interface only has to know how to encode an API call as a bus transaction. The details of how to do the encoding is abstracted from the user by the interface.

For completeness, there are a number of "init" (initialize) and "destroy" functions within the hthread API, these functions do not require communication with their respective management cores. Rather, these functions only require the interface to initialize or clear the corresponding data structure. As an example in `hthread_mutex_init( &mutex_t, &mutex_attr_t )` an interface implementation only has to set the mutex number and type addressed by `mutex_t` with the values stored in `mutex_attr_t`.

A complete discussion of the semantic and implementation differences between each of the hthread API calls is in Chapter 5. As a note, the migration of synchronization services to hardware, as well as the bulk of the hthread runtime system, was largely completed prior to the start of the research presented in this thesis. It is described in [1,2,7,9,10,56,57,73].

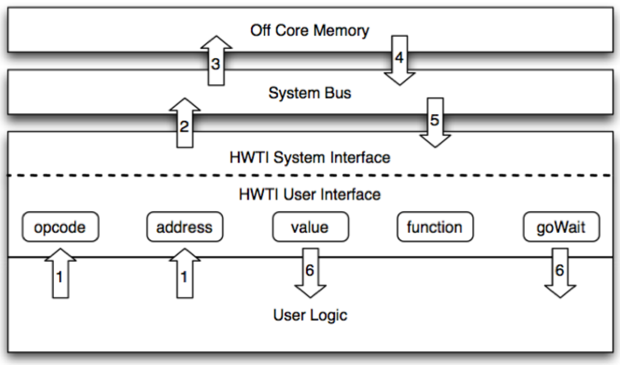
3.3 Abstract Communication

Threads communicate by sharing data in a global memory structure. Shared data is passed to a thread as a pointer to an abstract data structure (typically a `struct` in C) when the thread is created. When a thread starts up it reads the struct and starts execution. However, this creates a problem as it forces hardware threads to understand the meaning of a pointer.

Oddly, most C to hardware tools leave out support for pointers, Altera's C2H being the exception [65]. This is generally because these tools assume the CPU will pass all needed data to the hardware accelerator, instead of a pointer to the data. However, implementing support for pointers is not difficult. Hardware threads simply have to be instantiated as a bus master and use bus protocols to read and write memory. This is often referred to as "direct memory access."

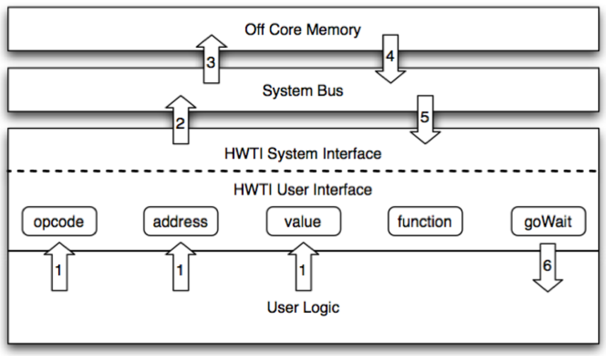
To read a memory location the user logic passes the address to read to the HWTI via the user interface. The HWTI in turn performs a standard bus read operation, the memory location responds, and the HWTI passes the value back to the user logic. This operation is depicted in Figure 3.6. To perform a write, the user logic passes both the address and the value to the HWTI, which in turn initiates the bus write operation. This operation is depicted in Figure 3.7.

In this manner the thread's user logic has access to global memory without knowledge of any low level system details. Bus signals and protocols are successfully abstracted from the user logic. The user logic only has to know how to interact with the HWTI user interface.



1. User Logic issues LOAD opcode, with pointer reference (memory address).
2. HWTI signals the bus to do a a read operation.
3. Bus communicates with the memory core responding to the address.
4. Memory core places the value of the address on the bus.
5. Bus signals the HWTI with the value.
6. HWTI returns the value to the user logic.

Figure 3.6. User interface load operation



1. User Logic issues STORE opcode, with pointer reference (memory address), and value to store.
2. HWTI signals the bus to do a a write operation.
3. Bus communicates with the memory core responding to the address.
4. Memory core acknowledges the operation to the bus.
5. Bus signals the HWTI that the operation is complete.
6. HWTI returns control to the user logic.

Figure 3.7. User interface store operation

Chapter 4

Extending High-Level Language Semantics to Hardware Resident Threads

Abstracting synchronization services for hardware and software threads was achieved by migrated synchronization services to independent hardware cores and creating interfaces, in both hardware and software, that know how to communicate with the appropriate system core. Abstract communication was achieved by providing a facade for direct memory access. Encapsulating hardware thread context was achieved by creating a standard interface representing the context inherent to the `hthread_create` and `hthread_join` calls. These three mechanisms alone can abstract the hardware/software boundary between CPU and FPGA resident threads. However, a *meaningful* abstraction still does not exist for hardware threads.

As a review, the overall goal is to allow software engineers, and not just hardware engineers, to describe a multi-core system on a programmable chip through a known high level language and programming model. Thus far only the programming model abstractions have been moved to hardware threads. A meaningful abstraction is still required to complete the goal. A meaningful abstraction implies that a thread's functionality may be moved from software to hardware without any consideration as to where it will run. Granted, a C to VHDL translator would be needed for the conversion, but the implementation in C should not be modified prior to converting to VHDL. Examples of why this is not possible with the mechanism already discussed are:

- Hardware threads must instantiate their own memory if they want a "scratch-pad" to locally store variables.
- If a hardware thread does instantiates their own memory, it is impossible to pass a pointer from that memory to other threads.
- Although a hardware thread may call a system function, there remains no interface for calling a user defined function.
- Hardware threads can not create, or join on, another thread. Creating a thread remains a software only capability.

The remainder of this chapter discusses four mechanisms added to the HWTI to creating a meaningful abstraction. They are globally distributed local memory, a function call stack, dynamic memory allocation, and remote procedural calls.

4.1 Globally Distributed Local Memory

The memory latency problem, the delay between a load or store request from the CPU to memory, is a well studied and understood problem. In traditional CPU systems it is addressed through memory hierarchies. The fastest and more expensive memories are placed close to the CPU, while the slower and inexpensive memories are placed further away [50]. In more complex architectures, such as chip multiprocessors [23, 111] or network on chip [75], cache and memory organization continues to be studied. However, there has been very little research in memory hierarchies for MCSoPC. De Micheli has looked at ways of representing memory for C to HDL synthesis [81], and Vuletic has looked at ways of using virtual memory in reconfigurable hardware cores [96, 97].

Hardware threads, which operate like application specific processors suffer from the same memory latency problems as CPUs. If hardware threads only operates on off chip data their performance would be greatly degraded. To avoid this penalty hardware threads need access to a localized fast memory. If the data size the hardware thread is operating on is small, the user logic could instantiate registers for each data item. Unfortunately, this solution is not practical for any non-trivial data set. For example, one 32-bit register occupies approximately 16 slices using the D flip flops on a Xilinx Virtex-II Pro FPGA [106]. Instantiating 32 32-bit registers occupies 512 slices. This represents roughly 4% of available slices on a Xilinx Virtex-II Pro 30 chip [107]. Depending on the application's needs, this to may be too expensive.

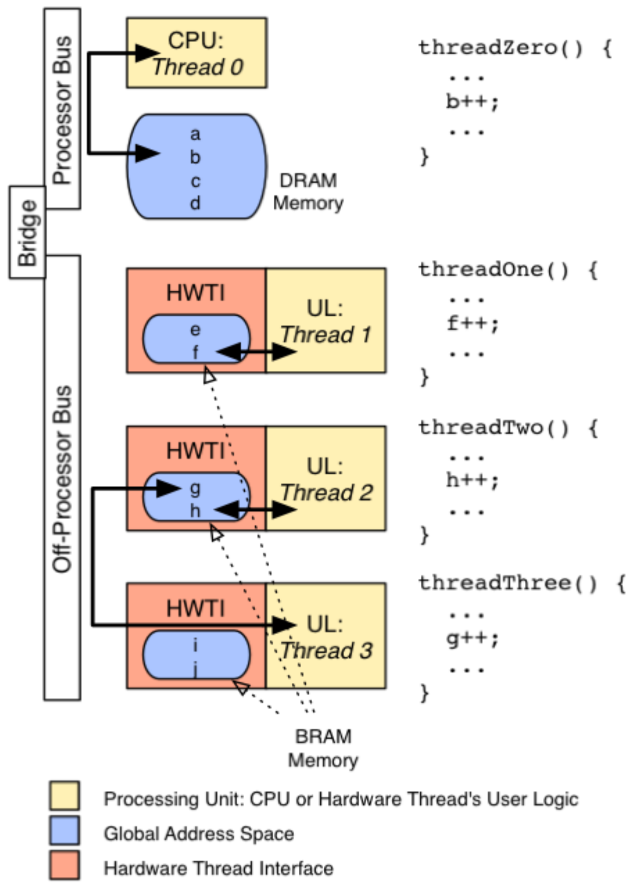
If the hardware thread is to remain small and access shared memory, it is clear that like CPUs, it too will need a mechanism similar to cache. However, for two reasons, a traditional cache is not practical. First the cache management logic would be too expensive to create. The number of resources grows if cache coherency protocols are needed, as is the case with multi-core system on a programmable chip (MCSoPC). Second, assuming a cache could fit within the FPGA resources, multi-core chips are often architected using separate buses or a hierarchy of busses. Snoopy cache protocols are useless on multi-bus architectures.

In a different approach, the HWTI instantiates a "globally distributed local memory." The global distributed local memory, or "local memory" for short, is not cache, but rather a fast memory accessible to both the user logic and other Hthread cores. The local memory is instantiated using on chip dual ported memory embedded within the FPGA fabric (referred to BRAM on Xilinx chips). Depicted in Figure 1.2, one port on the BRAM is used to allow access for the user logic, the second port on the BRAM is used to allow access for other Hthread cores. The local memory address space is a part of the hardware thread's address space. The term "distributed" is a reference to the fact that any two hardware thread's address space is non-continuous. Access to the local memory, for other cores, is through standard bus read and write protocols. The local memory is accessible to the hardware thread's user logic through HWTI's user interface using the same protocol as would be used for accessing traditional global memory. On each `LOAD` or `STORE` operation, the HWTI checks to see if the address range requested is "local" (within the HWTI) or "global" (outside the current HWTI). If the address is local, the HWTI accesses the memory through the BRAM's signaling protocol. If the address is global, the HWTI accesses the memory through bus operations. In this way, the HWTI abstracts the difference between local and global data. The programmer is only concerned with accessing memory, not where the memory resides.

An advantage of the HWTI's local memory, is that the user logic may access it without issuing a bus command. Consequently multiple hardware threads could perform simultaneous memory accesses, even when the threads share a common bus. To illustrate this consider Figure 4.1. In this figure four threads, three hardware and one software, are accessing global memory at the same time. The largest portion of global memory is on the same bus as the CPU. This is normally off chip memory, here shown as DRAM. The three hardware threads are sharing a bus, each thread contains a small segment of memory, shown here as BRAM. If the shared variables were all stored in traditional global memory, accessing them would be slower. Not only would each thread have to perform a bus operation, but the bus would effectively serialize them. When the variables are distributed, as depicted, four memory operations may be performed in parallel, furthermore accessing local memory is very quick. As seen in Table 4.1 it takes 3 clock cycles to load a value from local memory, 1 clock cycle to store a value. This compared with 51 and 28 clock cycles to load and store respectively to off chip memory (labeled global memory). Although slower than accessing its own memory, a hardware thread may access another hardware thread's local memory in 19 clock cycles for either a load or store operation (labeled HWTI memory).

Operation	**Clock Cycles**
`LOAD` (local memory)	3
`LOAD` (HWTI memory)	19
`LOAD` (global memory)	51
`STORE` (local memory)	1
`STORE` (HWTI memory)	19
`STORE` (global memory)	28

Table 4.1. Performance of memory operations



```
threadZero() {
  ...
  b++;
  ...
}

threadOne() {
  ...
  f++;
  ...
}

threadTwo() {
  ...
  h++;
  ...
}

threadThree() {
  ...
  g++;
  ...
}
```

Figure 4.1. Simultaneous memory access between four threads

4.2 Function Call Stack

Implicit in the goal of providing a meaningful abstraction is the goals for the HWTI to provide a target for HLL to HDL translation. As discussed in section 2.5, this is not an easy task. Where as software can target an existing processor's instruction set with a Von Neumann architecture behind it, hardware does not have any preexisting target. Because of this, there is not any pre-existing support for high-level language semantics. Pointers and a function call stack are often two capabilities left out. Support for pointers have largely been solved by translating a pointer address into a bus operation [6, 65]. Both the HWTI and Altera's C2H uses this method to support pointers. However, a hardware equivalent function call stack has not been addressed. Without a stack's functionality, parameter passing is difficult and true recursion is impossible.

To address this problem, the HWTI creates a function call stack using its local memory. The HWTI's function call stack works analogously to software function call stacks. There are three key differences. First, the stack and frame pointer are maintained as registers within the HWTI, pointing to its local memory instead of traditional global memory. During a call, the HWTI pushes the frame pointer and the number of passed function parameters values onto the stack. The stack and frame pointers are then appropriately incremented for the new function.

The second difference is specific to how parameters are passed and stored during the call. RISC CPU's such as the MIPS architecture [71] and the PowerPC use a register convention that reserves general purpose registers to save parameters during a call. However, the HWTI does not maintain any general purpose registers that are shared with the user logic prohibiting it from using this option. Instead it uses a method similar to CISC architectures, like the x86 where all function parameters are passed by pushing the values onto the stack. The HWTI has a `PUSH` operation for this purpose. Once called, the callee function reads the the parameters by using a `POP` operation.

The third difference is instead of saving the contents of the program counter during a function call, as done on CPUs, the HWTI pushes the user logic's *state machine's return state* onto the stack. The user logic is required to pass the return state to the HWTI, along with the function to call, during a `CALL` operation. To be more specific, the user logic passes a 16-bit variable representing the return state. The user logic is responsible for mapping this variable to its return state when control is returned to the caller function.

Function returns are implemented with the `RETURN` operation. Here, the stack register is set to the current frame register (minus the number of previously pushed parameters that was stored on the stack), and the frame registers is restored by popping the value from the stack. The return state and return value, limited to 32 bits, are passed back to the user logic.

The HWTI supports calling system and library functions, as well as user defined functions. The interface and protocol for calling any type of function is the identical for the user logic. The implementation difference is that for a system or library call, the HWTI performs the method on behalf of the user logic. For a user defined function call, the HWTI sets up the function stack for a new function, and then returns control to the user logic, specify the start state of the function.

In order to give the user logic easy access to the local memory the HWTI supports similar semantics to HLL variable declarations. To declare local variables, the user logic uses the `DECLARE` operation, with the number of words (4 bytes) in memory it wants to set aside for local variables. The HWTI reserves space on the stack by incrementing its stack pointer the specified number of words. The user logic access this memory using `READ` and `WRITE` operations in conjunction with an index number that corresponds to the declared variables. The first declared variable has index 0, the second declared variable has index 1, and so on. Since the variables are declared and granted space with the HWTI's local memory, they each have an address in global memory. The `ADDRESSOF` operator works by converting the index number into its equivalent memory address, taking into account the HWTI's base address and current frame pointer.

Using local memory as a mechanism to create a function call stack consequently allows the HWTI to support recursive function calls in hardware. A hardware thread may repeatedly call the same function without incurring the costs of duplicating function logic within the FPGA fabric. The caller function's state is saved to the HWTI's local memory, and then restored when the callee function returns. The recursive depth of a function is only limited to the availability of local memory. Two examples of recursive functions are given in the results section, they are quicksort (section 6.2.1) and factorial (section6.2.2).

To help understand how the HWTI function call stack is implemented, consider the pseudo code, and stack representation, given in Figure 4.2. This image depicts the state of the HWTI after calling the `foo` function.

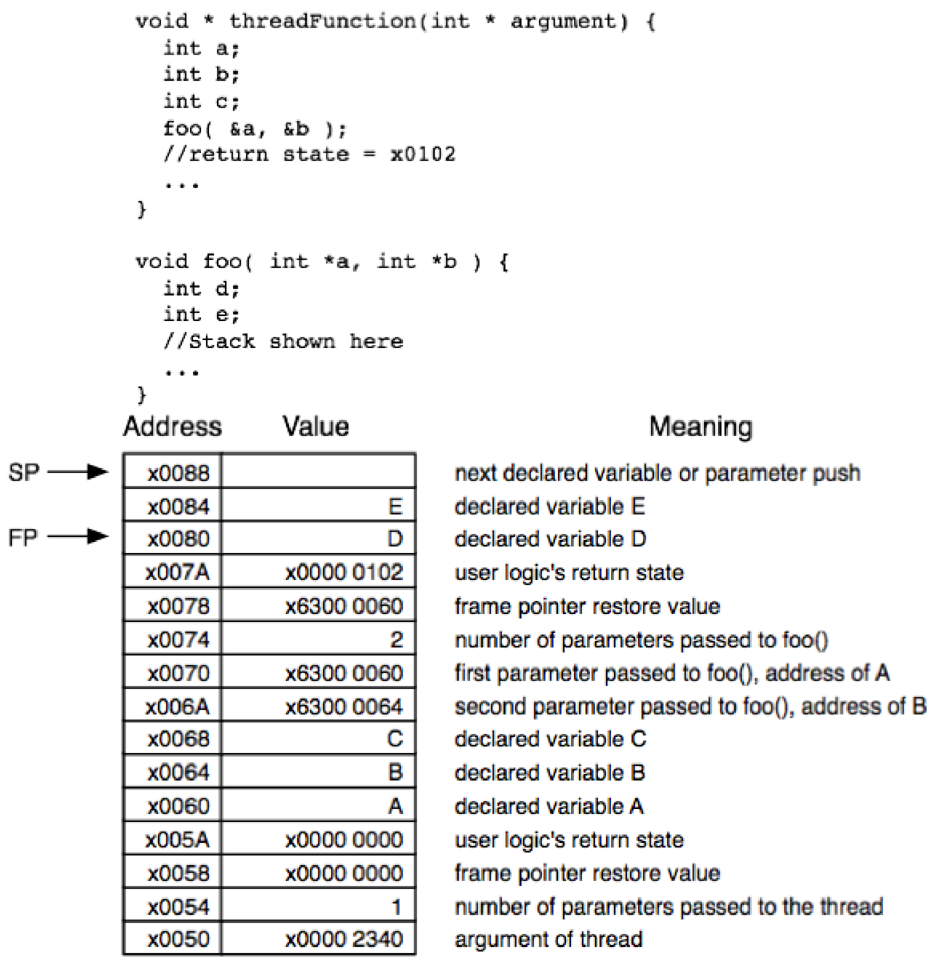
Lastly, Table 4.2 lists the performance of operations associated with the HWTI support for function calls, variable declaration, and variable use.

Operation	**Clock Cycles**
`POP`	5
`PUSH`	1
`DECLARE`	1
`READ`	3
`WRITE`	1
`ADDRESSOF`	1
`CALL`	3
`RETURN`	7

Table 4.2. Performance of function call stack operations

4.3 Dynamic Memory Allocation

In the related work Section 2.5 a number of HLL to HDL tools were discussed. Although these tools have made significant progress over the past decade to automatically and correctly translate an HLL source code into a hardware core



```
void * threadFunction(int * argument) {
  int a;
  int b;
  int c;
  foo( &a, &b );
  //return state = x0102
  ...
}

void foo( int *a, int *b ) {
  int d;
  int e;
  //Stack shown here
  ...
}
```

	Address	Value	Meaning
SP	x0088		next declared variable or parameter push
	x0084	E	declared variable E
FP	x0080	D	declared variable D
	x007A	x0000 0102	user logic's return state
	x0078	x6300 0060	frame pointer restore value
	x0074	2	number of parameters passed to foo()
	x0070	x6300 0060	first parameter passed to foo(), address of A
	x006A	x6300 0064	second parameter passed to foo(), address of B
	x0068	C	declared variable C
	x0064	B	declared variable B
	x0060	A	declared variable A
	x005A	x0000 0000	user logic's return state
	x0058	x0000 0000	frame pointer restore value
	x0054	1	number of parameters passed to the thread
	x0050	x0000 2340	argument of thread

Figure 4.2. HWTI's function call stack

or accelerator there are still limitations that are desirable to engineers. One of the most common limitations is dynamic memory allocation. Traditionally managed by the operating system, users have access to dynamically allocated memory through tools such as `malloc` and `free`. In [101] the authors survey many of the allocation and deallocation techniques for software based memory management. Despite memory allocation use and studies within software, with the exception of [81], a dynamic memory allocation for custom hardware cores has thus far been illusive. This is primarily because hardware cores do not have access to the operating system, which manages memory allocation. It would be possible for the HWTI to use its remote procedural call (discussed in section 4.4) to allocate and deallocate memory for the user thread. However doing so would allocate memory in traditional global memory. Consequently, the hardware thread would pay a latency price for accessing it.

Taking advantage of the existing design of the HWTI, namely its ability to provide operating system services and its local memory, the HWT implements its own light weight versions of `malloc`, `calloc`, and `free`. Like the existing Hthread library functions, when the user logic calls `malloc`, `calloc`, or `free`, the HWTI implements these functions on behalf of the user, acting like an operating system. The memory the HWTI allocates for the user logic is within its local memory.

To implement dynamic memory allocation the HWTI adds two limitations. First the same thread that allocates memory must deallocate it. Second, since the dynamic memory is allocated within the thread's local memory, there is a limit to the size and number of memory segments that can be allocated. The memory the HWTI allocates, known as the heap, is pre-allocated in 8B, 32B, and 1024B segments at the top of the local address range. These sizes were selected to assist with the dynamic creation of mutexes, condition variables, and threads, common structures within Hthreads programming. By preallocating memory the HWTI avoids implementing a defragmentation routine. This is advantageous since the HWTI should remain as small as possible. When the user logic calls `malloc`, the HWTI selects, using a "best-fit" algorithm, the smallest appropriate preallocated memory space and returns its address to the user. The HWTI marks the memory used in a malloc state table. If the requested memory size is larger than 1024B, the HWTI allocates this space by decrementing a heap pointer the specified amount and returning the appropriate address to the user. The heap pointer is maintained, like the frame and stack pointer, as a register within the HWTI, always pointing to an address within its local memory. The user logic may request only a single segment of memory larger than 1024B. If the user logic requests a memory space larger than the HWTI has available, the HWTI returns a null pointer (represented as address x0000 0000). When the user logic calls `free`, the HWTI marks the appropriate malloc state table entry as unused.

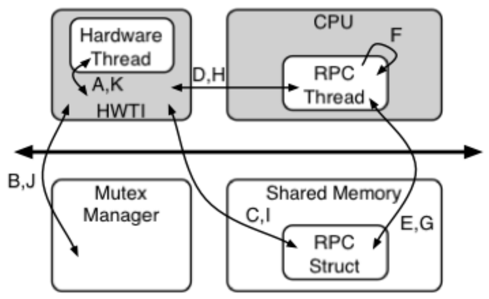
A significant advantage that the HWTI `malloc` routine has is that the memory it allocates is local and globally accessible. In [75], the authors discuss the problems of dynamic memory allocation for multi-core systems on a chip, specifically looking at softcore processors. Stating the primary challenge is how to allocate memory for a processor that is fast, but without the benefits of a data cache (multi-soft-core systems have the same issues with cache coherency as discussed in section 4.1). Their solution was to identify, at compile time, which dynamic memory allocation could be local and held in cache, and which have to be global. To be safe, they had to put any data in global address range, that could not be guaranteed to only be used by the local core. The HWTI alternatively, when it allocates memory (or even when it declares memory on the stack), all memory is globally accessible and fast for the calling thread. Eliminating the problem of segmenting memory at compile time as to what can be placed in global memory and what can be placed in local memory.

4.4 Remote Procedural Calls

There are some hthread API calls that are simply too difficult, or too expensive to implement as part of the HWTI, most notably `hthread_create` and `hthread_join`. However, to create a meaningful abstraction, their inclusion is necessary. Furthermore it would be beneficial to enable support for all library functions, most specifically C's standard library.

To enable support for all other software resident library functions, the HWTI relies on a remote procedural call (RPC) methodology to a special software system thread. This model is similar to hardware thread callbacks described in [97]. The RPC model uses existing hthread mechanisms and is completely abstracted from the user. A block diagram of the RPC methodology is in Figure 4.3. The user logic makes the library function call to the HWTI in the same manner as it would any other function. The HWTI recognizes the function (the function opcode has to be known at synthesis time) as an RPC function. Using a mutex, the HWTI obtains a lock protecting the RPC mechanism. Once granted, the HWTI writes the opcode and all arguments to a global RPC struct. Using a condition variable the HWTI signals the RPC thread to perform the function. The HWTI then waits for a return signal from the RPC thread. The RPC thread, which runs as a software thread on the CPU, will read the RPC struct and call the appropriate function (the function must be known at compile time) with the passed in arguments. The return value, if any, is written back to the RPC struct. The RPC thread signals the HWTI indicating the function is complete. When the HWTI receives the condition variable signal, it wakes up, reads the return value, unlocks the RPC mutex, and passes the return value to the user logic.

The RPC model may be used to support any library function or operation too expensive to implement within the FPGA. For instance, Table 4.3 lists a number of functions the RPC model was used to implement along with their execution time. The RPC model does have disadvantages. The CPU has to be interrupted



A: UL calls a function not supported by HWTI.
B: HWTI obtains lock on RPC.
C: HWTI writes opcode and arguments to RPC struct.
D: HWTI signals RPC Thread and waits.
E: RPC thread reads the opcode and arguments.
F: RPC performs function on behalf of HWT.
G: RPC writes result to RPC struct.
H: RPC signals HWT operation is complete.
I: HWTI reads results from RPC struct.
J: HWTI releases lock on RPC
K: HWTI returns results to UL.

Figure 4.3. Remote procedural call high level diagram.

to perform the RPC which may impact real time constraints. Depending on the priorities of the threads in the system, a hardware thread may have to wait a significant amount of time before the RPC is complete. Multiple hardware threads may be trying to access the RPC thread simultaneously, although a mutex protects the mechanism, hardware threads will may have to block until they gain access to the RPC thread. Finally, even when the HWTI gains immediate access to the RPC mechanism, there is a performance penalty due to the RPC protocol and RPC thread context switching. There is an approximate 106μs penalty for invoking a RPC function, which does not include the cost of executing the function. The penalty is high enough that implementing the function with a RPC is orders of magnitudes greater than if it was implemented directly in the HWTI. For instance, `malloc` is executed in .1μs as a HWTI resident function and in 122μs as a RPC function.

Library Call	Execution Time
`hthread_create` (hthread.h)	160μs
`hthread_join` (hthread.h)	130μs
`malloc` (stdlib.h)	122μs
`free` (stdlib.h)	120μs
`printf` (stdio.h)	1.66ms
`cos` (math.h)	450μs
`strcmp` (string.h)	114μs

Table 4.3. Performance of selected library calls using the RPC model.

4.5 Unsupported CPU Features

In the above sections the HWTI features enabling a meaningful abstraction were described. This discussion would not be complete without also including, at least briefly, a description of features that may be argued for to achieve the goals of the HWTI.

The first such feature is the inclusion of global variables. Currently, there is no way automatically share global variables created in software with a hardware thread. Doing so would require knowledge of global variable addresses. Global variable addresses are resolved during software's link time. Although a software thread have access to this information, a hardware thread that was synthesized separately (and usually earlier) does not. If global variables have to be used, a programmer has the option of passing the address of a global variable to a hardware thread in the thread's argument struct. In this case, a global variable in software becomes just a "normal" variable to hardware.

A second feature that the HWTI could provide is a standard general purpose register set. This could be useful during function calling. As described in Section 4.2 all parameters must be pushed onto the HWTI's stack instead of quickly stored in general purpose registers as typically done with RISC architectures. The reason the HWTI does not provide general purpose registers is first, and simply, the HWTI is not a CPU. The HWTI provides services and the interconnect for the user logic, which acts more like a CPU. Second, if the HWTI did provide a general purpose register set, it would limit the user logic's options for instantiating its own register set. Some hardware thread user logics may be designed to run with few or even zero registers, while others may best perform with many registers. The option of how many general purpose registers should be a implementation decision of the user logic entity.

Chapter 5

Semantic and Implementation Differences between Hardware and Software

Extending the threaded programming model to hardware is conceptually simple. This thesis demonstrates that such a move is possible. However, with this migration, there remains key questions to be asked and solved: what is the relationship between a software thread context and a hardware thread context? is a clean system call implementation in hardware possible? are there performance limitations? is the migration natural? or does a system call take on a new meaning for a hardware thread? It is these types of questions that this chapter aims to address.

This chapter first explores the similarities and differences of hardware thread's context and software thread's context. This chapter then compares system calls implemented in software and system calls implemented in hardware using three criteria. They are the semantic differences, that is, what is the meaning of a system call implementation in either hardware or software. Second, the relative size difference for the implementation. Third, the run time performance.

In most cases the difference between hardware and software stems from the fact that in software the CPU can, and should, be shared between multiple threads. A number of the implementation syscall routines for software must take this into consideration. Syscalls like `hthread_join`, `hthread_mutex_lock`, `hthread_cond_wait` and `hthread_yield`, must include code to perform a context switch if the current thread has enter a wait state. Where as in hardware, the resources are dedicated to a single thread. A hardware thread can not, by definition, be context switched off a CPU. For these system calls the hardware implementation is cheaper and faster. In most cases the HWTI blocks the user logic from running until the semantics of the programming models allows it to continue. The disadvantage is if a thread has to be blocked, for instance during `hthread_mutex_lock`, a hardware thread's resources remain unused, alternatively in software the CPU resource may be shared with a different thread.

Another primary difference between hardware and software is that a number of system calls require knowing the current thread ID. In software, this is stored in the `thread_id` register on the Thread Manager. When the CPU needs to know this value it performs a load request to the Thread Manager. However, in hardware, the thread ID is kept local in a dedicated register (on the system interface). Reading the thread ID is just a matter of propagating the output of the `thread_id` register signals.

On both the hardware and software sides, there are a number of system calls that will be identified as effectively "not having a meaning." This is not to say that a hthreads programer should not know conceptually what the system calls are doing, but rather the hthread implementation make these calls "noops." These functions, are mostly the "destroy" system calls. For example, when calling `hthread_mutex_destroy`, the programmer believes he is marking that mutex unusable. However, in hthreads a mutex is physically implemented in the Mutex Manager. It is neither possible to create nor destroy a mutex at runtime. In hthreads mutex are "created" at synthesis time. When calling `hthread_mutex_init` a programmer is simply telling the system which mutex he or she wants to use. Since the Mutex Manager was designed without a mutex destroy interface, the `hthread_mutex_destroy` does not have to make a call to the Mutex Manager to update it. The `hthread_mutex_destroy` function simply returns to the user.

Finally, extending the implementation of traditional software system calls to hardware is for the most part possible. This is largely due to the fact that the logic that controls synchronization and thread management routines within the hthread kernel was previously migrated to hardware. The internal implementation to lock a mutex, signal a condition variable, or join on a thread are all bus read operations. Since a HWTI has a master slave interface, it too can complete these same operations that software does. The one area that was proven to be too difficult to migrate to hardware is the creation, and subsequently joining on, threads. For these system calls, a hardware thread has to rely the remote procedural call model to proxy the syscall implementation. While the RPC model makes these system calls possible, the operations are comparatively slow and potentially lead to deadlock.

5.1 Comparing the Context of Hardware and Software Resident Thread

A significant component of migrating threads to hardware is the creation of an equivalent context. However doing so faces key challenges in that independent hardware threads do not have direct access to the software resident data structures and CPU controls that encapsulate traditional software threads. These include such context items as the CPU's program counter, location of the thread's stack, or even the unique thread's ID. Vuletic's work [96–98] manages this problem by creating a software wrapper for each hardware thread. The context of the thread is split between software and hardware structures. The downside to this approach is that hardware remains a slave to the CPU. To be independent of the CPU hardware resident threads must be able to maintain their own context, which requires hardware resident data structures.

To meet this challenge hardware threads encapsulate the context of a thread through a number of sources. These are the system interface, the Thread Manager, the HWTI resident function call stack, and the user logic entity.

The system interface captures the context implied within the system calls `hthread_create` and `hthread_join`. When a thread is created through `hthread_create` four parameters are passed to the system: a pointer to a thread structure, a pointer to a thread attribute structure, a pointer to the function to run, and a pointer to the thread's arguments.

The pointer to the thread structure is a unique identifier for the child thread. The system populates this structure for the parent thread's use during thread creation. Currently executing software threads learn their thread ID by reading the `thread_id` register maintained by the Thread Manager. Hardware threads alternatively maintain their own independent `thread_id` register as part of their system interface.

The thread attribute structure contains information on how to create a new thread. Although the system needs this information when a software thread is created, the attributes of a hardware thread are either held constant or managed by the Thread Manager. For example, hardware threads are ran with the highest priority, have a constant stack size (set at synthesis time), and have a detached/joined state managed by the Thread Manager.

The address of the start function, an unique identifier for the thread's functionality, is encapsulated within the implementation of the hardware thread's user logic entity. A key difference is that many software threads may be started using the same function pointer. In hardware, to create multiple simultaneous instances of a thread requires instantiating multiple hardware threads with the same user logic code. Ideally the system would map the starting address of thread to a hardware thread base address. This would enable true seamless creation of hardware or software threads. When a user creates a thread, if a hardware thread that maps to the function the user is trying to create is not used, the system would create a hardware thread instead of creating a software thread. Currently this mapping does not exist, identifying which thread to create is managed by the user through setting a hardware threads base address in the thread attribute structure.

To start executing, hardware threads receive a `RUN` command in their `command` register. The `RUN` command is received from the Scheduler when the Scheduler in turn receives an `add_thread` command. This is equivalent to the CPU performing a context switch, learning the thread ID of a new thread (one that has not ran on the CPU yet) and loading the starting address of a software thread into the program counter.

The thread's argument is passed to the hardware thread during the create call by the system to the `argument` register on the system interface. The HWTI copies the argument value to its stack making it available to the user logic through its user interface `POP` operation. This is similar to software where the argument value is placed on the new software thread's stack.

When a hardware thread exits the return value is stored in its `result` register. When the parent thread calls `hthread_join` on a hardware thread the system reads the result value from the `result` register. In software, the return value is stored within an internal data structure.

In software, a thread exists within a finite set of statuses. A software thread is either running on the CPU, waiting for the CPU in the ready to run queue, blocked on a mutex, condition variable, or child thread, or exited. Hardware threads have similar status but with two key differences. At start up, hardware threads have an additional "not used" status due to their physical existence within the FPGA fabric. Second, although a hardware thread may wait on a mutex, condition variable, or child thread, their status is more accurately described as "blocked." This is due to hardware's threads unique nature that they persist with the FPGA fabric. A blocked status, enforced by the HWTI, prevents the user logic from interacting with the system. The status of a hardware thread may be read at anytime through its `status` register.

During execution of a thread, the context of a thread is encapsulated, at a high level, by a set of instructions and data. At a more detailed level instructions in hardware threads are encapsulated within the functionality of the user logic. The HWTI does not restrict users to any specific implementation of the user logic, but it is convenient to use sequential state machines (or at least think of the user logic as a sequential state machine). If this is the case there is a rough 1 to 1 mapping of instructions in a software program to states in a hardware implementation.

The function call context for a software thread is stored within a memory based stack with the CPU maintaining the stack and frame pointers. Hardware threads have an equivalent function call stack but implemented within the HWTI's local memory, stack pointer, and frame pointer.

Data is conceptually the same for both software and hardware threads. They each operate in the same global memory address space. Software threads allocate space for local variables within their stack space, as do hardware threads. Again, the difference is the location of the stack. Also, unless a hardware thread programmer creates his or her own memory representation within the user logic entity, all memory declared by the user logic remains globally accessible. There is no "scratch-pad" memory.

Software and hardware threads each have access to dynamically allocated memory. When software threads call `malloc` or `calloc` the system allocates memory from the traditional global memory based heap. Hardware threads receive dynamically allocated memory from within the HWTI's heap. Both types of allocated memory may be shared with other threads.

During operation the CPU uses a set of general purpose registers to perform local data operations. The user logic may implement its own set of general purpose registers for the same purpose. The HWTI does not restrict or mandate the user logic to any fixed number of registers, or even any registers.

Lastly, the CPU uses data and instruction caches to speed up performance. Instruction caches do not have a equivalent context in hardware since all "instructions" are synthesized as a circuit. Hardware threads also do not use, or even have access to, a data cache. This is largely due to the expense of creating a cache within the FPGA. However, since the HWTI's local memory is relatively fast, it provides a "cache like" performance to hardware threads.

Creating an equivalent context in hardware is important as it enables a consistent computational model across the CPU/FPGA boundary. Both software threads running on the CPU and hardware threads running in the FPGA may be viewed as independent homogenous sequential computational units. Without this equivalent context a consistent programming model targeting MIMD architectures would not be possible. It is important to note however that the context of a hardware thread is not *equal* to the context of a software thread, there is only an equivalency between them.

5.2 Semantic Differences

The functional difference between a syscall implemented in hardware and the same syscall implemented in software is non-existent. A programmer may use the same system call function in either hardware or software and produce equal results. This is of course a fundamental tenant of abstracting the hardware/software boundary. However, there are some subtle and important difference in a set of syscall implementations. It is these subtle difference that will be explored and elaborated on next.

5.2.1 hthread_attr_init

The meaning and practical implementation for `hthread_attr_init` is the equivalent for hardware and software. In each instance the user passes a pointer to a hthread attribute's structure. The system call initializes the attributes with default values and then returns to the user. The default values specify that a thread created with these attributes should be joinable, ran in software, have an initial stack size of 16KB, a null stack pointer (the stack is allocated during `hthread_create` if it is null), and have a priority of 64. The implementation of this function is straight forward, as it only requires storing predetermined values to the hthread attribute structure.

5.2.2 hthread_attr_destroy

The implementation meaning of `hthread_attr_destroy` is in fact the same as the meaning of `hthread_attr_init`. The passed in attribute struct is set to default values, and then control is returned to the calling function. In software there are two separate functions (although the compiler may use common code elimination) between the `hthread_attr_init` and `hthread_attr_destroy`. Because the operations are exactly the same, the HWTI implements `hthread_attr_destroy` function by redirecting to the `hthread_attr_init` set of states.

5.2.3 hthread_create

Creating a thread is the process of starting a new concurrent thread of control. In much the same way as a programmer does not care which processor an individual *Pthread* may be scheduled, the distinction as to where an *hthread* is running (either hardware or software) is also irrelevant. A subtle but important difference in creating a hardware thread, verses a software thread, is that a hardware thread has dedicated resources that run in true concurrency, instead of the pseudo-concurrence prevalent with software threads.

When a software thread is created, the system creates a new thread from the Thread Manager, sets up the thread's attributes in an internal data structure, sets the thread's priority in the Scheduler, dynamically allocates space for the thread's stack (if space was not allocated previously and referenced in the attributes), and finally the thread is added to the ready to run queue. Conceptually, the process of creating a hardware thread is much the same. There are however a few key differences. The first difference is that the system does not need to dynamically allocate space for the thread's stack since the stack is allocated using the HWTI's local memory during synthesis. Second, the system must set the thread's argument in the HWTI's `argument` register, instead of the internal data structure. Finally, the priority of a thread, when passed to the Scheduler, is set to the address of the HWTI's `command` register. This is the indication to the Scheduler that the thread is a hardware thread and not software. Hardware threads practically run as the highest priority in the system. As a matter of fact, the only hthread core that knows which thread is hardware and which is software is the Scheduler. This is a key subtlety as its permits the hthread cores to treat hardware and software threads simply as "threads."

If hardware threads were restricted to creating only other hardware threads, the creation process would be straight forward. The parent thread would only interact with the Thread Manager, Scheduler, and the child's HWTI. The interactions would be performed through standard bus read and write protocols. The data structure used internally of the software kernel would not need to know about the created thread. However, the process of creating a software thread from a hardware thread is quite difficult. The reason is two fold. First the hardware thread would need access to the internal data structure holding thread attributes.

Although somewhat kludgy, this could be accomplished by passing a pointer to the data structure to each hardware thread prior to execution. Alternatively, the data structure could potentially be maintained by expanding the Thread Manager. However, the second difficulty presents a larger problem, namely how to dynamically allocate space for a software thread's stack.

The HWTI does have a limited dynamic memory allocation capability. However, when the HWTI allocates memory it does so through its local memory. The current implementation of the HWTI gives each thread slightly less than 32KB of local memory. The default size of a software stack is 16KB, at this size, a hardware thread could, at most, create one thread without failing (not being able to create a thread). Furthermore, the software thread would have to operate out of the hardware threads memory instead of traditional global memory. Given these two reasons, it was decided that trying to create a software thread from hardware directly was not practically possible.

The first proposed method to address this problem was to expand the Thread Manager to hold all of the thread's attributes. To solve the dynamic memory allocation problem the HWTI could issue a CPU interrupt. The CPU would then context switch to a special "create a thread" thread, read the attributes from the Thread Manager, and create the thread as normal. Once the interrupt is issued, the HWTI could return control to the user logic with the assumption that the thread was created successfully. However, creating a thread could still fail if the call to `malloc` fails, or there are no more thread IDs available. The parent thread, who called `hthread_create`, would have received a SUCCESS return value even though the create failed. This would be a significant redefinition of the create a thread process.

The second proposal, and the one used to implement creating a thread from hardware, is to implement `hthread_create` using the remote procedural call model discussed in section 4.4. Using the RPC model, a hardware thread may create either software or hardware threads, without redefining the meaning of `hthread_create`. The disadvantage is the CPU has to be interrupted to perform the RPC. Furthermore, depending on the priorities of the various threads in the system, a hardware thread may have to wait a significant amount of time before the RPC is complete.

An important consequence in using the RPC model to create new threads is that the RPC system thread will always be seen by the Thread Manager as the parent thread. This of course is incorrect since the true parent is the hardware thread that originated the call. Thankfully, the practical consequence of the proxy relationship is trivial. The parent thread ID stored by the Thread Manager is only used when the parent thread joins on the child thread. As discusses next, the hardware thread implementation of `hthread_join` also uses the RPC thread to join on threads, thus addressing the problem.

5.2.4 hthread_join

Joining a thread is a synchronization policy allowing a parent thread to wait for one of its child threads to exit. This meaning is the same regardless of hardware or software.

The only implementation difference for joining on a hardware thread verses joining on a software thread is where the result value is pulled from. In software it is pulled from the exiting thread's stack, where as in hardware, it is pulled from the HWTI's system interface `result` register.

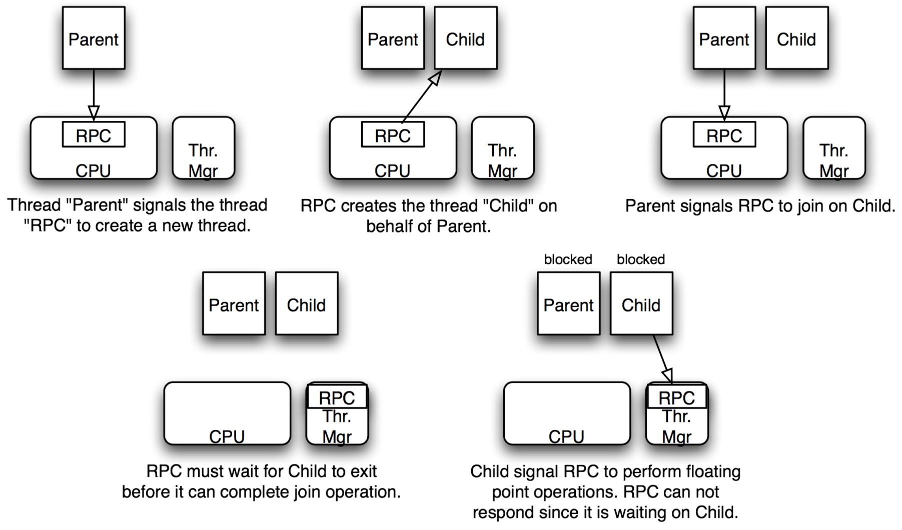
In much the same reason that the hardware implementation for `hthread_create` has to use the RPC model so too does `hthread_join`. The implementation in hardware is consequently the set of states to call the RPC software thread.

There is a potential of deadlock using the RPC model to call `hthread_join`. This danger lies when the RPC thread has to wait for the child thread to exit. While the RPC is waiting, all other threads trying to use the RPC must also wait, since the RPC thread would not be allowed to run.

To illustrate this consider the example in Figure 5.1. In this example a hardware thread creates a child thread to perform an operation. For this example, assume that the child takes a relatively long time to complete. The parent hardware thread at some point will join on the child thread. Since the join is implemented as an RPC that hardware thread signals the RPC thread to perform the join on its behalf. The RPC thread receives the signal and executes the join. However, the child thread has not yet completed its execution, therefor the system context switches the RPC thread off of the processor. The RPC thread will not be allowed to resume execution until the child thread has exited. In the meantime, the child thread needs the RPC mechanism to perform a floating point operations. The child thread must block until it has access to the RPC. However, the RPC is waiting on the child thread. The system is in deadlock.

5.2.5 hthread_self

The practical meaning of `hthread_self` is to return the thread ID of the calling thread. `hthread_self` technically returns a `hthread_t` variable, however, `hthread_t` is an enumerated unsigned int representing the thread ID. The meaning between hardware and software is the same.



Parent
RPC
CPU
Thr. Mgr
Thread "Parent" signals the thread "RPC" to create a new thread.
Parent
Child
RPC
CPU
Thr. Mgr
RPC creates the thread "Child" on behalf of Parent.
Parent
Child
RPC
CPU
Thr. Mgr
Parent signals RPC to join on Child.
Parent
Child
CPU
RPC
Thr. Mgr
RPC must wait for Child to exit before it can complete join operation.
blocked
blocked
Parent
Child
CPU
RPC
Thr. Mgr
Child signal RPC to perform floating point operations. RPC can not respond since it is waiting on Child.

Figure 5.1. Example of danger using `hthread_join` in hardware

The implementation difference is subtle but important. In software, the CPU must read the Thread Manager's `thread_id` register. In hardware the thread ID is stored within the HWTI system interface's `thread_id` register. The local access of the thread ID information makes this call noticeable faster in hardware than in software. Although the `hthread_self` call is seldom used in practical programming the need to access the thread ID information is used for a number of system call implementations. Having this information local gives the HWTI a timing advantage over software for these cases.

5.2.6 hthread_equal

The `hthread_equal` call is the process of determining if two threads, specified by the `hthread_t` variable are the same thread. Since `hthread_t` is enumerated as an unsigned int, both hardware and software implement this function in the same manner. This is, simply returning the boolean result of testing if the first thread ID is equal to the second thread ID.

5.2.7 hthread_yield

`hthread_yield` has one of the stanchest semantic and implementation difference between hardware and software. The meaning of calling a yield function has traditionally always meant that the thread currently running on the CPU is voluntarily giving up the CPU to allow a different thread (of equal priority) to run on the processor. Note that in Pthreads, the equivalent call is `sched_yield`. The implementation in software is straight forward: add the current thread to the ready to run queue, ask the scheduler for the next thread, and then perform a context switch if the next thread and the current thread are different. However in hardware the meaning is quite different.

A hardware thread, by definition, does not run on the CPU, instead it uses dedicated FPGA resources. Therefor it is impossible for a hardware thread to yield the CPU since it never uses it. Consequently, the hardware thread conceptual meaning of `hthread_yield` is the same as a software thread calling `hthread_yield` but without resulting in a context switch.

The implementation of `hthread_yield` is consequently very different. Hardware implements `hthread_yield` by returning immediately, as opposed to attempting to call the Scheduler for the next thread. Calling hthread_yield in hardware has no effect on the status of the HWTI, and practically only serves as a "noop."

If partial runtime reconfiguration becomes a reality, `hthread_yield` may have a very interesting and different meaning in the future. Take for example a system where threads could switch, in run time, between running in hardware and running on the CPU. In this system, an `hthread_yield` call in hardware may mean to reconfigure (instead of context switch) the user logic portion of the hardware thread, for the next thread in the ready to run queue. The resource that the thread is voluntarily giving up is the HWTI. With partial runtime reconfiguration, the semantic meaning of a context switch may refer to reconfiguring the FPGA resources for a different the user logic.

5.2.8 hthread_exit

The `hthread_exit` call indicates that the thread's work is complete and should no longer be allowed to run. Both hardware and software implementations make a call to the Thread Manager indicating that the calling thread has exited. The Thread Manager in return marks the appropriate thread as exited and add its parent thread to the ready to run queue (assuming the parent thread has joined on the calling thread). This is useful since in software the Thread Manager insures that an exited thread is not given the CPU. In hardware it is still possible for the hardware thread to continue running its processes within the user logic entity. To prevent the user logic from effecting the state of the system the HWTI sets its `goWait` signal to `WAIT` and stops responding to `opcode` requests from the user logic. This may only be changed by a `RESET` command to the HWTI's `command` register.

Calling `hthread_exit` has the same meaning as calling "`return <value>`." The HWTI is able to implement this by checking the return state value stored on the stack. If the return state value is 0x0000, which the HWTI sets during stack initialization, the HWTI knows this is the thread's main function returning and to implement this return as an `hthread_exit` call.

5.2.9 hthread_mutexattr_init

The `hthread_attr_init` function initializes a mutex attribute structure to default values. Within hthreads, this is implemented by setting the mutex type to a blocking mutex and the mutex number to 0. The same mutex attribute may be used in the creation of multiple mutexes, however this is not necessarily wise. Unless `hthread_mutexattr_setnum` is used to change the value of the mutex's number each mutex initialized with the same attributes will point to the same mutex locking mechanism. This is true for both hardware and software. It is important to note that the meaning of `pthread_mutexattr_init` is different, in that with Pthreads there is not a notion of a mutex number, instead mutexes are uniquely created during the `pthread_mutex_init` call.

The hardware and software implementation is the same. The passed in parameter is always a pointer to a mutex attribute structure. It is assumed that the user has previous allocated space for the mutex attribute. The implementations simply set the mutex type and mutex number.

5.2.10 hthread_mutexattr_destroy

`hthread_mutexattr_destroy` call conceptually uninitializes a mutex attribute. However, the implementation in both hardware and software is to simply return immediately to the caller. This is because there is no memory to deallocate (allocated memory for the attribute structure must be done by the user) and there are no actions required by the Mutex Manager.

Considering the implementation of this function is to return immediately, a compiler could remove this function call from the user's code without having any adverse effect on the program's functionality.

5.2.11 hthread_mutexattr_setnum

Unlike Pthreads, in Hthreads the programmer must specify the mutex to use in his or her code. This is performed by setting the mutex number in a mutex attribute structure prior to creating the mutex in `hthread_mutexattr_setnum`. The mutex number is generally a value between 0 and 255. By default the mutex number is 0. The implementation of `hthread_mutexattr_setnum` is simply to store the passed in mutex number to the mutex attribute structure.

5.2.12 hthread_mutexattr_getnum

`hthread_mutexattr_getnum` is the functional opposite of `hthread_mutexattr_-setnum`, it returns, through a pointer, the value of the mutex attribute's number. The implementation of `hthread_mutexattr_getnum` is simply to store the mutex number to the passed in address.

5.2.13 hthread_mutex_init

The `hthread_mutex_init` call conceptually creates a new mutex. However, in hthreads all mutexes are "created" during synthesis of the Mutex Manager. The practical meaning of the `hthread_mutex_init` call is simply to set up a data structure used in later lock, unlock, and trylock calls to know which mutex in the Mutex Manager to interact with.

The mutex structure is specifically defined to be the same as the mutex attribute structure. The implementation, in both hardware and software, is to copy the mutex type and number from the attribute structure to the mutex structure. If the user does not pass in a mutex attribute structure, the mutex is initialized with default values (blocking mutex and mutex number 0).

Although the HWTI does not support recursive or error checking mutexes, it can create either of these types if the passed in mutex attribute has this type specified. However, this can only be achieved through the Hthread API if the mutex attribute structure is passed into the hardware thread.

5.2.14 hthread_mutex_destroy

In Pthreads, the `pthread_mutex_destroy` destroys a mutex, disabling it from future use. In Hthreads this system call has conceptually the same meaning, but works significantly different. This is again because in Hthreads mutexes are created and allocated during synthesis in the Mutex Manager. During runtime is not possible to "destroy" a mutex. When the user calls `hthread_mutex_destroy`, the system simply returns control to the calling function. This is true for both hardware and software. A compiler could remove all calls to `hthread_mutex_destroy` without effecting the programs functionality.

5.2.15 hthread_mutex_lock

Mutexes are a mechanism to allow threads to gain exclusive access to a resource, usually a set of data. The correct way to use a mutex is to request a lock on a mutex (the system will halt execution of the calling thread until the lock is granted) the thread then operates on the corresponding piece of data, and finally unlocks the mutex to allow other threads to gain access to the data. `hthread_mutex_lock` is the system call that allows a thread to lock a mutex.

The Mutex Manager centrally controls mutex ownership. Both hardware and software implementations use encoded bus transaction to the Mutex Manager asking for a lock on the specified mutex. If the lock is granted immediately, the system returns control to the calling function, and the Mutex Manager assigns the calling thread as the owner to the mutex. If unsuccessful, the software system will context switch to the next thread in the ready to run queue. Where as the HWTI will enter a `BLOCKED` status. In hardware, the user logic only sees the `goWait` signal as being a `WAIT` and may be interpreted as meaning the HWTI is still executing the system call. Once a thread has been granted a lock on the mutex, the Mutex Manager sends the thread ID to the Scheduler to be added to the ready to run queue. If its a hardware thread, the Scheduler immediately sends a `RUN` command to that thread's `command` register. The hardware thread interprets this as meaning the mutex lock has been granted, and resumes execution of the user logic by setting the `goWait` signal back to `GO`.

Also important to point out is that the software system call version supports three types of mutexes, blocking, recursive, and error checking. Hardware only supports, and as a matter of fact assumes, blocking mutexes.

There is a subtle but very important difference in the meaning of locking a mutex between a software and hardware thread. That is, if a thread tries to lock a mutex but can not, because a different thread currently owns the mutex, in software the calling thread is context switched off to allow another thread to run on the CPU. In hardware, the HWTI blocks the user logic from further execution. Consequently, the resources of a hardware thread are sitting idle waiting for the mutex, where as the CPU may still do useful work by allowing a different thread to run. Conversely, if a thread has to wait on a mutex, when the thread is granted the mutex, a hardware thread can resume execution within a few clock cycles. A software thread may have to wait in the ready to run queue before it has an opportunity to run again. Interestingly though, a software thread that has a lock on a mutex, and is sitting in the ready to run queue may also be forcing a hardware thread, who has also requested the same mutex, to wait. Preventing hardware threads from being blocked may be a important design decision to make, when deciding which threads should run as software and which to run in hardware.

5.2.16 hthread_mutex_unlock

`hthread_mutex_unlock` is the functional opposite of `hthread_mutex_lock`. It unlocks a mutex the calling thread owns, permitting other threads to gain access to the data the mutex protects. Like the mutex lock function, the logic is centrally located within the Mutex Manager. Both hardware and software signal the Mutex Manager through encoded bus transactions. The only difference is that the HWTI only supports the unlocking of blocking mutexes.

5.2.17 hthread_mutex_trylock

`hthread_mutex_trylock` is similar to `hthread_mutex_lock`. With try lock the system call always returns control to the calling function regardless of the grant's outcome. The system call returns a `SUCCESS` or `FAILURE` value to the user indicating if the lock was granted. `hthread_mutex_lock` and `hthread_mutex_trylock` work exactly the same if the lock is granted, they differ only when the lock is not granted.

With `hthread_mutex_trylock`, if the Mutex Manager can not grant the lock to the calling thread, it does not add that thread to the mutex queue list. Because of this, and unlike `hthread_mutex_lock`, there is no chance for the calling thread to be context switch off the processor or `BLOCKED` from executing in hardware.

5.2.18 hthread_condattr_init

The `hthread_condattr_init` function initializes a condition variable attribute structure to default values. The hthreads implementation simply sets the condition variable number to 0. The same condition variable attribute may be used in the creation of multiple condition variables, however this is not necessarily wise. Unless `hthread_condattr_setnum` is used to change the value of the condition variable's number, each condition variable initialized with the same attributes will point to the same condition variable mechanism. It is important to note that the meaning of `pthread_condattr_init` is different, in that with Pthreads, there is no notion of a condition variable number, instead condition variables are uniquely created during the `pthread_cond_init` call.

The hardware and software implementation are the same. The passed in parameter is always a pointer to a condition variable attribute structure. It is assumed that the user has previous allocated space for the attribute. The implementations simply sets the condition variable number to 0.

5.2.19 hthread_condattr_destroy

The `hthread_condattr_destroy` call conceptually uninitializes a condition variable attribute. However, the implementation, in both hardware and software, returns immediately to the caller. This is because there is no memory to deallocate (allocated memory for the attribute structure must be done by the user) and there are no action required by the Condition Variable Manager.

5.2.20 hthread_condattr_setnum

Unlike Pthreads, in Hthreads the programmer must specify the condition variable to use in his or her code. This is performed by setting the condition variable number in a condition variable attribute structure, that is later passed to a condition variable structure during creation. The condition variable number is a value between 0 and 255. By default the condition variable number is 0.

The implementation of `hthread_condattr_setnum` is simply to store the passed in condition variable number to the condition variable attribute structure. In hardware this function shares the same states as the `hthread_mutex_setnum` function. This is possible since the number of both the condition variable attribute structure and the mutex attribute structure is defined first within the struct. Storing the mutex or condition variable number is the same as storing the number to the address of the attribute structure.

5.2.21 hthread_condattr_getnum

`hthread_condattr_getnum` is the functional opposite of `hthread_condattr_setnum`, it returns, through a pointer, the value of the condition variable attribute's number.

`hthread_condattr_getnum` implementation, in both hardware and software stores the condition variable number to the passed in pointer address. For the same reason as given with `hthread_condattr_setnum`, the get number implementation in hardware reuses the states in `hthread_mutex_getnum`.

5.2.22 hthread_cond_init

`hthread_cond_init` is the process of creating a new condition variable. However, in hthreads all condition variables, similar to mutexs, are “created” during synthesis. The `hthread_cond_init` meaning, in both hardware and software, sets up a condition variable structure that allows later calls to `hthread_cond_wait` or `hthread_cond_signal` to point to the right condition variable.

The condition variable structure is specifically defined to be the same as the condition variable attribute structure. The implementation, in both hardware and software, is to simply copy the condition variable number from the attribute structure to the condition variable structure. If the user does not pass in a condition variable attribute structure, the condition variable is initialized with default condition variable number 0.

5.2.23 hthread_cond_destroy

In Pthreads, the `pthread_cond_destroy` destroys a condition variable, disabling it from future use. In hthreads this system call has conceptually the same meaning, but works significantly different. This is again because in hthreads condition variable are created and allocated during synthesis in a separate hardware core. During runtime is not possible to “destroy” a condition variable. When the user calls `hthread_cond_destroy`, the system simply returns control to the calling function. This is true for both hardware and software. A compiler could remove all calls to `hthread_cond_destroy` without effecting the programs functionality.

5.2.24 hthread_cond_wait

From a simplified viewpoint calling `hthread_cond_wait` puts the calling thread into a wait state. The thread resumes execution when a separate thread issues a signal or broadcast. The implementation of wait is slightly more complex because each condition variable is also associated with a mutex. Each of these items are used in conjunction to allow a thread to wait for a condition variable signal. For example, to correctly use `hthread_cond_wait` a programmer may write:

```
hthread_mutex_lock( myMutex );
hthread_cond_wait( myCondVar, myMutex);
hthread_mutex_unlock( myMutex );
```

With this sequence of code, a thread has three opportunities to be blocked before it can resume execution. The first is on `hthread_mutex_lock`. After being granted the lock, and once `hthread_cond_wait` is called, the system unlocks the mutex and simultaneously waits for the condition variable signal. The third opportunity is after the signal is received. Before the system returns control to the calling thread, the system must reacquire a lock on the mutex.

According to the Pthread specifications, when `pthread_cond_wait` is called the mutex is unlock autonomously with the condition variable wait. To implement this behavior, both hardware and software call to the Condition Variable Manager to indicate a wait, and then call the Mutex Manager to unlock the mutex.

Like locking a mutex, waiting for a condition variable has a subtle but important difference between hardware and software. The difference is when a hardware thread waits for a signal, it has to block execution of the user logic, thus FPGA resources will go unused. In software, the CPU has the opportunity to context switch to another thread, allowing useful work to continue.

5.2.25 hthread_cond_signal

Calling `hthread_cond_signal` conceptually signals the first thread who is waiting on the specified condition variable to run. In Pthreads, the highest priority thread on the condition variable is signaled, however in hthreads, the first thread, not yet signaled, is chosen.

The implementation of the signal system call is the same for both hardware and software. In each case the system communicates the system call to the Condition Variable Manager. The logic as to who gets signaled is managed by the Condition Variable Manager. The "signal" comes in the form of adding the thread ID to the Scheduler's ready to run queue. If the thread ID references a hardware thread, the Scheduler sends a RUN command to the HWTI's `command` register. After a thread is signaled, it does not resume execution immediately, the system must first reacquire a lock on the associated mutex.

5.2.26 hthread_cond_broadcast

`hthread_cond_broadcast` is very similar to `hthread_cond_signal`, the difference is that calling `hthread_cond_broadcast` will signal all threads waiting on the condition variable, instead of the first thread to wait for it. Only one thread will be permitted to run though, this is because the system still has to relock the associated mutex variable.

The implementation of the broadcast system call is the same for both hardware and software and is similiar to the signal system call. In each case the system communicates the broadcast to the Condition Variable Manager. Once received the Condition Variable Manager sends each thread ID waiting on the condition variable to the Scheduler.

5.3 Size Comparisons

Comparing the size of system call implementations in software to hardware is a bit like comparing apples to oranges. Software is of course a series of instructions that feed into the processor. Hardware is a set of logic gates, registers, and wires organized to adhere to some boolean logic function. Furthermore, the generally accepted method of stating the size of a software program is to give the number of bytes of the executable file. With FPGAs the generally accepted method is to state the number slices or (with Xilinx chips) the number of 4 input LUTs. Trying to compare the system call size using these methods is fairly difficult, since in the hthread software implementation all system calls are compiled into a single hthread.a library. In hardware, all supported system calls are generated together into the HWTI. The size of any one system call is difficult to pull out.

To address these problems, I defined the method for calculating the size of a software system call, and the size of a hardware system call.

- The software implemented system call size definition is the number of assemble instructions, as compiled by gcc, for the specified system call function, including any hthread sub-routines and system call handlers it may call. In particular the gcc compiler for the PowerPC 405 will be used, as this is the processor chip on the Xilinx ML310 development board used in the evaluation of this thesis.

- The hardware implemented system call size definition is the difference in slice count, generate by the xst [108] tool, between a hardware thread with base functionality that immediately exits, and a hardware thread with base functionality that makes the specified system call and then exits. The HWTI generally contains the code for all supported system and library calls. For this comparison though, the HWTI will be stripped of all system and library calls, except the one being measured.

The software system calls used in this comparison were taken from Hthread build number 1453 (as recorded by the subversion repository). This is known to be a stable build. With the notable exceptions of `hthread_create` and `hthread_join` the bulk of the software implementation has largely gone unchanged in the previous two years. `hthread_create` and `hthread_join` were modified for interacting with both software and hardware threads.

The results of the size comparisons are in Table 5.1. Note that by definition, the size of the `hthread_exit` call is 0. Reviewing these numbers we can gain some interesting insight as to the why the size of certain functions are what they are.

- The largest software functions, except for `hthread_create`, are all system calls that could result in a context switch. The reason is the context switching code is comparatively large despite being written in assemble language (and not directly in C). This size trend does not carry over to hardware, since hardware threads do not have a context switch mechanism.

- Software functions that a Pthread programmer may predict to be relatively large, such as `hthread_mutex_unlock`, `hthread_mutex_trylock`, `hthread_cond_signal`, or `hthread_cond_broadcast`, are in fact quite small. The primary functionality of these system calls is implemented in dedicated hardware cores whose size is not included in this study. The software and hardwarea portions of these system calls are simply wrappers for encoded bus transactions.

System Call	SW Size (Instruction Count)	HW Size (Slice Count, 4 Input LUT Count)
`hthread_attr_init`	29	116, 203
`hthread_attr_destroy`	29	116, 202
`hthread_create`	619	401, 731
`hthread_join`	403	356, 655
`hthread_self`	15	13, 17
`hthread_equal`	16	69, 112
`hthread_yield`	303	12, 16
`hthread_exit`	581	0, 0
`hthread_mutexattr_init`	17	113, 119
`hthread_mutexattr_destroy`	11	20, 35
`hthread_mutexattr_setnum`	24	137, 231
`hthread_mutexattr_getnum`	16	132, 225
`hthread_mutex_init`	32	268, 466
`hthread_mutex_destroy`	11	26, 49
`hthread_mutex_lock`	412	73, 129
`hthread_mutex_unlock`	75	66, 115
`hthread_mutex_trylock`	95	54, 93
`hthread_condattr_init`	14	78, 132
`hthread_condattr_destroy`	11	20, 35
`hthread_condattr_setnum`	24	137, 231
`hthread_condattr_getnum`	16	132, 225
`hthread_cond_init`	25	198, 337
`hthread_cond_destroy`	11	27, 50
`hthread_cond_signal`	43	115, 197
`hthread_cond_broadcast`	43	117, 204
`hthread_cond_wait`	470	197, 336

Table 5.1. System call size comparison between hardware and software

- The absolute smallest functions, `hthread_mutexattr_destroy`, `hthread_mutex_destroy`, `hthread_condattr_destroy`, and `hthread_cond_destroy`, are each 11 instructions. They are the same size because they are implemented identically. Specifically, they immediately return to the user, providing no effective work. In hardware these functions also return immediately to the user and are among the smallest hardware implemented functions. In hardware their sizes range from 20 to 27 slices.

- The smallest system calls in hardware are `hthread_self` and `hthread_yield`, 13 and 12 slices respectively. These functions, like the destroy functions, also return immediately (with `hthread_self` setting the `value` register to the thread ID). The reason `hthread_self` and `hthread_yield` are smaller than the destroy functions is that the destroy functions must reset the stack pointer concurrently with returning, as each of them take a single argument. The `hthread_self` and `hthread_yield` functions do not accept arguments and therefor the user logic does not push parameters onto the stack that later have to be removed.

- Comparatively the largest size difference between hardware and software is `hthread_yield`. In software it is one of the largest at 303 instructions, and in hardware it is the smallest (not including exit) at 12 slices. Understanding this difference requires understanding the meaning of a context switch between hardware and software. At stated in 5.2.7, in software yielding means to give up the CPU to possibly allow another thread to run. Hardware threads can not give up the CPU and therefor `hthread_yield` has no meaning.

- Hardware system calls that share states in their implementations, `hthread_attr_init` and `hthread_attr_destory`, `hthread_mutex_getnum` and `hthread_cond_getnum`, and `hthread_mutex_setnum` and `hthread_cond_setnum`, predictable have the same size, 116, 132, and 137 respectively. Sharing states is possible since these sets of functions have the same hthread meaning and the HWTI may only perform one system call at a time. It is not possible for the user logic to call, for example, call `hthread_attr_init` and `hthread_attr_destory` concurrently. Sharing states is advantageous since both functions may be synthesized without the cost of additional FPGA resources.

- Two of the largest functions in software, `hthread_create` and `hthread_exit` are also the largest two in hardware. In software `hthread_create` and `hthread_exit` are 619 and 581 instructions respectively. In hardware these functions are 249 and 235 slices respectively. Interestingly, they are large for different, but related, reasons. In software these functions are large because they have a lot of requirements to fulfill. Within hthreads `hthread_create` and `hthread_exit` are the only two functions whose semantics are mostly implemented in software. The Thread Manager and Scheduler do play a role in each system call implementation but not to the same extent as other synchronization system calls. In hardware, precisely because the semantic meaning is implemented in the hthread software library, `hthread_create` and `hthread_exit` must be implemented through the remote procedural call mechanism to gain access to the software library. The RPC implementation uses 31 of the 147 states in the HWTI user state machine, which explains the large size of `hthread_create` and `hthread_exit` in hardware.

- Unlike in software, in hardware there is not a clear explanation as to why some system calls are larger than others. In software a system function is small if it returns immediately, it is quite large if it performs a context switch, if it does neither, it is about 30 to 60 instructions long. In hardware, only the system calls that return immediately are small. To learn why some system calls are larger will take more analysis.

  As it turns out there is not a single explanation as to the size of the non-trivial hardware system calls. The size of a system call seems to be combination of three reasons: number of states in the implementation, if the call does one or more load or store operations, and VHDL code techniques.

  Table 5.2 expands on the information shown in Table 5.1. Namely it shows the best case number of states the user state machine transitions through to complete the call, and if the system call implementation does either a load (labeled 'L'), a store ('S'), or both ('LS') operation to any address on the bus. Figure 5.2 charts the relationship between number of states and size of the system call.

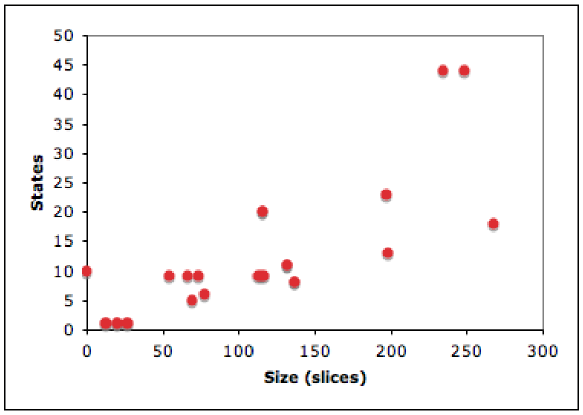


Figure 5.2. Relationship between size of each system call and number of states

System Call	HW Size (Slice Count)	State Transitions	Memory Operations
`hthread_attr_init`	116	20	S
`hthread_attr_destroy`	116	20	S
`hthread_create`	249	44	LS
`hthread_join`	235	44	LS
`hthread_self`	13	1	
`hthread_equal`	69	5	
`hthread_yield`	12	1	
`hthread_exit`	0	10	L
`hthread_mutexattr_init`	113	9	S
`hthread_mutexattr_destroy`	20	1	
`hthread_mutexattr_setnum`	137	8	S
`hthread_mutexattr_getnum`	132	11	LS
`hthread_mutex_init`	268	18	LS
`hthread_mutex_destroy`	26	1	
`hthread_mutex_lock`	73	9	L
`hthread_mutex_unlock`	66	9	L
`hthread_mutex_trylock`	54	9	L
`hthread_condattr_init`	78	6	S
`hthread_condattr_destroy`	20	1	
`hthread_condattr_setnum`	137	8	S
`hthread_condattr_getnum`	132	11	LS
`hthread_cond_init`	198	13	LS
`hthread_cond_destroy`	27	1	
`hthread_cond_signal`	115	9	L
`hthread_cond_broadcast`	117	9	L
`hthread_cond_wait`	197	23	L

Table 5.2. System call size comparison between hardware and software

From Figure 5.2, there is clear relationship between the number of state transitions and the size of a system call. However, this is not an absolute rule. `hthread_create` and `hthread_join` both require 44 state transitions and take 249 and 235 slices respectively. However, the largest system call, `hthread_mutex_init` at 268 slices has less than half the number of state transitions with 18. In this case, the relative large size of the system call is likely due to its VHDL implementation. The code is written such that one state requires the instantiation of two 13 bit subtracters. Subtracters are relatively large circuits to synthesize, and since they exist in a single state, the synthesizer can not take advantage of resource sharing.

There is also an apparent relationship between if a system call uses a load or store operation, and the size of the implementation. On average, a system call with at least one store uses 116 slices, with at least one load uses 104 slices (not including `hthread_exit`), with at least one load and one store 202 slices, and without either a load or store 27 slices. However, this relationship is explained better noting that system calls with either a load or a store have more state transitions that those without, and system calls with both a load or a store have more state transitions than those with only one. This non-relationship is note worthy because there is a strong relationship between the number of loads and stores a hardware implemented system call completes and its execution time.

5.4 Run Time Performance

This section compares the run time performance of the supported system calls between hardware and software implementations. In general the performance of a software system call was measured by calculating the difference between the value of a timer register before and after the system call. The timer register is implemented in a separate hardware core, thus unaffected by any thread running on the CPU or the state of the hthreads system. The performance of hardware system calls was determined in simulation by inspection.

In some cases, it is very difficult to measure the performance of a system call because the system call requires action from other threads. For example, the execution time of `hthread_join` is dependent on wether the thread being join on has exited or not. System calls with these types of dependencies are noted, and the method for measuring them is specified. In two cases, `hthread_exit` and `hthread_cond_wait` a measurement was not possible. The syscall performance results are listed in Table 5.3, with the analysis below.

- With the exception of `hthread_create` and `hthread_join`, the hardware implementation out performs the software implementation. In some instances the degree of improvement is an order of magnitude or greater. `hthread_create` and `hthread_join` are slower in hardware since they rely on the RPC mechanism, which calls out to a software thread to implement the function.

- The biggest relative difference in time is `hthread_self`. In software `hthread_-self` runs in 11.9μs, where as in hardware is runs in .02μs. The reason for the difference is again the semantic meaning of `hthread_yield` in software verses hardware. In software a context switch may remove the thread from the CPU, where as in hardware a thread does not run on the CPU and therefor may not be context switch off. As a note, the 11.85μs listed in Table 5.3 is the time it takes `hthread_yield` when there are not any other threads ready to run. It takes 33.70μs if a context switch is needed.

- In software there two general groups of system calls, those that run in 5μs and those that run in 30μs (and up to 60μs). The set of system calls that execute in 30μs (`hthread_create`, `hthread_join`, `hthread_yield`, `hthread_-mutex_lock`, `hthread_mutex_unlock`, `hthread_mutex_trylock`, `hthread_-`

System Call	SW	HW	Notes
`hthread_attr_init`	6.78μs	1.08μs	
`hthread_attr_destroy`	6.81μs	1.08μs	
`hthread_create`	57.6μs	160μs	Hardware measured on the board.
`hthread_join`	28.6μs	128μs	Child thread already exited. HW measured on the board.
`hthread_self`	7.22μs	.02μs	
`hthread_equal`	5.05μs	.10μs	
`hthread_yield`	11.9μs	.02μs	For software, no other thread in the ready to run queue.
`hthread_exit`			Not applicable, system call does not return to user.
`hthread_mutexattr_init`	5.21μs	.40μs	
`hthread_mutexattr_destroy`	4.07μs	.04μs	
`hthread_mutexattr_setnum`	5.88μs	.27μs	
`hthread_mutexattr_getnum`	5.39μs	.44μs	
`hthread_mutex_init`	5.94μs	.51μs	`mutexattr_t` is NULL.
`hthread_mutex_destroy`	4.21μs	.04μs	
`hthread_mutex_lock`	36.4μs	.39μs	Thread obtains the lock.
`hthread_mutex_unlock`	34.0μs	.39μs	
`hthread_mutex_trylock`	36.6μs	.39μs	Thread obtains the lock.
`hthread_condattr_init`	4.74μs	.30μs	
`hthread_condattr_destroy`	4.04μs	.04μs	
`hthread_condattr_setnum`	5.99μs	.27μs	
`hthread_condattr_getnum`	5.32μs	.44μs	
`hthread_cond_init`	5.24μs	.23μs	`condattr_t` is NULL.
`hthread_cond_destroy`	4.39μs	.04μs	
`hthread_cond_signal`	31.6μs	.44μs	
`hthread_cond_broadcast`	32.2μs	.44μs	
`hthread_cond_wait`			Not applicable, timing depends on other threads in the system.

Table 5.3. Syscall runtime performance comparison, measured in clock cycles, between hardware and software

`cond_signal`, and `hthread_cond_broadcast`) are the exact same set that utilize the 'sc' assemble instruction. 'sc' is the nemonic for system call for the PowerPC. Even though the system call code to was written in assembly language, it is note worthy that all functions using 'sc' have significantly longer execution. The HWTI does not make the distinction between privilege and user mode that CPUs do.

- In hardware the amount of time it takes to complete a system call is directly dependent on the number of load or store operations involved, the exceptions to this rule are system calls utilizing the RPC model. To better illustrate this relationship Figure 5.3 compares the number of load and store operations to the system call execution time (`hthread_create` and `hthread_join` were excluded from this chart). This relationship is because hardware implements system calls via a sequential state machine. Each state, in the state machine only requires 1 clock cycle, except for the load and store states that are waiting for acknowledgments from the bus. This relationship is particularly noteworthy since there is not a relationship between the size of a hardware implemented system call and the number of loads or stores it completes.

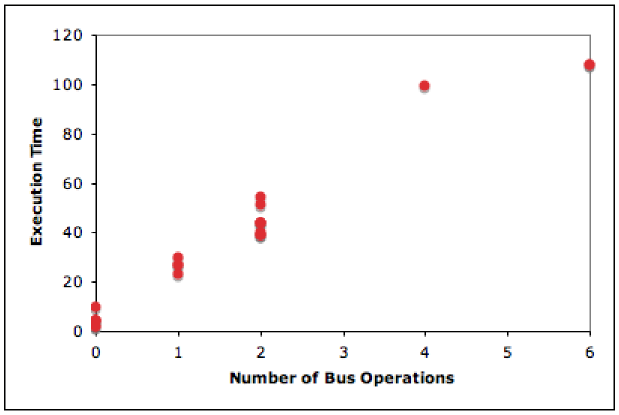


Figure 5.3. Relationship between number of bus operations and execution time for hardware implemented system calls

Chapter 6

Evaluation and Results

In this chapter, the HWTI is evaluated based on conformance testing, stress testing, and its applicability for use with well known algorithms. Conformance tests show that the hardware and software interfaces correctly abstract the synchronization and communication services mandated by the shared memory multithreaded programming model. Stress testing shows that the interfaces produces the same results for repeated calls to the same functions. These two sets of tests make up a Hthread test-suite that is explained in section 6.1. Evaluating HWTI for use in the implementation of well known algorithms demonstrates the HWTI's ability to be used for "real" applications, as well as its performance within this applications. This set of tests make up an Application Testsuite and is explained in section 6.2.

Unless stated otherwise, all tests were conducted on a Xilinx ML310 Development Board [103,107]. The ML310 contains a Virtex II Pro 30 [106] and 256MB of DRAM memory (and a number of other features not used in Hthread evaluation). The Virtex II Pro 30 contains 2 embedded Power PC 405 CPUs (only one used during evaluation) and approximately 30,000 equivalent gates. The Virtex 2 Pro is chosen because of existing institutional knowledge and availability within the ITTC's CSDL lab. The Hthread system was synthesized such that the PowerPC would run at 300 MHz, and the DRAM and all FPGA cores run at 100 MHz. In some cases tests were conducted in simulation. ModelSim was used as the simulation software. Each simulation test was conducted around a bus functional model of Hthreads.

6.1 Hthread Test-Suite

The purpose of the hthread test-suite is to show conformance to hthread system requirements. More specifically it shows the an implemented interface correctly abstracts the communication and synchronization aspects of the shared memory multi-threaded programming model. If an implementation passes all tests in the test-suite, then one can reasonably conclude the interface correctly abstracts the communication and synchronization primitives.

In general, hthread functionality is a sub-set of Pthread functionality. The system calls supported by the HWTI is a further subset (albeit a key subset) of the hthread API. The Venn Diagram in Figure 6.1 shows this relationship. The additional API calls supported by hthreads is due to hthreads hybrid hardware/software model. Conversely, Pthreads is intended only for software.

The creation of a test-suite that can adequately test hthreads is potentially a difficult endeavor. Rather than trying to develop a comprehensive test-suite from scratch, an existing open source Pthreads test-suite was ported to evaluate hthreads. The test-suite chosen was the Open POSIX Test Suite [38]. The Open POSIX Test Suite is a SourceForge project that aims to create a complete, implementation independent, test-suite for the entire POSIX set of APIs. Consequently

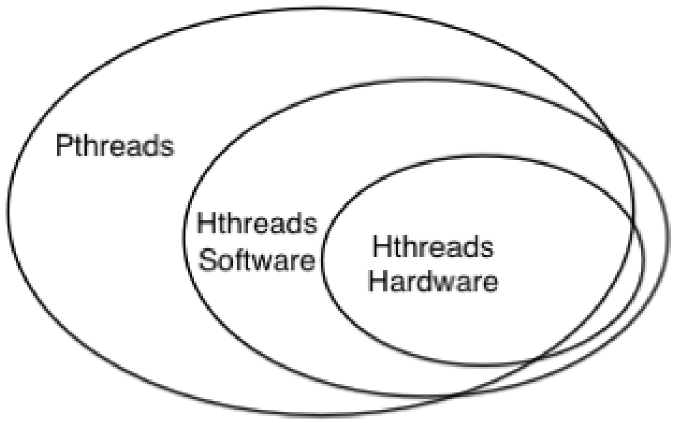


Figure 6.1. Venn Diagram showing relationship between Pthread, hthreads software implementation, and hthreads hardware implementation

the POSIX test-suite includes tests for beyond the Pthreads API, it also includes tests for additional POSIX functionality such as message queues, signals, and semaphores. The complete POSIX test-suite contains over 2000 tests. Of these, roughly 300 tests relate to Pthreads APIs. It were these tests that were the basis of the hthread test-suite. Also, only the "conformance" and "stress" tests were used. Conformance tests evaluate if an implementation method call performs the expected behavior. Stress tests evaluate an implementation's reliability under a heavy load. The last category of tests "functional" were not converted from the Pthreads test-suite. This is largely due to the fact that the Pthread functional tests remain incomplete in the latest version of the POSIX test-suite.

Since the Pthreads test-suite was written with the assumption of a pure software implementation, the test cases for the hthread test-suite were modified to reflect the software/hardware nature of hthreads. Therefore, the hthread test-suite is not be an exact port of the Pthread test-suite. However the hthread test-suite does adhere to the Pthread test-suite's intention. Test-cases that evaluated Pthread functionality not supported by hthreads were not ported. Lastly, some additional test cases had to be added to evaluate the communication and synchronization capabilities across the CPU/FPGA boundary. The best example of this is in the creation of threads. In Pthreads test cases for `pthread_create` just has to evaluate if a new thread (running in software) was created. In hthreads, this test case has to be extended to show that a software thread can not only create another software thread, but that it can also create a hardware thread. Furthermore, the HWTI implementation had to show that a hardware thread can create both a software and another hardware thread.

Two version of the hthread test-suite were created, one to evaluate the software interface, the second to evaluate the hardware interface. Each version evaluates the functionality supported by the HWTI, and was derived from a Pthread test case. The software version was written in C using hthread.h, the hardware version was hand written in VHDL using the HWTI. All software test cases were evaluated with on-the-board testing. Hardware test cases were either evaluated using simulation or evaluated on-the-board. Due to the lengthy time to synthesize a hthread system (up to three hours each), only test cases that could not be simulated were evaluated on-the-board. These generally were test cases that had to create a thread or communicated with a software thread. Thread creation in hardware is done through the remote procedural call mechanism that can not be adequately simulated. Some stress tests were also evaluated in simulation instead of on-the-board because the stress test called for more hardware threads than could fit on the ML310 development board.

Although not done as part of the current research, the hthread test-suite may later be used to also evaluate a C to VHDL compiler that targets the HWTI. If all test cases in the software version work, then a C to VHDL compiler could create the equivalent test case in VHDL. Once synthesized the results of the test cases in hardware should be equivalent to the test cases in software. This type of testing would not only evaluate the HWTI but also evaluate the C to VHDL compiler.

The conformance test cases ranged from mundane to interesting. For example "does `hthread_attr_init` return `SUCCESS`?" was one test case, as well as "does `hthread_create` create a new thread who's thread ID is different from the calling thread?" An interesting twist to the conformance tests is that to confirm that the abstract interface implemented the API call correctly required knowledge of the backend implementation. As an example, to verify that `hthread_mutex_lock` locks a mutex with the calling thread as the owner, required reading the Mutex Manager interface register set. There were a total of 65 conformance test cases applicable for hthreads, with a version of each test case implemented in hardware, software, and where applicable both. All test cases, both hardware and software versions passed.

The stress tests typically involving a sequence of creating (using `malloc`) and initializing hthread structures, a series of API calls on the structures, and finally destroying and deallocating the structures. A few stress tests were designed to continue running until the system ran out of resources. For example, one test continuously created threads, until there were no more thread IDs left. There were a total of seven stress tests written. They were written to be configurable between the number of and type of threads, number of iterations, and, in some cases, number of synchronization primitives (i.e. number of mutexes). Six of the seven stress tests passed for all evaluated permutations. The seventh test, evaluating mutexes, passed all permutations except those involving multiple mutexes. When multiple mutexes were used in this stress test, regardless of the type of thread locking and unlocking the mutex, the Mutex Manager was discovered to send the incorrect thread ID to the Scheduler. This error points out a number of interesting effects of the hthread hardware/software codesign model. First, since the error was centrally located in the Mutex Manager, it was effecting both hardware and software threads. Second, using user level API calls as wrappers for interacting with system hardware cores practically leaves a single location to debug and fix the error. In other words, even though the mutex API calls are implemented with two different interfaces, there is only a single design to fix.

6.2 Hthread Application Suite

The hthread application suite was assembled to demonstrate the HWTI's ability to support well known algorithms. Unlike the test suite which demonstrates the HWTI's conformance to existing system calls, the application suite is targeted towards applicability and overall performance. The algorithms in this section were chosen for various reasons. The quicksort and factorial algorithms are included to demonstrate that the HWTI can correctly maintain a function call stack within a recursive algorithm. The IDEA and Huffman algorithms were implemented based on existing, and well tested, software versions, thus showing the HWTI ability to be a target for HLL to HDL translation. The IDEA and Haar DWT algorithms are included to allow a comparison with previous or different implementations of hardware threads.

All algorithms in this section were implemented as sequential state machines in the user logic entity within a hardware thread. Each thread was translated by hand, from a C version to VHDL targeting the HWTI. The only memory construct created within each user logic entity were registers. Each application depended on the HWTI for global memory access, system and library functions, a function call stack, and local variable address space. The applications were written without regard to the underlying chip architecture, bus structure, or hthread system.

6.2.1 Quicksort

To demonstrate the HWTI's globally distributed local memory and function call stack, the well known quicksort algorithm was implemented as a hardware thread. Quicksort was chosen because it can demonstrate the HWTI's support for recursion, and its performance characteristics are well known and understood.

In the hthread version at start up, the main thread (running in software) creates an array of random integers and passes a pointer to the array with the array length to the quicksort thread. The thread then, using the quicksort algorithm, sorts the array. Using the HWTI's function call stack to handle the recursion and the local memory to store intermediate values, the hardware thread correctly sorts the passed in array.

Figure 6.2 shows the performance of the quicksort thread, using either a software or hardware implementation, and operating on an array in either global memory or local memory (for a software thread this is either running with data cache off or on). The hardware and software versions were purposefully written to be functionally equivalent. Optimizations to take advantage of possible instruction level parallelism in hardware or compiler optimizations in software were not used. The quicksort hardware thread was implemented in 2770 slices.

From these results we can learn the following.

- Migrating quicksort to hardware, targeting the HWTI, does not change the

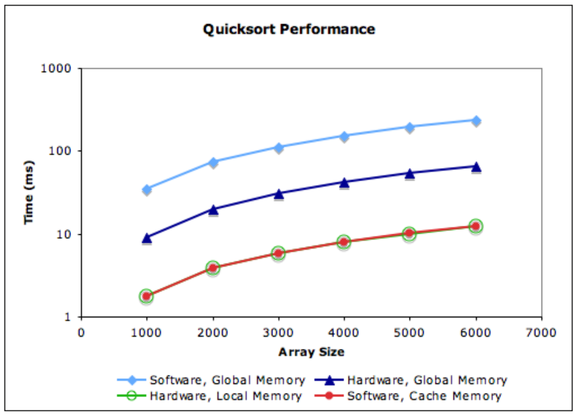


Figure 6.2. Quicksort performance, comparing software and hardware threads, and local and global memory.

performance characteristics of the algorithm. Regardless of where the algorithm ran or the location of data the predicted $O(nlogn)$ complexity exists.

- The HWTI stack correctly manages the recursive nature of quicksort. The recursive depth of quicksort is limited only by relatively cheap BRAM, and not to the more expensive FPGA resources.

- The hardware thread operating on an array in its local memory performs almost identical to a software thread using data cache. This characteristic strongly supports the notion that the HWTI's local memory can give a "cache-like" performance without actually being cache. Indeed, the PowerPC 405 cache has a 1 clock cycle latency, the HWTI's `READ` operation has a 3 clock cycle latency.

- Operating from an array in traditional off-chip global memory limits hardware thread's performance. In this example the hardware thread has a roughly a 6x slowdown, and the software thread has a 19x slowdown. The reason the software thread slows down more, when cache is turned off, is that all of the thread's variables must be reference in global memory. The hardware thread was implemented to only operate on the array to sort in global memory, keeping all other intermediate values as either registers or data within its local memory.

6.2.2 Factorial

Factorial is a well known mathematical algorithm. It is widely used in both statistics and calculus. The operation is loosely defined as:

```
n! = n * (n-1) * (n-2) * ... * 1
```

A practical implementation typically uses a while loop to perform the operation.

```
while ( n>1 ) result = n * (--n);
```

The operation may also be written recursively.

```
int factorial( int n ) {
if ( n > 1 ) return factorial( n-1 ) * n;
else return 1;
}
```

Both versions were implemented as hardware threads using the HWTI, translated from a software version. Although this algorithm is simple, and not a practical stand alone thread, the recursive version adds a second example demonstrating HWTI's support for recursive function calls. Each implementation correctly calculated results for $1 <= n <= 12$. Higher values of n were not tested since the result would have been greater than a signed 32 bit register can hold. The recursive factorial hardware thread version was implemented with 2044 slices, the while loop hardware thread was implemented with 1756 slices.

The performance results of the two implementations is displayed in Figure 6.3. In this graph software execution times with and without cache turned on are provided along with the hardware execution times. The following are lessons learned from this application.

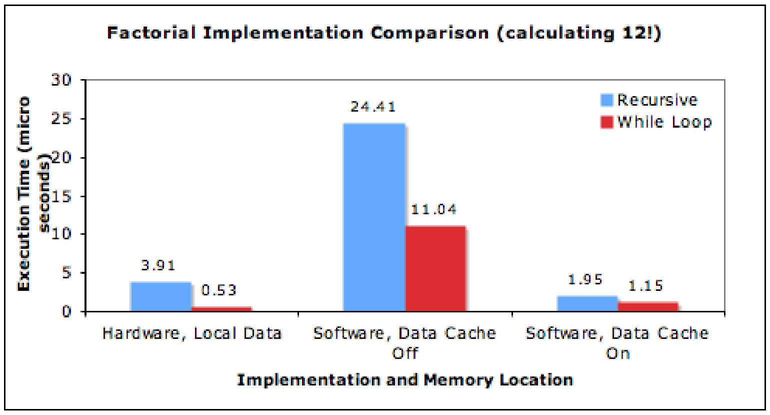


Figure 6.3. Execution times for factorial algorithm, implemented as either a recursive algorithm or a while loop.

- The hardware implementation performs closer to the software version with data cache on. This reinforces the notion that the HWTI's local memory has a "cache-like" performance but without being a cache.

- There is a 7x speedup in the factorial calculation moving from the hardware recursive implementation to the while loop implementation. In software with data cache on there is only a 1.7 speedup. Furthermore, the hardware while loop implementation is faster than software, while the software recursive implementation is faster than hardware. This suggests that the hardware method of calling functions is slower than software's method, this includes the time to save local data to memory to avoid being overwritten.

- Although not apparent in Figure 6.3, as the size of the input (n) grows, the rate of execution time change remains constant for hardware, and near constant for software. Although in a simple factorial example having predictable times is trivial, in a real time application, the ability to exactly know how long a thread will take may be very important. When a hardware thread can operate only on data within it's core, that is operate without bus transactions, it's execution time becomes very predictable. The only variance comes from the input data. Even if a software thread operates only on local data and does not need to call out to any hthread cores, its performance will vary depending on cache misses and access to global memory.

6.2.3 Huffman Encoding

The Huffman Encoding algorithm is a classical compression algorithm first presented in [51]. The algorithm works by making two passes over a data array. In the first pass, the algorithm determines and ranks the most widely used values in the array, and then creates a binary tree that optimally encodes each value. The second pass, encodes and packs each of the original data values with the code derived in the first pass. Ideally, the encoded version of the array will be shorter than the original.

The Huffman Encoding algorithm may easily be divided into multiple tasks, which consequently makes it easily divided into multiple threads. That is, one thread to derive the encoding, the second thread to perform the encoding. Creating the implementation in this manner demonstrates the ability to pass abstract data types between threads.

The hthread implementation started with an existing C implementation of the Huffman algorithm. The original code came from the KUIM image processing library [42], it was modified to work under hthreads. In the final version of the code, the first thread, that creates the code, was kept as a software thread. The second thread, that encodes the data string, was written as a hardware thread. This was intentionally done to demonstrate that abstract data could be passed from a software to a hardware thread. Specifically, the software derivation thread has to pass an input array, output array, size the array, and a separate data structure containing the encoding. Consistent with the hthread programming model, the main thread created this data structure to pass between threads in traditional global memory. Also important to note, the hardware thread was written to be functionally equivalent to the original encoding code from the KUIM library. The Huffman encoding algorithm was implemented in 2479 slices.

Figure 6.4 shows the result of Huffman encoding, comparing the hardware version to a software version, with data cache turned on and off. Looking only at the encoding thread, there is a 4.2x slowdown for the hardware thread compared to the software thread with cache turned on and only a 3.4x speedup over software without data cache. This occurs despite the fact that the hardware thread could use its local memory for temporary variables. This example shows a limitation of converting straight C code into a hardware thread. Although the code can be translated (in this case by hand) and ran correctly, the hardware thread has to pay a stiff price for operating on data out of global memory. It could not take advantage of its local memory to store the encoding array, an array that once it is passed to the encoding thread does not change and gets read potentially multiple times. Nor could it use hardware's bit shifting capabilities in the bit packing portion of the algorithm. These are two examples of techniques a carefully crafted hardware version of the Huffman encoding algorithm may use to increase performance.

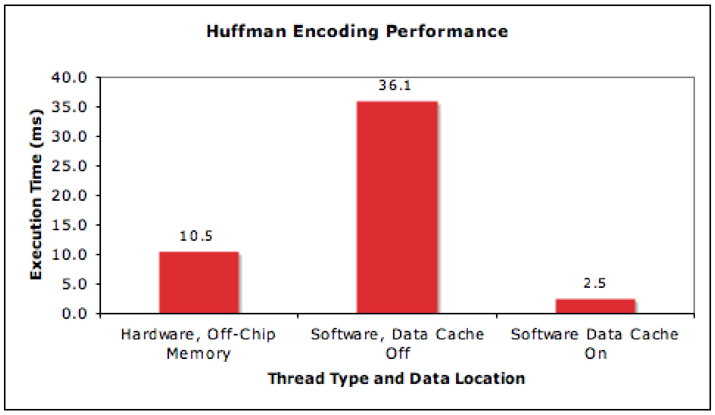


Figure 6.4. Huffman encoding performance, comparing software and hardware threads.

Figure 6.5 shows an even more alarming comparison with total execution time. This is the time from when the first thread is created (derivation thread), determines the coding, passes the data to the second thread (encoding thread), encodes the array, and finally exits. Although using a hardware thread to encode the data array shows a slight speed up over software running without data cache, there is a 16x slowdown compared to software running with data cache. With this example, it is important to remember that in a hybrid hardware/software environment to share data correctly data cache must be turned off. This example is noteworthy since these results demonstrate that any task level parallelism improvements gained by using dedicated hardware threads, must be significant enough to overcome the penalty when you have to turn cache off in the processor.

Amdahl's law [5] teaches us that the maximum speed up due to concurrently

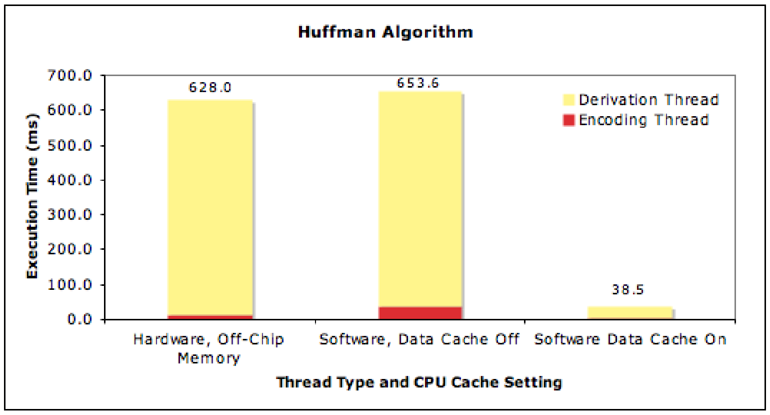


Figure 6.5. Huffman algorithm performance, comparing software and hardware threads.

running tasks is limited by the percentage of sequential code. Even with a hypothetical situation where the encoding algorithm's work could be divided between currently running threads, only 10.5ms of time can be eliminated, the sequential derivation thread still must complete prior to the encoding thread(s). However, the manner in which the Huffman algorithm was implemented follows along the more traditional approach to using FPGAs, that is to use the FPGA as a hardware accelerator (discussed in Section 2.3). The lesson here is not all algorithms are a good fit for the hybrid runtime system.

6.2.4 Haar Wavelet Transform

The Haar discrete wavelet transform is the oldest, and simplest known, transform with othornomal wavelet basis functions [46, 49]. Although Haar is rarely used in practical applications today, wavelet transforms are often used in compression algorithms, such as the Cohen-Daubechies-Feauveau wavelet transform used in the JPEG2000 standard [58]. The Haar DWT is relatively simple to compute, needing only integer arithmetic.

The Haar DWT was added to the test-suite as a means of comparing the existing HWTI version and programming model with a previous HWTI version and a VHDL programming model that used hand optimizations. The existing version of the HWTI, the one described in this thesis, is referred to as "Version 3." "Version 2" was a prototype of the HWTI, described in [6], that provided a minimal set of hthread system calls and abstract communication. It did not have the services version 3 has to provide the *meaningful* abstraction described in Chapter 4, in particular it did not have support for a function call stack or local variables.

The DWT implemented for version 2 has a distinct performance advantage in that the user logic instantiates its own block of memory, using the embedded dual ported BRAM. Version 2 can not use the HWTI's local memory since this is only available with version 3. When reading and writing to the instantiated memory the version 2 DWT may take full advantage of the dual ported BRAM reading or writing two values simultaneously. Version 2 also does not have to pay the overhead price for going through the HWTI user interface to read or write to local memory. The version 3 DWT uses the local memory within the HWTI. In a manner of speaking comparing these two implementations is a comparison between a "general" user logic implementation using version 3, and a "custom" user logic implementation using version 2. Both Haar DWT versions were based on the same C code. As a note, the version 2 implementation was written by Lance Feagan as a class project for EECS700 Reconfigurable Computing (Fall 2005).

Figure 6.6 compares the execution time between the two implementations, Figure 6.7 compares the complexity of the two implementations using lines ofcode.

Both hardware threads were ran against an 1 dimensional array of 5000 integers. Both versions intentionally work similarly. They each create a temporary array in their respective local memories, copy the array to transform into their local memory, perform the DWT on the temporary array (requires 2 reads and 2 writes to each element in the temporary array), and then write the temporary array back to the original array location. The difference is that the version 2 implementation uses its own instantiated local memory, while the version 3 implementation uses the local memory with the HWTI.

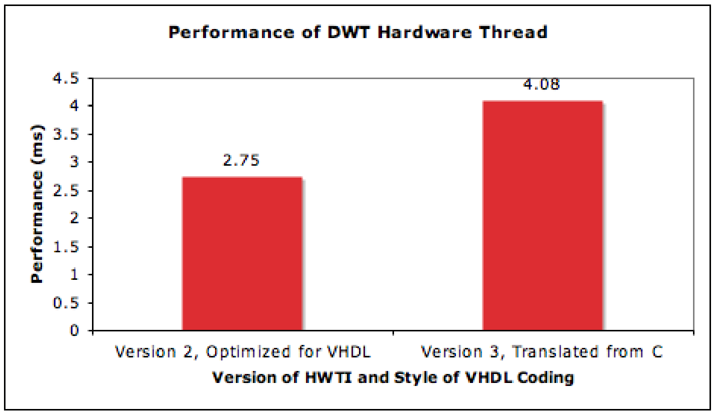


Figure 6.6. DWT performance comparison between version 2 and version 3 of HWTI using an 5000 integer array

As seen in Figure 6.6, the version 2 implementation has a 1.5x speedup over version 3's implementation. This speedup is directly related to being able to access dual ported BRAM directly instead of having to go through the HWTI's protocol layer. By instantiating its own local memory, the implementation may be faster, however it gives up the shared programming model that the HWTI (in version 3) provides. As seen in Figure 6.7, the version 2 implementation pays for this performance in its complexity, as it is over 4 times as long. Although lines of code is not the quintessential measurement of complexity, it is recognized as at least

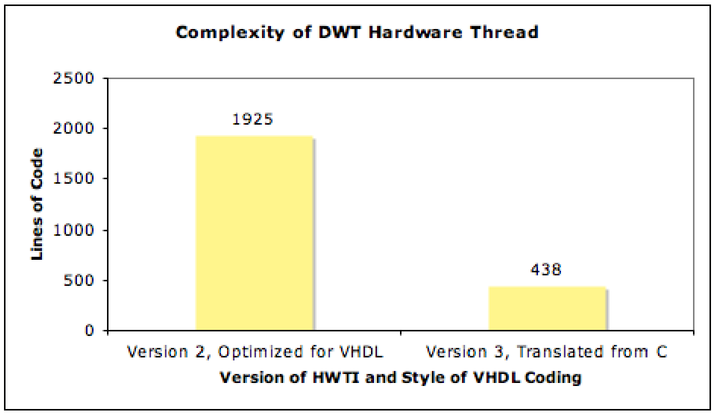


Figure 6.7. DWT complexity comparison between version 2 and version 3 of HWTI

one measurement of complexity [89]. The fact that the version 2 implementation is 4 times as long does suggest that the HWTI (version 3) has provided a useful and meaningful abstraction.

6.2.5 IDEA Encryption

The International Data Encryption Algorithm (IDEA) is a 64 bit block cipher algorithm using 128 bit keys. It was designed in 1991 by James Massey and Xuejia Lai as an improvement over the existing Proposed Encryption Standard [63]. Unlike DES, the default standard encryption algorithm of the time, which can only be efficiently implemented in a hardware circuit, IDEA was designed to be implemented efficiently in either hardware or software, making performance comparisons interesting.

The IDEA hardware thread was implemented based on C code from Bruce Schneier's Applied Cryptography [79]. Implementing the user logic this way demonstrates the HWTI's potential as a target for HLL to HDL translation. The VHDL version is functionally equivalent to the C version. At a high level, the algorithm works by encrypting 64 bit blocks in 8 rounds. In each round the data block is operated on by a series of module multiplications, xors, and additions with an expanded key. Prior to the encryption rounds the key is expanded from 128 bits to 832 bits. For each 64 bit block to encrypt, the implementation must perform 28 reads and 2 writes (of 32 bit words). The hardware implementation uses 43 states and synthesized to 2616 slices. Of the 43 states, 31 states implement the bulk of the encryption algorithm, with 7 of these states issuing `LOAD` commands to the HWTI (2 other states issuing `STORE` commands). Comparatively, the IDEA implementation is I/O intensive. We will see shortly how this property effects the performance of hardware threads, in particular the memory location of the data and key.

Figure 6.8 shows a comparison of execution time between IDEA implementations in either hardware or software. Each thread encrypted a short (16 bits) data array of length 1000. The data and key input to each thread was kept constant. For the hardware thread, the location of the data array and key varied. First the hardware thread operated on the data and key stored in traditional off-chip global memory. The second experiment shows results when the key was moved to local memory prior to the start of the thread's execution. The third result is when both the data array and key was stored in the HWTI's local memory. The two software threads results are with the software thread running with and without data cache, and the data and key in traditional off-chip memory. Each implementation encrypted the data array correctly.

These results show the importance of hardware thread's local memory. The hardware thread received a 3.8x speedup just by moving the key from global to local memory, and a 5.9x speedup moving both the key and data array. This subset

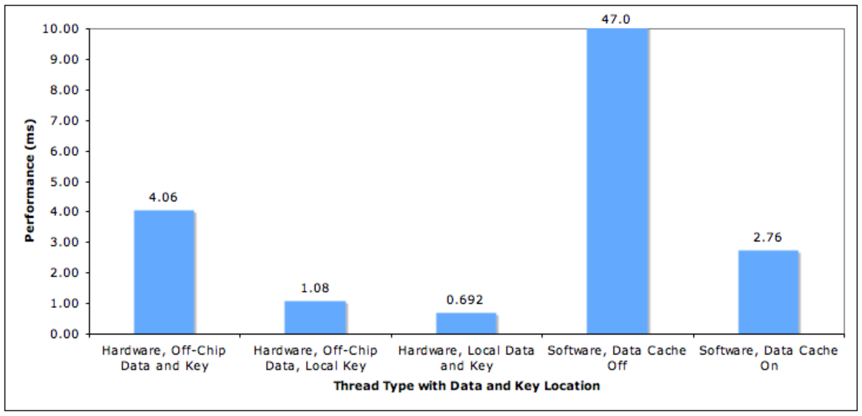


Figure 6.8. Execution times of IDEA implemented in either hardware or software, with various data locations.

of the results speaks to the memory latency problem between hardware threads and off-chip global memory. It is also interesting to note the 17.0x speedup software gets when cache is on. Similar to the results from the Quicksort algorithm, this demonstrates the impact of being able to use the HWTI's local memory to store and manage declared variables. The HWTI provides a cache-like performance for local variables without the need for a real cache and while maintaining the shared global memory programming model. Software threads do not have access to the same or equivalent mechanism creating an exaggerated slowdown.

Figure 6.9 shows the execution time of an IDEA hardware thread, with and without a software thread (without data cache) concurrently executing. In this experiment, the hardware thread (in both setups) was operating on a data array of length 1000 stored in off-chip memory. Intuitively, the hardware thread operating by it self should be faster than the system with a concurrently executing software thread since both software and hardware threads have to compete for access to the off-chip memory controller. However, the hardware thread has a small 1.01x speedup in the system with the software thread and not the expected slowdown.

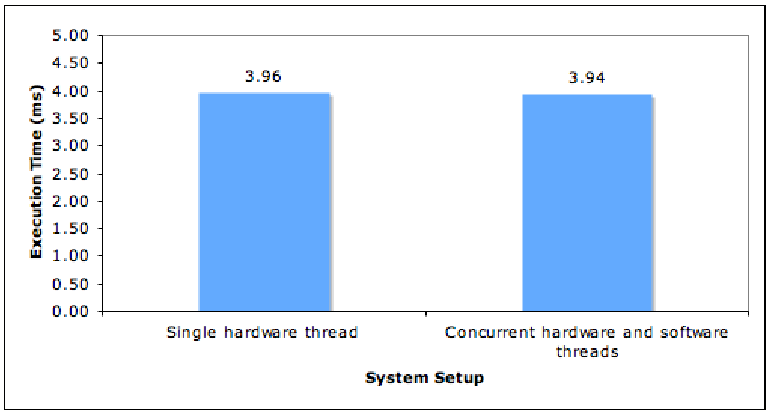


Figure 6.9. Execution times of an IDEA hardware thread running with and without a concurrently running software thread.

Conceptually the system with only the hardware thread does not have to compete for the bus or access to memory. However, an analysis of the system as a whole reveals the hardware thread does have to compete for access to the bus with the system's idle thread. The system idle thread is, in essence, `while (1) hthread_yield();`. During an `hthread_yield` the CPU makes a call to the Thread Manager to determine if it needs to context switch to another thread. As a review, the Thread Manager and hardware threads are on the OPB, while the off-chip memory controller and the CPU are on the PLB, with communication bridges in between the busses. With this system configuration, the hardware thread has to contend with the idle thread for the OPB and PLB bus. Where as the system with concurrently running threads, the hardware and software threads only have to contend for the PLB bus. This is enough of a difference to give the hardware thread a slowdown when operating pseudo-independently.

The final analysis of the IDEA implementation is in Figure 6.10. This figure shows the execution time of three dual thread systems. Again, each thread was operating on an array of 1000 shorts. By comparing the results in Figures 6.8

and 6.10 we see the benefit of true concurrency exhibited by hardware threads as opposed to the pseudo-concurrency of software threads. Not surprisingly, it takes the processors nearly twice as long to encrypt two data arrays as it does one. However, there is only a slight slowdown for two hardware threads running concurrently to encrypt two data arrays. The slowdown is due to the overhead of creating and joining threads. The benefit of true concurrency is also apparent with one hardware thread and one software thread running together. In this instance, the overall execution time is nearly identical to a single software thread. Again the slowdown is due to the overhead of creating and joining on threads. If the threads ran against larger arrays the slowdown would approach 1.0 (no slowdown).

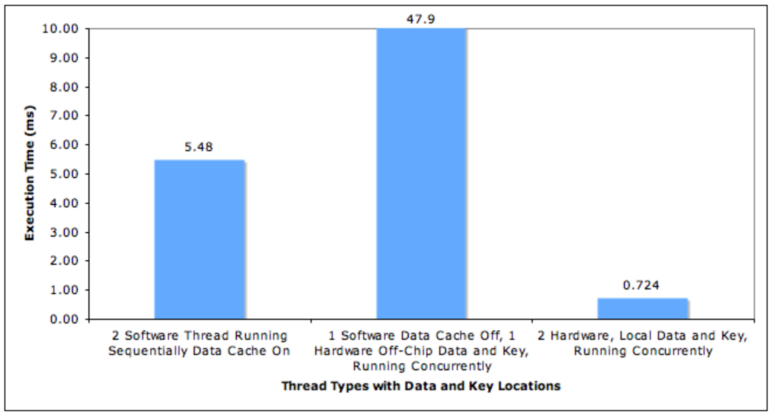


Figure 6.10. Execution times of dual IDEA threads implemented in either hardware or software, with various data locations.

As a note, it was not possible to run a dual hardware thread system with the key and data in off-chip memory. This is due to a bug in the OPB to PLB bridge that prevents two hardware cores from continually accessing the bridge. A separate experiment was conducted with two hardware threads, such that the key and data the thread operated on was located in the opposite's hardware thread's local data. In this experiment, the data block was encrypted in 2.63ms, a 3.8x slowdown over operating on its own local memory.

To complete the IDEA discussion, the hthread implementation will be compared against Vuletic's hardware thread virtual memory manager implementation given in [96, 97]. In both versions we extend the concept of a thread to include hardware cores and abstract the hardware software boundary through an system support layer. However, Vuletic uses his hardware threads as an accelerator for the CPU and the system support layer is a mix of hardware and software design. Hybridthread threads have hardware based system support layer and consequently run independent of the CPU. To maintain a consistent memory model Vuletic designed his hardware threads with virtual memory, the CPU is used to load and store values, with possible prefetching, into the hardware threads virtual memory manager. Where as hthread hardware threads independently access a shared global memory. Although not explicitly stated, Vuletic's IDEA implementation is using a systolic array type configuration to increase parallelism within the hardware thread accelerator. The hthread version sequentially encrypts each block.

Figure 6.11 compares the performance of the the two systems. Both systems were implemented on a Virtex II Pro 30. The HWTI Quad Thread number is estimated based on existing results (it is not possible to fit four IDEA hardware threads on a V2P30). Comparing results, for a single thread system, Vuletic's IDEA thread outperform hthread IDEA thread. This is due to Vuletic's optimized VHDL implementation compared to the straight forward C to VHDL version implemented in hthreads. Also important is that these results do not explicitly express the repeated CPU interrupts to feed data to the Vuletic's hardware threads. In the hthread implementation the CPU is free to perform meaningful work. The hthread system may also take advantage of task level parallelism. When multiple

IDEA hardware threads exist in the system, the average time to encrypt each block decreases nearly linearly. In short, Vuletic's threads work better as an accelerator (which is what they were designed to do), and hthread threads work better at increasing task level parallelism (which is what they were designed to do).

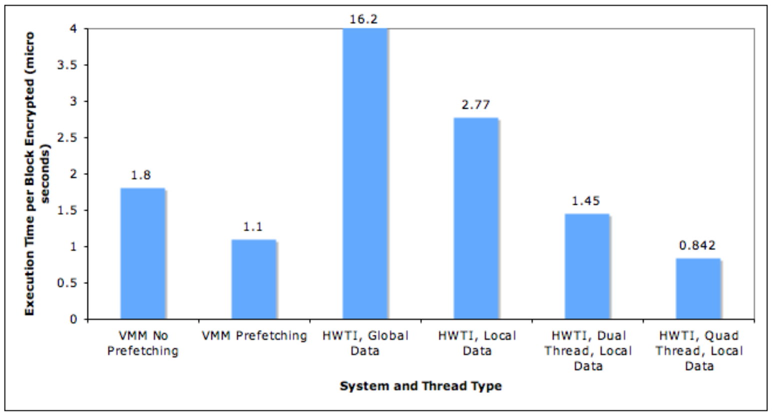


Figure 6.11. IDEA performance, based on time per encrypted block, comparing Vuletic's Virtual Memory Manager threads, and hthread threads using the HWTI.

Chapter 7

Conclusion

The research presented in this thesis demonstrates that a programming model and high-level language constructs can be used to abstract the existing hardware/software boundary that currently exists between CPU and FPGA components. This was shown by extending the concept and context of a thread to a hardware core and by providing a key subset of the hthread API to user level functionality. A standard system support layer, known as the Hardware Thread Interface was created to provide the shared memory multi-threaded programming model policies. To enable a meaningful abstraction the HWTI included support for high-level language semantics. The enabling technology was globally distributed local memory. This local memory gives hardware threads access to a fast "cache-like" memory, without the expense of a cache while maintaining the shared memory model. The HWTI leveraged this local memory to create a consistent function call model, support for local variables, and dynamic memory allocation. Migrating synchronization, communication, and high-level language semantics to hardware created an equivalent context to CPU bound threads. To analyze the hardware and software implementation, a comparison was made describing the meaning, size and performance of each support hthread system call. The HWTI support for abstract communication and synchronization was demonstrated by evaluating it against an existing POSIX test-suite ported to hthreads. The HWTI support for high level language semantics was demonstrated by adopting a number of well known algorithms as hardware threads.

7.1 Future Work

With any research project, there are lessons learned that are both positive and negative, furthermore there are far more questions that get asked than answered. The work presented in this thesis is no exception. This section seeks to explore many of these yet unfulfilled issues and questions.

7.1.1 Inter-Thread Communication

A success of this research was to abstract communication between hardware and software. The HWTI achieves abstract communication by acting as a facade for direct memory access. Within hthreads this is more specifically achieved by hiding the bus read and write protocols. By its very nature "abstracting" any layer of computation will lead to either a slowdown, or at the very best equal performance. In the case of hardware threads the HWTI adds 9 clock cycles for a single read transaction compared to a traditional hardware core as investigated in [77,78]. This is roughly a 20% penalty. Although by itself it is not a significant slowdown, it is also not desirable. Furthermore, there are a number of related system issues that effect overall performance.

First, the HWTI is not designed to leverage burst transactions, which can significantly improve communication performance. The vendor supplied IPIF [104] is capable of sustaining 16 consecutive and contiguous word reads or writes, one per clock cycle, without additional overhead compared to a single read. Although it is technically feasible to implement burst transactions within the HWTI, this raises questions as to how this would effect abstract communication. The HWTI only supports the notion of single word read and write. How then could the HWTI be extended to support up to 16 words without relying on the user to specify burst transactions, and thus breaking the abstraction?

The second issue is the fact that to achieve a shared memory system within hthreads you must turn off data caching on the CPU. This of course does not directly effect hardware thread performance but, as shown in Section 6.2.3, can significantly degrade CPU performance and consequently overall system performance. The question is then, how could the CPU keep data cache on for non-shared variable but write through to memory all data that is shared. While researching multi-core system on a chip, the authors in [75] addressed this problem by relying on static analysis techniques and programmer designations to indicate which data is shared and which is kept local. Although, static analysis techniques will have to be explored, any modifications in the programming model as the authors advocate will again break the abstraction. One possibility that hthreads should explore is keeping data cache on but flushing the cache on each mutex unlock, create, or join operation. This would make these operations more expensive, but may allow a consistent shared memory programming model.

An second possibility is to use a different programming model. The other well known abstract parallel programming model is message passing. Message passing, such as MPI [36] is widely used in multi-processor environments. Although immature there is an effort by Starbridge Systems to use MPI as an abstract programming model for FPGA systems [76]. A message passing model has the potential to allow CPUs to continue to run with data cache on, as well as allow hardware threads to continue use of local memory. The downside to message passing is overhead in communication. It is also likely that system support for message passing will have to be migrated to a hardware core, much like the Mutex Manager and Condition Variable Manager within hthreads. Until a message passing protocol can be implemented, it is unknown how this model will effect abstract communication and performance.

Lastly new interconnect networks between hardware threads need to be investigated. In [77,78] Schmidt points out that bus communication within the FPGA, especially with multiple cores, is inefficient. This effect was even seen in a simulation run of the `hthread_cond_init` stress test. In this test, additional hardware threads were added, and each thread ran until a total number of condition variables were initialized between each thread. To synchronize the total, threads had to compete for a mutex and then increment a global variable. The tests showed that with four or more threads each thread spent more time waiting to gain access to the bus than time performing all other portions of the stress test. It is natural to conclude that the more hardware threads in a system the more bus contention there will be, and the more each thread must wait for bus access. Solutions to this problem are not cheap. Two possibilities are to construct a network on a chip [62] structure for the FPGA, or create a ring network between hardware threads.

7.1.2 Leveraging Reconfigurable Computing

A concerning issue of hardware threads is the amount of time the hardware thread resources go unused. A hardware thread resources may be unused for a number of reasons: the thread has not yet been created, the thread has to block on a mutex or condition variable, the thread is waiting on the CPU to complete a RPC call, or the thread has exited. Where as processors are designed to allow context switching between tasks, the HWTI is not. There are two possibilities that could extend the HWTI to support multiple tasks.

The first, and probable easier to research, is having multiple thread's functions reside in the user logic of the hardware thread. Since the HWTI already has a `function` register that specifies for the user logic which function to execute, it is conceivable that the HWTI could "context switch" between threads in the user logic by specifying the appropriate state machine mapping for the appropriate thread in the `function` register. To accomplish this a number of changes and investigations would have to be made. The thread's start function value would have to be passed by the system to the HWTI, such that the HWTI knows which function to start on a `RUN` command. Second, the HWTI would have to be extended to support multiple function call stacks. Finally, the HWTI with perhaps the support of the Scheduler, would have to manage knowing which thread is currently running.

The second possibility is to use partial reconfiguration to swap out a blocked or exited user logic thread for a thread that is not blocked but needs to run. The difficulty with this solution is partial runtime reconfiguration is still immature. Only recently has Xilinx released a set of tools (Xilinx Processor Studio version 8.2i) that has support for partial reconfiguration of blocks [30] within the FPGA fabric instead of whole columns [109]. Unfortunately, these sets of tools are still unproven outside of the Xilinx labs. Furthermore, assuming that partial reconfiguration does work, there is still an issue of reconfiguration time. Can the performance improvement related to better use of FPGA resources overcome the reconfiguration cost?

In either scenario, hardware threads will have to be associated with a priority. Currently hardware threads are effectively given the highest priority in the system, they always have a computational unit to run on. If hardware resident threads may be context switch, for fairness a notion of priority needs to be established. This should not be difficult considering hthreads already has built in support for a priority scheduler.

Lastly, there is the very interesting issue of context switching a thread that is currently resident in hardware to run in software, or vis-a-versa. This may be advantageous in cases where the programmer wants to run the highest priority threads in hardware and the lower priority threads in software, or have a thread run in hardware during a computationally expensive segment. Context switching a thread between hardware and software will be no easy feat. Although hardware threads have an *equivalent* context in hardware, it is not a *equal* context. Issues ranging from stack value to program counter values will have to be mapped 1 to 1 between hardware and software.

7.1.3 Remote Procedural Calls

The remote procedural call model was added to hthreads as a mechanism to give hardware threads access to system and library calls too difficult or too expensive to implement within the HWTI. Although the RPC model was used to successful implement a number of library calls, including `hthread_create` and `hthread_join` there remains a potential deadlock danger with the RPC model. As described in Section 5.2.4, a hardware thread who joins on a child thread will lock the RPC mechanism until the child thread exits and the `hthread_join` call completes. Simultaneously, the child thread may be waiting on the RPC mechanism for a library call. Each thread is waiting on the other!

To address this problem the system could create a RPC thread for each hardware thread? This would prevent the described deadlock possibility. However, there remains questions as to how well this model would scale. Using this model would mean creating one software RPC thread, two mutexes, and one condition variable per hardware thread. All of these resources need to be allocated even before the system starts running.

Closely related to the RPC model is the potential to support functions that are implemented as highly efficient hardware cores. An example of this would be a fast Fourier transform (FFT), a complicated and expensive function to implement. However, an efficient hardware FFT core could be developed by hand and shared between threads. Either a hardware or software thread could call a FFT function and the system would redirect the call to the FFT core. The FFT core would be shared between all threads and access to the core could be managed through a model similar to the RPC model. Such a model would allow for efficient implementation of common or complicated functions while maintaining the existing programming model.

7.1.4 Design Issues

Although the functionality of the HWTI is complete, and as the test-suite showed, works as intended, there are a handful of low level design changes that may be made to improve performance.

The first is to have the HWTI return control to the user logic for calls and system functions that do not return a value, or always return the same value. The simplest example of this is a `STORE` command. Currently the HWTI forces the user logic to stall until the bus is complete with the write operation. This is an inefficient use of resources since the user logic could continue working while the HWTI performs the task. If the user logic makes another request to the HWTI while the HWTI is completing the previous request, the HWTI only then should have to stall the user logic. Similarly, the HWTI could return control to the user logic on system calls that always return the same value, such as `hthread_mutex_unlock` or `hthread_cond_signal`.

The second change that should occur relates to how the HWTI handles, within implementation of system calls, local reads and writes. When the user logic makes a `LOAD` or `STORE` operation, the HWTI decides if the call is local or global depending on the address. However, during an system call, if the HWTI needs to perform a `LOAD` or `STORE` it always issues a bus operation even when the address is local within the HWTI. This works because the user state machine issues the bus request on the master IPIF interface and the system state machine fulfills the request through the slave IPIF interface. Although this works, and was simple to program, it causes a slowdown for the function. Furthermore it also locks the bus until the local read or write is completed.

Finally, although the user state machine is designed as a state machine, the Xilinx synthesizer is not correctly picking it up as a state machine. This is disadvantageous since the synthesizer can more optimally allocate resources for what it believes to be a state machine, verses common logic.

7.1.5 HLL to HDL Translation

To complete the goal of designing, implementing, and testing a hardware/ software system in a known highlevel language (HLL), an efficient HLL to hardware descriptive language translator is needed. The Hybridthread Compiler (HTC) project is already underway addressing this need [91]. To compete with existing HLL to HDL translators, such as Handel C [22] or C2H [65], the HTC will need to show that its speedup using task level parallelism is in the range of their speedups using instruction level parallelism. To be successful, the HTC may have to rely on instruction level parallelism of its own within each hardware thread. However, this presents a difficulty since the HWTI implies a sequential model. Either the techniques used to develop user logic code needs to be modified, or the HWTI has to be modified to allow for more opportunities for instruction level parallelism.

7.1.6 Power Constraints

In Chapter 5 a study of the size, speed, and meaning of the hthread API was made comparing hardware and software implementations. An important study comparing power constraints was left out (although as a note a studying concerning power usage between individual functions is not practical). Considering one of the argument Mudge [69] makes for moving towards a multi-computational unit implementation of an application is energy conservation, a study of power usage of hthreads is warranted.

7.2 Concluding Remarks

This thesis demonstrated that the hardware/software boundary *can* be abstracted through a shared memory multi-threaded programming model. An important follow up question is *how well* does the the shared memory multi-threaded programming model abstract the boundary. Hardware threads, which behave very much like application specific processors, suffer from the same memory bottle necks, bus contention, and cache coherency problems for hardware tasks that traditional CPUs have, but with the limited FPGA resources. Although these problems can by *managed* through globally distributed local memory they can not be *eliminated*. The success of hybrid hardware/software systems will depend on finding meaningful and inexpensive mechanisms to move data between cores and memory layers while still maintaining an abstract communication model.

In conclusion, this researched focused on the shared memory multi-threaded programming model and is evidence that a parallel programming models may be used to abstract the CPU/FPGA boundary. Perhaps the most important lesson from this research is that the chosen programming model's middle-ware must create computational units in hardware that have an equivalent context and capabilities to traditional CPU bound tasks. This detail enables a homogenous computational model spanning hardware and software, which in turn enables an abstract programming model to target both domains.

Acknowledgment

The work in this thesis was partially sponsored by National Science Foundation EHS contract CCR-0311599.

Appendix A

System and User Interface Protocols

This section provides a detailed description of the system and user interfaces. The system interface documentation may be used by a hthreads kernel programmer when writing drivers that interact with the HWTI. The user interface documentation could be used to by a either a hardware core developer or a HLL to HDL developer (targeting the HWTI).

A block diagram of the HWTI system and user interface register set is show in Figure A.1. As mentioned already, the HWTI sits as a layer between the hthread system and a hardware thread's user logic. The system interface permits any hthread core, which includes CPU drivers, IP cores, or other hardware threads, to interact with the hardware thread regardless of the hardware thread's functionality. On the other side, the user interface acts much like a facade design patter [40], in that it allows the user logic to interact with all other system components , which include services provided by the HWTI, memory, and other Hthread cores, in a unified manner.

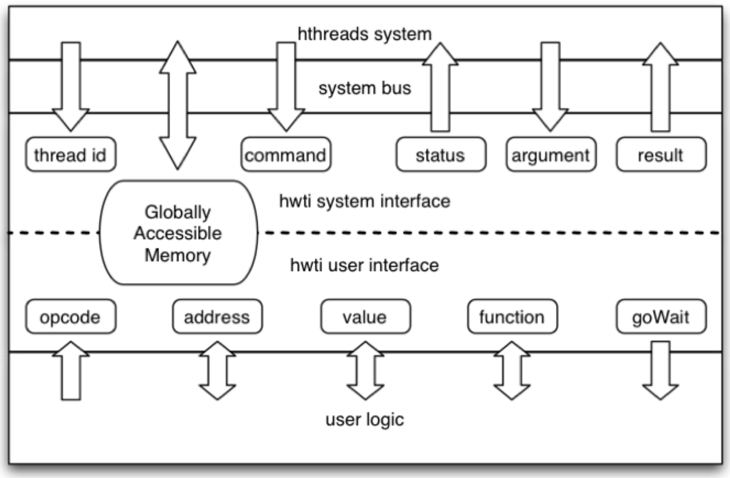


Figure A.1. HWTI register block diagram

A.1 System Interface

The system level API consists of a set of five memory mapped registers for controlling the interface, a set of debug and state information registers, and an address space for reading and writing the HWTI's local memory. The primary system interface register are `thread_id`, `command`, `status`, `argument`, and `result`. It is these five registers that incapsulate the context of a running thread in hardware. The specification and protocol of each of these registers, plus a timer register (that does not contribute to the thread's context), are in the below subsections. All registers are 32 bits wide.

Table A.1 lists all of the system interface registers, their memory offset (from the base address of the hardware thread), and a brief description.

Read and write access for the hardware thread's local memory is through the system interface, using standard bus protocols. Each hardware thread has 32 KB of local memory. The local memory address range starts at offset 0x0050 and continues to offset 0x8000. Although any thread or system core may write to

Register	Offset	Description
`thread_id`	0x0000	The thread ID of the thread
`command`	0x000C	Mechanism to allow Hthread kernel to start or reset the thread.
`status`	0x0008	Indicates if the thread is running, blocked, exited, or not used.
`argument`	0x0010	Initial argument of the thread.
`result`	0x0014	Return value for the thread.
`timer`	0x0004	Number of clock cycles between starting the thread and exiting.
`system_debug`	0x0018	Debug values set by the system state machine.
`user_debug`	0x001C	Debug values set by the user state machine.
`master_read`	0x0020	Last value read through the master interface.
`master_write`	0x0024	Last value stored through the master interface.
`stack_ptr`	0x0028	Stack pointer value.
`frame_ptr`	0x002C	Frame pointer value.
`heap_ptr`	0x0030	Heap pointer value.
`8B_mem_table`	0x0034	Allocation table for 8B units.
`32B_mem_table`	0x0038	Allocation table for 32B units.
`1024B_mem_table`	0x003C	Allocation table for 1024B units.
`unlimit_mem_table`	0x0040	Allocation table for unlimited units.
not used	0x0044	Address offset not used.
`rpc_numbers`	0x0048	Mutex and Condition Variable numbers used in RPC.
`rpc_struct`	0x004C	Address of the RPC struct.

Table A.1. HWTI User Interface Register Set

any portion of a hardware's threads local memory, it should only do so within the context of the programming model. Writing indiscriminately to a hardware thread's local memory may damage a hardware threads function call stack or other state information.

The debug and state information registers, offsets 0x0018 to 0x004C, are read only (the RPC registers being the exception). The HWTI maintains these registers as part of its internal operations. They provide a glimpse as to the internal state of the HWTI.

A.1.1 `thread_id` Register

Overview: The `thread_id` register stores the unique ID given to a hardware thread by the system. The thread ID is assigned by the system at runtime. Specifically, when a thread is created, the system asks the Thread Manager for a thread ID, the system then assigns the thread ID to this register.

The `thread_id` register is both readable and writable.

Protocol: On system start up, and after a reset, the `thread_id` is set to 0. When a write occurs to the `thread_id` register, the status changes from `NOT_USED` to `USED`. The `thread_id` may only be written to when the `status` register reads `NOT_USED`. With all other statuses, writing to this register has no effect. Bits 24 to 31, of the system bus data lines, are used to set the `thread_id` of the hardware thread. The thread ID must be non-zero, consequently the minimum thread ID is 1. The maximum thread ID is 255.

The `thread_id` register may be read from at anytime. The read operation does not have any side effects.

The `thread_id` must be set prior to the `command` register receiving a `RUN` command.

A.1.2 `command` Register

Overview: The system may issue one of two commands to the `command` register, either `RUN` or `RESET`. A `RUN` command serves two purposes. First to tell the hardware thread to start executing, second, if the hardware thread is `BLOCKED` waiting for a mutex or condition variable, to wake up and check the status of the synchronization manager core, and potentially resume running. The `RESET` command tells the hardware thread to reset all variables and registers specific to the control and execution of the thread's user logic. This applies both to the registers in the HWTI and the user logic. After issuing a `RESET`, the status is returned to `NOT_USED`. The system should issue a `RESET` command prior to creating the thread.

Protocol: A `RUN` may be issued to the HWTI only if the status register is either `USED` or `BLOCKED`. Issuing a RUN at any other time has no effect on the hardware thread.

Issuing a `RUN` while the status is `USED` changes the status to `RUNNING`. More importantly a `RUN` command results in the HWTI telling the user logic to start executing. On the user logic interface, the `goWait` register is updated to a `GO`, and the `function` register is updated to the `FUNCTION_START` value.

Issuing a `RUN` while the status is `BLOCK`, tells the HWTI to recheck the operation causing the block, either a mutex or condition variable. If successful the HWTI updates the user interface allowing the user logic to resume execution.

Issuing a `RESET` at anytime sets the `status` register to `NOT_USED`, the `thread_id` register to 0, and resets the user interface. The user logic is responsible for resetting any variables it may use. To insure the hardware thread is in an initialized state, the system should `RESET` at start up. The system must also issue a `RESET` if, after the hardware thread exits, the system wants to reuse the hardware thread component as a new thread.

The `command` register may be read from at any time, and has no side effect on the hardware thread. Reading the `command` register returns the last command the hardware thread received.

The binary values of each command are as follows:

- `RUN` (0001)
- `RESET` (0010)

Bits 28 to 31, of the system bus data lines are read to determine the value of the command.

A.1.3 `status` Register

Overview: The status register is a read only register, indicating to the system the state of the hardware thread. It is intended for debugging purposes. The possible states the hardware thread may be in are `RUNNING`, `BLOCKED`, `EXITED`, `EXITED_WITH_ERROR`, `USED`, `NOT_USED`.

Protocol: The HWTI will report each state for the following conditions. Binary values are in parenthesis.

- `NOT_USED` (0000 0000): This is the state of the hardware thread on system start up and after a `RESET` command. No other commands have been issued.
- `USED` (0000 0001): This is the state after the system assigns a value to the `thread_id` register, but before the system issues a `RUN` command.
- `RUNNING` (0000 0010): This state indicates that the user logic is executing its state machine. This state occurs when the `thread_id` register has been populated, the system issued a `RUN` command, the hardware thread is not waiting on a mutex or other blocking type of operations, and the hardware thread has not exited.

- `BLOCKED` (0000 0100): A thread will transition to a `BLOCKED` state only from a `RUNNING` state. This state occurs when the user logic is waiting for a synchronization primitive (such as a mutex). Once the lock is obtained, status transitions back to `RUNNING`.

- `EXITED` (0000 1000): The hardware thread will transition to this state after the user logic is done executing. It indicates that the value in the result register is valid (specific to the meaning of the thread).

- `EXITED_WITH_ERROR` (0010 0000): The hardware thread will transition to the state, upon command from the user logic. This state indicates that the user logic could not complete its execution as expected, due to an error (for example, divide by zero).

- `EXITED_WITH_OVERFLOW` (0100 0000): During operation of the thread, if the stack space and heap space collide, an overflow will occur. This may occur either because the function call stack grows to large, or the user requests more dynamic memory than available. The hardware thread will terminate, call exit on the Thread Manager, and set its status to `OVERFLOW`.

The argument register may be read from at any time without side effect. Writing to this register has no effect.

A.1.4 `argument` Register

Overview: Consistent with the Pthreads protocol, when a thread is created by the system, the system may pass one argument to the thread. The system uses the `argument` register to pass this argument. If used, the system must set the `argument` register after setting the `thread_id` register and prior to issuing a `RUN` command.

The meaning of the value of the `argument` register is thread specific. Generally it is an address pointer to data the thread is to operate on. Setting the `argument` register is not required. If not set, and the user logic asks for its value, the HWTI returns 0.

Protocol: The system may write to the `argument` register only if the `status` register is `USED`. This means that the system, when it wants to start the hardware thread must first issue a `RESET` command, set the `thread_id` register, set the `argument` register (if used), and then issue a `RUN` command to the `command` register.

The user logic is allowed to read the argument value in the same way it reads any passed in arguments to a user defined function. That is, the user logic issues a `POP` command, indicating the 0 indexed parameter, to the HWTI. The HWTI will respond by placing the value of the `argument` register in the `value` register on the user interface.

The `argument` register is readable at any time, and writable only when the status is `USED`.

A.1.5 `result` Register

Overview: When a thread is created as joinable, runs, and then exits, the thread has the option of passing results back to the parent thread. To pass back results to the parent, the hardware thread places the value in the `result` register. For consistency with the Pthreads interface, the result value should be a pointer, although this is not required.

Protocol: When the user logic calls the `hthread_exit` function, it may pass one argument. This argument is passed in the same manner as any other function call, that is, the user logic will `PUSH` the result to the HWTI prior to calling `hthread_exit`. Once the HWTI receives the `hthread_exit` call it will copy the value of the parameter into the `result` register.

The system may read the `result` register at any time, although, it only has meaning when the thread's status is `EXITED`. Writing to this register has no effect.

A.1.6 `timer` Register

Overview: The `timer` register reports the number of clock cycles the HWTI has been running for, if still running, or the number of clock cycles it ran for, if it has exited.

Protocol: The `timer` register begins counting when the initial `RUN` command is issued, and stops counting when the user logic calls `hthread_exit`.

The system may read from this register at anytime without side effect. Writing to this register has no effect.

A.2 User Interface

The user interface is designed to provide communication and synchronization capabilities for the user logic entity of a hardware thread. Each registers on the user interface may only be accessed by the user logic. The Hthread system has no direct access to their values.

The user interface has three sub-interfaces: memory, function, and control. The block diagram in Figure A.2 shows the HWTI with these three sub-interfaces. Each sub-interface will be explained as if it were physically separated from the other two. However the physical user interface merges the sub-interfaces into a single unit. The user interface has four registers that the HWTI uses to signal the user logic. These are the `address`, `value`, `function`, and `goWait`. Alternatively, the user logic has four registers to signal the HWTI. These are `address`, `value`, `function`, and `opcode`.

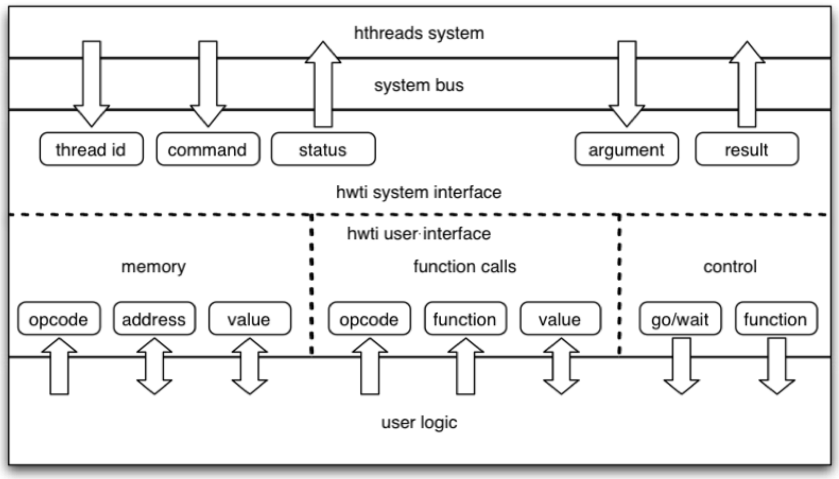


Figure A.2. HWTI sub interfaces block diagram

The `address` and `value` registers are 32 bits. The function register is 16 bits. The `opcode` register is 6 bits. The `goWait` register is 1 bit.

A.2.1 Memory Sub-Interface

Overview: The memory sub-interface is composed of three registers, `opcode`, `address`, and `value`. The `opcode` register is writable by the user logic and enables the user logic to request operations from the HWTI. When requesting an action from the HWTI, the user logic must set the `address` and `value` registers (as appropriate) in the same clock cycle. The `address` register is both readable and writable and will be used to indicate memory addresses. The `value` register is both readable and writable and will be used to indicate data values.

There are six operations associated with the memory sub-interface, listed in Table A.2. These operations are intentionally similar to high level language operations. Depending on the operation, either the `value`, `address`, or both, will be used.

Opcode	**Meaning**
`LOAD`	Read a value from memory
`STORE`	Write a value to memory
`DECLARE`	Add space on the stack for local variables
`READ`	Read a variable from the stack
`WRITE`	Write a variable to the stack
`ADDRESSOF`	Retrieve a variable's global address

Table A.2. Memory sub-interface opcodes

The user logic may only request a service to the HWTI when the `goWait` register reads `GO`. If a request is made while the `goWait` signal is a `WAIT`, the HWTI ignores the request.

When issuing an opcode, the user logic must set all appropriate user interface registers on the same clock cycles. On the clock cycle following the request the HWTI keeps the `goWait` register a `GO`, consequently the user logic must wait during this clock cycles. After the initial wait, the user logic must continue to wait until the `goWait` signal returns high. Any appropriate response will be available, in either the `address` or `value` registers, to the user logic when the `goWait` signal returns to `GO`. Lastly the `opcode` register, when the user logic issues an operation, must only be set for one clock cycle. In other words, the user logic must drive a `NOOP` to the `opcode` register following the request.

Protocol: Each of the six permitted operations, there protocols, and high level language equivalent are listed below.

- `LOAD`: Performs a word (4 bytes) load operation to memory at the address specified in the `address` register. `LOAD` may be used for accessing standard global memory, the HWTI's local memory, or any other location in the hthread address space. Once the operation is complete, the `value` register will hold the value of the address. This operation is equivalent to pointer dereferencing, for example `*ptr`.

- `STORE`: Performs a word (4 bytes) store operation to memory at the address specified in the `address` register with the value given in the `value` register. `STORE` may be used for saving information to standard global memory, the HWTI's local memory, or any other location in the hthread address space. The HWTI does not return any information to the user logic with this operation. This operation is equivalent to storing a value to a dereferenced pointer, for example `*ptr = 4`.

- `DECLARE`: Allows the user logic to request space for local variables within the HWTI's function stack. Each declared variable will have an address is accessible by the user logic and other Hthread cores. When requesting a `DECLARE` operation, the user logic specifies the number of words it wants to reserve space for in the `value` register. The HWTI does not return any information to the user logic with this operation. The user logic is allowed to issue multiple declare statements within a function. However, requesting a `DECLARE` after a PUSH request is prohibited. The allocated space is accessible by the user logic using `READ` and `WRITE` operations. The HWTI will maintain space for the declared variable until the function returns. This operation is equivalent to declaring a integer, for example `int x, y, z`.

- `READ`: Allows the user logic to read the value of an declared variable. The variable to read, is specified as an zero based index, in the `address` register. For example, if a `DECLARE` 4 was issued previously, to read the second declared variable the user logic would issue a `READ` 1. The HWTI responds by placing the value of the variable, in the `value` register. This operation is equivalent to reading a variable.

- `WRITE`: Allows the user logic to write a value to a declared variable. The variable to write, is specified as a zero based index, in the `address` register. The value to write is specified in the `value` register. For example, if a `DECLARE` 4 was issued previously, to write a 1234 to the second declared variable the user logic would issue a `READ` 1 1234. The HWTI does not return any information to the user logic with this operation. This operation is equivalent to writing to a variable, for example `x = 1234`.

- `ADDRESSOF`: Allows the user logic to request the address of a declared variable. The variable to learn the address of is specified, as a zero based index, in the `address` register. For example, if a `DECLARE` 4 was issued previously, to learn the address of the second declared variable the user logic would issue a `ADDRESSOF` 1. The HWTI responds by returning the address of the variable in the `address` register. This operation is equivalent to the address of operator, for example `&x`.

A.2.2 Function Sub-Interface

Overview: The function call sub-interface is composed of three registers: `opcode`, `function`, and `value`. The `opcode` register, allows the user logic to request operations from the HWTI. There are four non-noop operations, listed in Table A.3. By passing parameters via the stack, this enables a consistent function call protocol regardless of the number of parameters.

Opcode	**Meaning**
`POP`	Read a function parameter from the stack
`PUSH`	Push a function parameter to the stack
`CALL`	Call a function
`RETURN`	Return from a user defined function

Table A.3. Function sub-interface opcodes

The `function` register tells the HWTI which function the user logic wants to call. The HWTI reserves a number of values, x8000 to x8FFF, for system calls it supports, and x9000 to xFFFF, for future library calls. The supported system calls and their defined opcodes are listed in Table A.4. The value x0000 is a signal to the user logic to reset itself, x0001 signals the user logic to execute any state it wants, and x0002 signals the user logic to execute its start function. Values x0003 to x7FFF are reserved for user logic defined functions and states. The user logic defined function values are analogous to starting addresses for functions in software. In hardware, these values may be implemented as states in a state machine. The HWTI will pass control to these states, through the control sub-interface.

A brief pseudo-code example of how the function sub-interface for calling mutex lock is given in Figure A.3. In state x0101, the user logic pushes the address of the mutex onto the stack, in this case the mutex is at x0023 8F20. In state x0102, the user logic calls `hthread_mutex_lock` (the Hthread function codes are listed in the Control Sub-Interface section), while specifying that once the mutex lock function is completed, the HWTI should return control to the user logic in state x0103.

As is the case in the memory sub-interface, when issuing an opcode, the user

Function Call	**Library**	**Function code**
hthread_attr_init()	hthread.h	0x8000
hthread_attr_destroy()	hthread.h	0x8001
hthread_create()	hthread.h	0x8010
hthread_join()	hthread.h	0x8012
hthread_self()	hthread.h	0x8013
hthread_yield()	hthread.h	0x8014
hthread_equal()	hthread.h	0x8015
hthread_exit()	hthread.h	0x8016
hthread_exit_error()	variation of hthread_exit	0x8020
hthread_mutexattr_init()	hthread.h	0x8021
hthread_mutexattr_destroy()	hthread.h	0x8022
hthread_mutexattr_setnum()	hthread.h	0x8023
hthread_mutexattr_getnum()	hthread.h	0x8024
hthread_mutexattr_init()	hthread.h	0x8030
hthread_mutex_destroy()	hthread.h	0x8031
hthread_mutex_lock()	hthread.h	0x8032
hthread_mutex_unlock()	hthread.h	0x8033
hthread_mutex_trylock()	hthread.h	0x8034
hthread_condattr_init()	hthread.h	0x8040
hthread_condattr_destroy()	hthread.h	0x8041
hthread_condattr_setnum()	hthread.h	0x8042
hthread_condattr_getnum()	hthread.h	0x8043
hthread_cond_init()	hthread.h	0x8050
hthread_cond_destroy()	hthread.h	0x8051
hthread_cond_signal()	hthread.h	0x8052
hthread_cond_broadcast()	hthread.h	0x8053
hthread_cond_wait()	hthread.h	0x8054
hthread_malloc()	stdlib.h	0xA000
hthread_calloc()	stdlib.h	0xA001
hthread_free()	stdlib.h	0xA002
hthread_memcpy()	string.h	0xA100

Table A.4. HWTI's supported library calls

State	Operation
x0101	push x0023 8F20
x0102	call x8032, x0103
x0103	...

Figure A.3. Pseudo-code example for `mutex_lock( &mutex )`

logic must wait the clock cycle after the request is made to the HWTI. On the clock cycle following the mandatory wait the user logic must wait only if the `goWait` signal is a `WAIT`. The user logic must continue to wait until the `goWait` signal returns to `GO`. When this occurs, any appropriate response will be available to the user logic.

Protocol: Each of the four permitted operations and there protocols are listed below.

- `PUSH`: Prior to calling a function, the user logic may pass parameters to the soon to be called function using the `PUSH` operation. Each `PUSH` places the parameter specified in the `value` register onto the HWTI's stack. The user logic may push as many parameters as needed for a function. Each parameter is 32 bits. The HWTI does not return any value to the user logic for the `PUSH` operation. When pushing parameters onto the stack, the user logic should push the last parameter first. For example, if the user logic is calling `foo(a, b, c)`, the user logic should push the value of `c` first, then `b`, and finally `a`.

- `POP`: Once the HWTI transfers control to a new set of states in the user logic, representing an user defined called function, the user logic may use the `POP` operation to retrieve the values of the parameters. To allow the function to read any of the parameters any time prior to a `RETURN`, the user logic specifies the parameter it wants to read in the `value` register. For example, if the function `foo(a, b, c)` was called, to read parameter `a`, the user logic would request `POP 0`, to read parameter `b`, the user logic would request a `POP 1`, and so on. The HWTI responds with the value of the parameter in the `value` register.

- `CALL`: After all parameters are pushed to the HWTI, the user logic may use `CALL` to invoke a function. Once the called function finishes, the HWTI returns control to the user logic where the call was made. The `CALL` operation may be used for either a system or library function, or transfer control to a function defined locally within the user logic. When using the `CALL` operation, the user logic must specify the function it wants to invoke, and the state to return control to after the call is complete. The function to invoke is specified in the `function` register. The return state is specified in bits 16 to 31 of the `value` register (bits 0 through 15 are ignored by the HWTI). When the called function issues a `RETURN`, the HWTI returns control to the specified return state, with any return value set in the `value` register.

- `RETURN`: To return from a user defined function, the user logic issues a `RETURN` operation. The user logic may pass back one 32 bit value to the caller function. The value to return is specified in the `value` register. Any declared variables (from the memory sub-interface) are deallocated from the function stack on a `RETURN`. The caller function may read the return value in the `value` register.

A.2.3 Control Sub-Interface

Overview: The control sub-interface details how the HWTI manages the execution and delays of the user logic entity. This sub-interface has two registers, `goWait` and `function`.

Protocol: The `goWait` register tells the user logic to either continue execution or wait. The HWTI halts the user logic's execution to give it time to fulfill a request. For example, during a load to global memory, the HWTI may require up to 50 clock cycles to finish the request. It is necessarily for the user logic to stop execution until the load is complete and the HWTI can report the value back to the user logic. The single bit `goWait` register creates a simple hand shaking protocol between the HWTI and user logic. The user logic may only request a service from the HWTI when the `goWait` register reads '1' (a `GO`), and must halt execution when it reads '0' (a `WAIT`). The user logic, must also halt execution on the same clock cycle a request to the memory or function sub-interface is made.

To be more specific, a `WAIT` signal does not mean the user logic has to halt execution, but rather it means it must not request any new services from the HWTI. The user logic is free to continue processing, however if it makes a request while `goWait` is `WAIT`, the HWTI will ignore the request.

The `function` register enables the HWTI to tell the user logic which logic to execute, or which state (for a state machine implementation) to be in. It is analogous to the program counter in a CPU. Intentionally like the `function` register, in the function call sub-interface, certain values have reserved meaning. A value of x0000 tells the user logic to reset it self, it will be the default value on power up and HWTI reset. A x0001, tells the user logic to control its own execution. A x0002, tells the user logic to execute its first instruction. Values x80000 to xFFFF will not be used (since they were reserved for system call or library functions implemented outside the user logic). Values x0003 to x7FFF will tell the user logic to execute specific states or logic within its implementation.

Appendix B

State Machine Implementations

This section describes the HWTI implementation. Appendix A gave the user and system interface protocols without regard as to how it is implemented, in effect this was the “black box” requirements. This chapter describes the content and internal design of the HWTI. Source code for the HWTI may be found from within the hthread subversion repository [61].

Loosely speaking, the HWTI is implemented as two state machines, the system state machine, and the user state machine. Each state machine is written as a two-process finite state machines as described in [108], using behavioral VHDL. The system state machine controls the system interface. It monitors the registers attached to the bus and address range into the HWTI’s local memory. The user state machine controls the three sub-interfaces forming the user interface, maintains the function call stack, implements the system and library calls, and performs master bus transactions for addresses outside of the HWTI’s address range. A block and logical diagram of the two state machine implementation is in Figure B.1.

Figure B.1 also demonstrates a number of features important to discuss as to

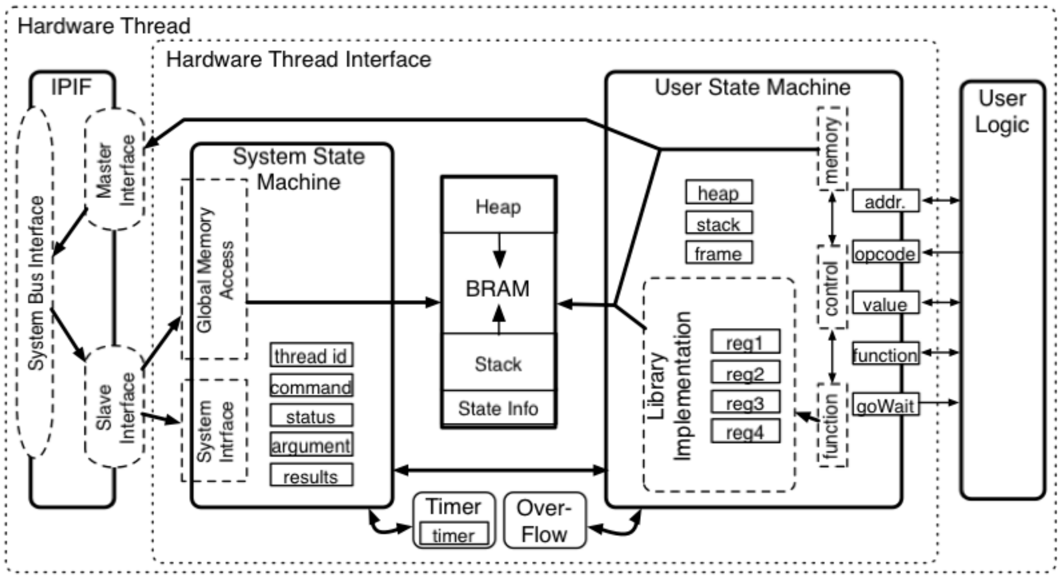


Figure B.1. Block diagram of HWTI implementation

its implementation.

- A hardware thread is made up of three entities: The IPIF [104] that provides the connection to the OPB bus (not shown in the diagram), the HWTI acting as the abstraction layer, and the user logic entity.
- The HWTI is further broken down into three primary entities. The system state machine, user state machine, and a dual ported BRAM.
- Each state machine has access to one of the ports on the BRAM entity. Not shown in the diagram, the BRAM blocks are configured to be an array of 32 bit words.
- The HWTI is implemented with four additional minor processes (two of which are not shown). The first process maintains the `timer` register for the system interface. The second process watches the stack and heap pointer for

overflow detection. The two processes not shown work together for timeout suppression of the OPB bus.

- Access into the HWTI, from outside the hardware thread, is through the vendor supplied IPIF which in turn connects to the system interface. There is no direct access to the user logic.

- Access to the HWTI's local memory, from the system bus, is through the system state machine. This is note worthy since the system state machine represents an overhead (although small) for accessing the local memory.

- The system state machine maintains five primary registers, `thread_id`, `command`, `status`, `argument`, and `results`. The sixth register, in the system interface, `timer` is maintained by a separate process.

- The low valued addresses, in the local memory, are reserved for debug and state information. This information, not shown, includes the stack pointer, frame pointer, heap pointer, debug values, and tables to maintain dynamically allocated memory, and remote procedural call data structures.

- The HWTI's stack begins at the lowest address possible, right above the state information, and grows up. The heap starts at the top of the address range and grows down.

- The user state machine maintains 12 registers. Three of the registers, `heap`, `stack`, and `frame`, are the heap, stack, and frame pointers. There are four general use registers, `reg1` through `reg4`, used by library implementation states, and five registers, `address`, `opcode`, `value`, `function`, and `goWait`, comprising the user interface.

- The three user sub-interfaces, function, control, and memory are only virtual interfaces and their implementation share the five physical registers making up the user interface.
- A significant portion of the user state machine is the implementation of the supported library functions.
- The HWTI implementation may be modified in the future, provided that the implementation does not alter the HWTI protocols.

B.1 System State Machine

The system state machine has three responsibilities. First enforce the system interface protocol. It does this by monitoring the bus for commands (writes to the system interface registers), and checking the hardware thread state, prior to acting on the command. Second, provide read and write access into the hardware thread's local memory for other hthread cores. Third, respond to the user state machine for updates to the threads status.

The first two responsibilities are achieved by monitoring the system bus for read and write requests, through the IPIF slave interface. Each hardware thread has the same memory mapped offsets, including the size of the local memory.

The third responsibility, responding to status change requests from the user state machine, is done through an internal register named `system_request`. When the user state machine has a status change it wants to make to the hardware thread, it does so via the `system_request` register. For example, when the thread exits (the `hthread_exit` implementation is done in the user state machine), the user state machine sets the values of the `system_request` register to

`CHANGE_STATUS_TO_EXIT`. The system state machine, on the next available clock cycle reads this request and updates the `status` register appropriately. The user state machine maintains (continuing the example) the `CHANGE_STATUS_TO_EXIT` value until it sees the value of the `status` register updated. It is through the `system_request` register that the user state machine may communicate with the system state machine. Communication in the opposite direction, from the system state machine to the user state machine, is either from the user state machine reading the register values on the system interface (as in the example given here), or from exchanging information on the stack, which both state machines have access to through the dual ported BRAM.

On power up, or after a `RESET` command, the system state machine initializes the system interface registers. This is handled within one clock cycles. After initialization, the system state machine, spends most of its time in the `IDLE` state. As depicted in the pseudo-state machine diagram in Figure B.2, during each clock cycles the `IDLE` state checks for one of three actions. It first checks to see if there is a write request coming off of the bus. If there is a write request, it checks the address and for either a system interface register or for access to the local memory. Second it checks to see if there is a read request coming off of the bus. Here to, it checks the address for either a system interface register (including the implementation specific registers), or for access to the local memory. If there is not any activity on the bus, it checks the `system_request` register for status updates from the user state machine. It is important to note that the system state machine only responds to requests from the user state machines if there is not a read or write request coming from the bus. This may delay the user state machine's request from being completed.

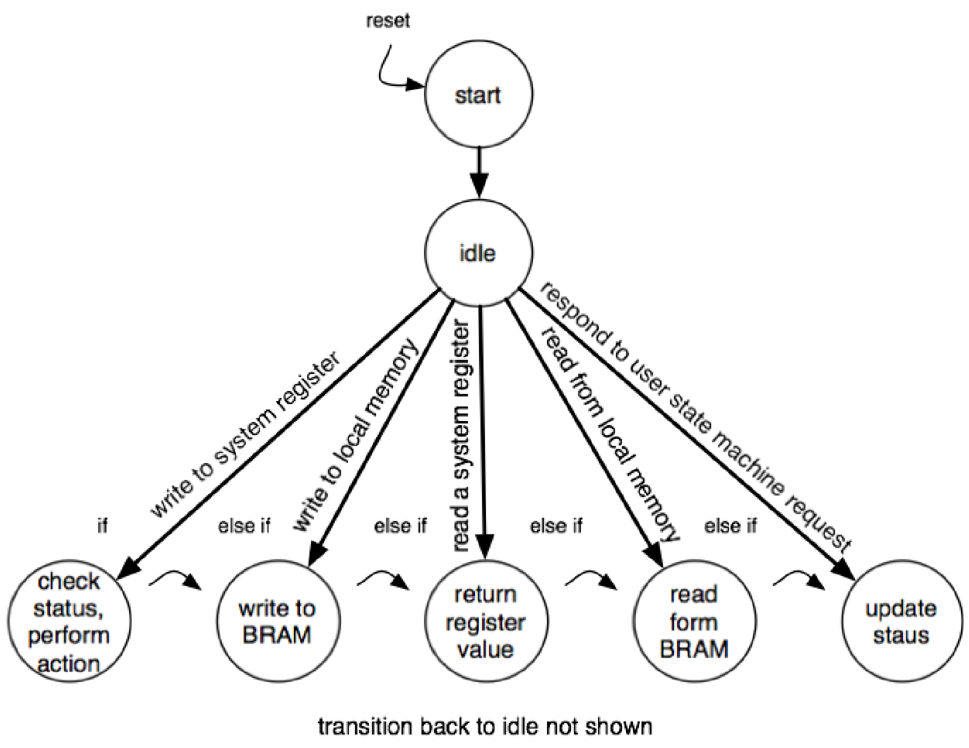


Figure B.2. Block diagram of system state machine

The system state machine is physically implemented with 8 states.

B.2 User State Machine

The user state machine, shown in Figure B.4 is significantly larger and more complicated than the system state machine. It is responsible for maintaining and enforcing the user interface protocol. Second, to provide access to both global memory and local memory to the user logic (through the user interface). Third, maintain and manage the function stack. Fourth, maintain and manage the heap, including dynamic memory allocation. Fifth, implement the supported system and library calls. And sixth, implement remote procedural calls.

Like the system state machine, the user state machines has an initialization phase, shown in Figure B.3, and a running phase, shown in Figure B.4. The user state machine enters initialization either on power up, or after a `RESET` command. Unlike the system state machine though, the user state machine initialization takes multiple clock cycle to complete. In fact, initialization steps occur synchronous to the hardware thread's state changes. During initialization, internal registers are reset, the stack and heap initialized, and the user logic is given its own reset command. The initialization of the stack and heap, which requires multiple writes to BRAM, is the primary reason why the user state machine requires multiple clock cycles complete.

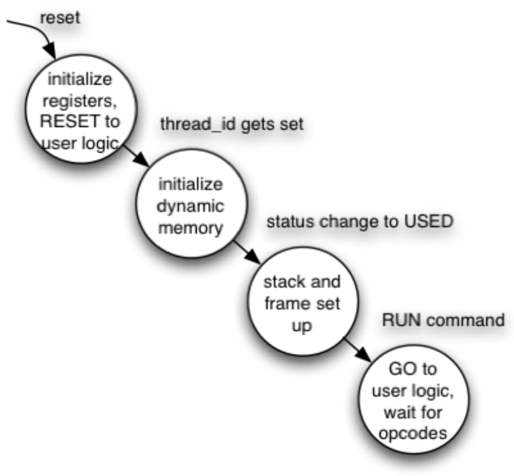

reset
initialize registers, RESET to user logic
thread_id gets set
initialize dynamic memory
status change to USED
stack and frame set up
RUN command
GO to user logic, wait for opcodes

Figure B.3. State diagram of user state machine initialization sequence

Once initialized the user state machine waits for opcode requests from the user logic. When an opcode is received, depicted in Figure B.4, the the user state machine transitions to the corresponding operation while simultaneously setting the `goWait` register to `WAIT`. There are 10 opcodes the user logic may issue.

- `LOAD` and `STORE` operations are implemented by inspecting the address passed by the user logic. If the address is local, within the threads address range,

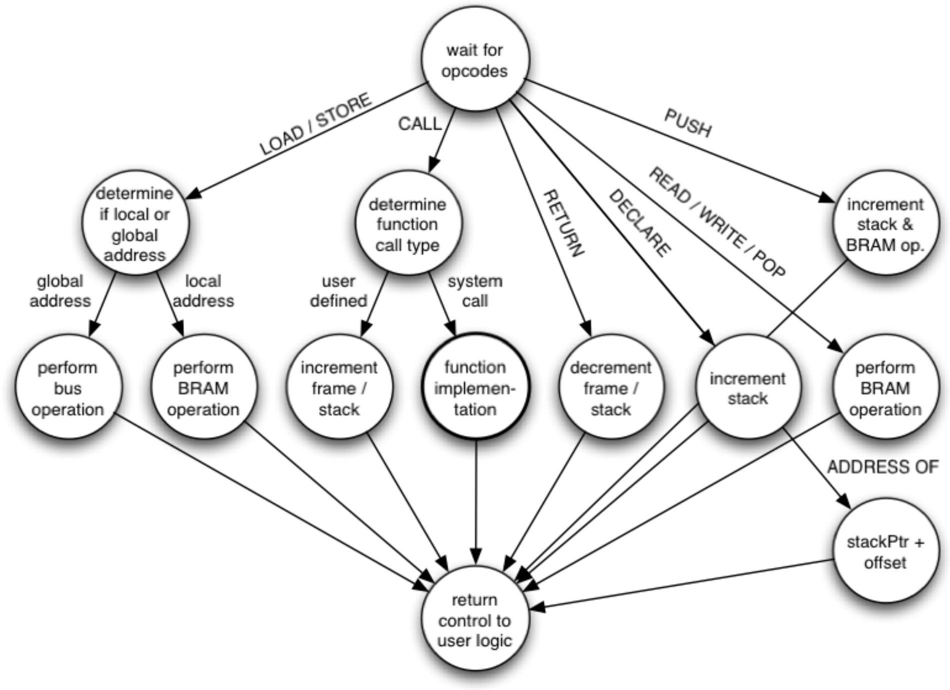


Figure B.4. Block diagram of user state machine implementation

the user state machine performs a BRAM read or write to the corresponding BRAM location. If the address is global, outside the thread's address range, the user state machine performs a bus master read or write. Since Hthreads does not use virtual memory, the address issued to the bus is exactly the address passed by the user logic. The user state machine returns control to the user logic when either the bus or BRAM operation is complete.

- `CALL` operations are performed by decoding the `function` register. If the specified value is known, that is the function being called is either a supported system or library call, the user state machine transitions to the appropriate function start state, implemented internal to the user state machine. Implementations of each of the supported system and library calls are case specific. If the `function` value is not known, the user state machine assumes that it is a user defined function. In which case, it saves the number of parameters passed (in previous `PUSH` operations) onto the stack, saves the frame pointer on the stack, saves the user logic return state to the stack, and finally increments the stack and frame pointer. As a note, the implementation of the supported system and library calls were covered in section 5.2.

- `RETURN` operations are implemented by restoring the stack and frame pointer, that were previously stored on the stack, and by driving the `function` register with the return state previously stored on the stack.

- `DECLARE` operations are implemented simply by incrementing the stack pointer a by the number of variables the user logic wants to reserve space for.

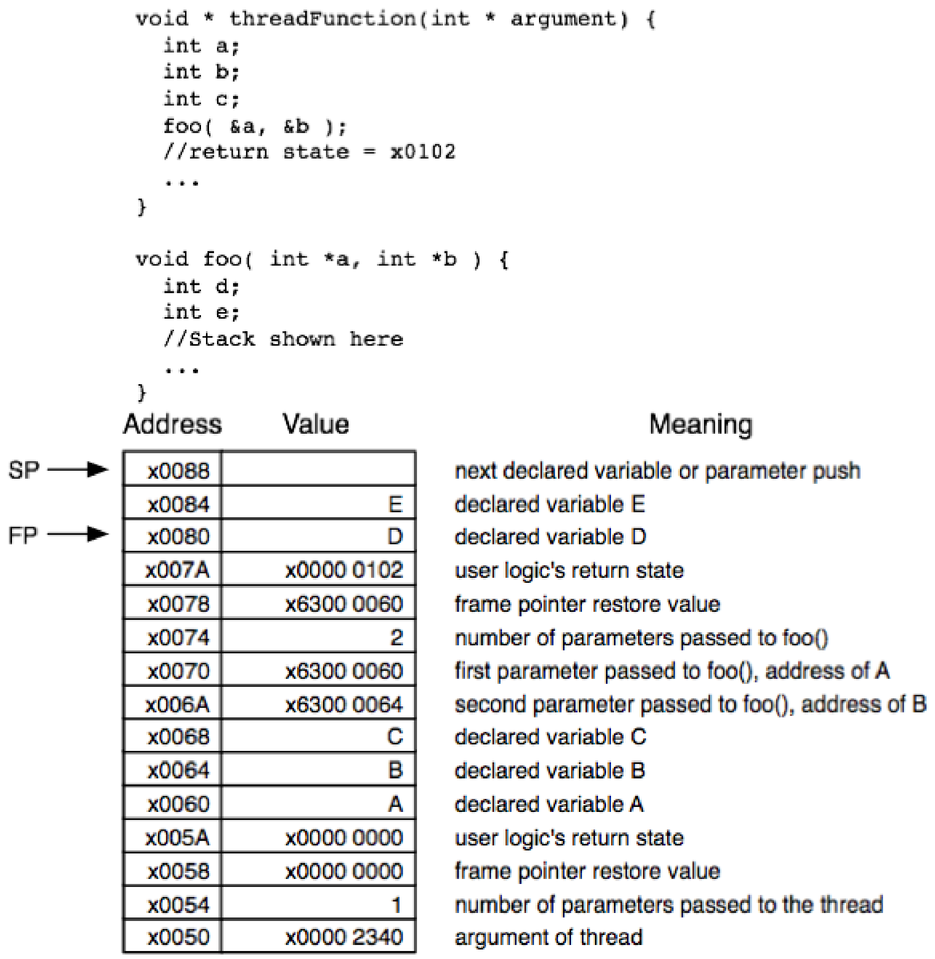
- `READ` and `WRITE` operations are implemented by performing BRAM operations (on the stack). The variable number the user logic passes via the `address` register is the frame pointer offset. The BRAM location to read or write to is the frame pointer plus the passed in variable number. The user state machine assumes that the user logic previously declared space for the variable.

- `PUSH` operations are implemented by performing a write BRAM operation, storing the passed in value to the current stack pointer address. Both the stack pointer, and a parameter count variable are incremented.

- `POP` operations are implemented very similar to READ operations. The difference is that the parameter number passed in the `address` register is the negative offset, minus 2, from the frame pointer.

- ADDRESSOF operations are implemented by returning the sum of the offset value in the address register, with the frame pointer and the base address of the thread.

To help demonstrate the function call stack implementation, consider the pseudo code, and stack representation, given in Figure B.5. A number of important items may be learned from this example.



```
void * threadFunction(int * argument) {
  int a;
  int b;
  int c;
  foo( &a, &b );
  //return state = x0102
  ...
}

void foo( int *a, int *b ) {
  int d;
  int e;
  //Stack shown here
  ...
}
```

	Address	Value	Meaning
SP →	x0088		next declared variable or parameter push
	x0084	E	declared variable E
FP →	x0080	D	declared variable D
	x007A	x0000 0102	user logic's return state
	x0078	x6300 0060	frame pointer restore value
	x0074	2	number of parameters passed to foo()
	x0070	x6300 0060	first parameter passed to foo(), address of A
	x006A	x6300 0064	second parameter passed to foo(), address of B
	x0068	C	declared variable C
	x0064	B	declared variable B
	x0060	A	declared variable A
	x005A	x0000 0000	user logic's return state
	x0058	x0000 0000	frame pointer restore value
	x0054	1	number of parameters passed to the thread
	x0050	x0000 2340	argument of thread

Figure B.5. HWTI's function call stack

- The HWTI sets up the stack for the primary thread function the same as it would for any user defined function. That is to say, when a hardware thread starts the HWTI automatically stores a value for the passed in argument, number of parameters, frame pointer, and return state. Being consistent in this manner is helpful. As an example, when the user logic issues a `POP` operation, because the user state machine does not have to explicitly track which function the user logic is currently working on. The behavior of `POP`, and other opcodes, is identical regardless of which state the user logic is in. The only operational difference is on a `RETURN` call. In this case the user state machine checks the value of the user logic's return state. A 0x0000 value indicates the user logic is returning from the thread function. Since this is the same as explicitly calling `hthread_exit`, the user state machine, implicitly calls the `hthread_exit` implementation.

- The stack pointer always points to the next available address. The frame pointer always points to the first address location used (or may be used) for the current function. Also, the stack pointer's address is always equal to or greater than the frame pointer, since the stack grows up.

- When calling a function, with multiple parameters, the parameters are pushed onto the stack in reverse order. This allows an easier `POP` implementation.

- The number of parameters in a function call is pushed onto the stack. This value is used to make sure the user logic does not issue a `POP` outside the number of parameters, and to restore the stack pointer on a `RETURN`. An alternative would be to push the value of the stack pointer. This was an implementation decision. The only advantage pushing the number of parameters instead of the stack pointer value, is that the HWTI could support (although it currently does not) variable length argument lists, such as is done with `printf`.

The complete user state machine is implemented in 147 states.

B.2.1 Dynamic Memory Allocation Mechanism

To implement dynamic memory allocation, the HWTI pre-allocates space at the top of its local memory address range on initialization. The HWTI preallocates 32 8B segments, 8 32B segments, and 2 1024B segments. Pre-allocation is important, as it prevents the HWTI from needing a de-fragmentation routine. However pre-allocation may also waste space, if for example, the user logic request 64B, the HWTI would have to return a 1024B segment (although the user would not know it is a 1024B segment). The HWTI tracks which segments have been allocated using 3 BRAM locations, 1 each for the different segment sizes. These BRAM locations are seen as memory mapped registers to the system interface.

When the user logic calls `malloc`, the HWTI determines if the amount of memory requested is less than 8B, 32B, or 1024B. If so, the HWTI pulls the appropriate segment allocation table from BRAM, and searches for the first segment not yet used. If the HWTI can not find space, it repeats the search on the next largest segment allocation table. If after the 1024B allocation table the HWTI still can not find space, the HWTI may issue 1 memory segment of up to 30,128B.

The variable size segment is maintained by the `heap_ptr`. At start up the `heap_ptr` points to offset 0x7600, indicating the space that has been pre-allocated. If a segment of memory larger than 1024B is needed (or all segments are allocated), the HWTI decrements the `heap_ptr` the requested amount. If the `heap_ptr` is already in use, the HWTI returns a NULL pointer to the user logic. Returning a NULL pointer is consistent with the stdlib.h implementation when memory space has been exhausted.

`calloc` is easily implemented by multiplying the count and size parameters passed to the HWTI and then transitioning to the `malloc` sets of states.

`free` is implemented using a large select statement. One case statement exists for each of the possible pre-allocated segments pointers. The case statement returns the BRAM location where the allocation table is stored, and the index into the allocation table. The HWTI completes the `free` process by updating the allocation table.

B.2.2 Remote Procedural Call Mechanism

Figure B.6 shows the pseudo code for the RPC mechanism. The RPC implementation does not reuse the mutex lock, unlock, or condition variable signaling functions already implemented within the HWTI. These pre-existing functions are designed to return to the user when complete, reusing them would require a substantial redesign of the HWTI. Instead, the HWTI re-implements the mutex and condition variable functionality for the RPC code in their own sets of states.

The pseudo code listed in Figure B.6 uses four global variables, the mutexes `rpc_mutex`, and `rpc_signal`, the condition variable `rpc_signal_mutex`, and the structure `rpc`. Since the HWTI does not support global variables, the addresses of these structures must be passed to the HWTI when the system starts. To simplify the implementation, the HWTI only needs to know the mutex numbers for `rpc_mutex`, and `rpc_signal`, and the condition variable numbers for `rpc_signal_mutex`. This is because when the HWTI communicates with the Mu-

```
int rpc_hardware( int opcode, int args[5] ) {
  // Lock the RPC mechanism
  hthread_mutex_lock( rpc_mutex );

  // Lock the RPC signal
  hthread_mutex_lock( rpc_signal_mutex );

  // Fill in the RPC arguments
  rpc->opcode = opcode;
  for( i = 0; i < 5; i++ ) rpc->args[i] = args[i];

  // Send the RPC
  hthread_cond_signal( rpc_signal );

  // Wait for the RPC to complete
  hthread_cond_wait( rpc_signal, rpc_signal_mutex );

  // Read the result
  result = rpc->result;

  // Unlock the RPC mechanism
  hthread_mutex_unlock( rpc_signal_mutex );
  hthread_mutex_unlock( rpc_mutex );

  // Return the result
  return result;
}
```

Figure B.6. HWTI's remote procedural call pseudo code.

tex Manager or Condition Variable Manager it only sends the respective index number. To pass this information to the HWTI, the HWTI two system state machine registers set aside, `rpc_numbers` and `rpc_struct`. The value of `rpc_numbers` is encoded with the three mutex and condition variable number. The value of `rpc_struct` is the address of `rpc`.

The HWTI instantiates one set of states for all functions utilizing the RPC. The implementation automatically handles the varying number of parameters that could be passed for each function using the parameter count variable that gets increment for each `PUSH` operation. The RPC opcode is intentionally the `function` opcode passed to the HWTI.

For completeness, the pseudo code the RPC system thread implements, showing implementations for `hthread_create` and `hthread_join`, is in Figure B.7. The main thread is responsible for creating the system RPC thread at start up.

```
void* rpc_software( void *arg ) {
  // Lock the RPC signal
  hthread_mutex_lock( rpc_signal_mutex );

  while( 1 ) {
    // Wait for an RPC message
    hthread_cond_wait( rpc_signal, rpc_signal_mutex );

    // Perform the requested function
    switch( rpc->opcode ) {
      case CREATE:
        rpc->result = hthread_create(
        rpc->args[0],
        rpc->args[1],
        rpc->args[2],
        rpc->args[3] );
      break;

      case JOIN:
        rpc->result = hthread_join(
        rpc->args[0],
        rpc->args[1] );
      break;
    }

    // Signal that the RPC is done
    hthread_cond_signal( rpc_signal );
  }

  //Never get here, but to be complete...
  hthread_mutex_unlock( rpc_lock );
}
```

Figure B.7. Software thread remote procedural call pseudo code.

References

[1] J. Agron, D. Andrews, M. Finley, E. Komp, and W. Peck. Fpga implementation of a priority scheduler module. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium, Works in Progress Session (RTSS WIP)*, Lisbon, Portugal, December 2004.

[2] 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.

[3] G. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin Cummings, 1994.

[4] Altera. Altera. http://www.altera.com/.

[5] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the AFIS Spring Joint Computer Conference*, volume 30, pages 483–485, Atlantic City, NJ, 1967.

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

[7] D. Andrews, D. Neihaus, and R. Jidin. Implementing the Thread Programming Model on Hybrid FPGA/CPU Computational Components . In *1st Workshop on*

Embedded Processor Architectures, Madrid, Spain, Feb. 2004.

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

[9] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden. Programming models for hybrid FPGA/CPU computational components: A missing link. *IEEE Micro*, 24(4):42–53, July/August 2004.

[10] 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.

[11] P. Athanas and H. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, 26:11–18, 1993.

[12] Atmel. Atmel. http://www.atmel.com/.

[13] K. Bondalapati, P. C. Diniz, P. Duncan, J. Granacki, M. W. Hall, R. Jain, and H. Ziegler. Defacto: A design environment for adaptive computing technology. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 570–578, London, UK, 1999. Springer-Verlag.

[14] A. Bouchhima, X. Chen, F. Petrot, W. O. Cesario, and A. A. Jerraya. A unified hw/sw interface model to remove discontinuities between hw and sw design. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 159–163, New York, NY, USA, 2005. ACM Press.

[15] Brigham Young University. Jhdl: Fpga cad tools. http://www.jhdl.org.

[16] J. C. Browne. Parallel Architectures for Computer Systems. *IEEE Computer*, 37 no 5:83–87, 1984.

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

[18] B. Buyukkurt, Z. Guo, and W. A. Najjar. Impact of Loop Unrolling on Area, Throughput and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs. In *International Workshop on Applied Reconfigurable Computing*, March 2006.

[19] T. Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures.* PhD thesis, University of California Berkeley, 2002.

[20] T. Callahan, J. R. Hauser, and J. Wawrzynek. The GARP Architecture and C Compiler. *IEEE Computer*, 33(4):62–69, 2000.

[21] C. L. Cathey, J. D. Bakos, and D. A. Buell. A reconfigurable distributed computing fabric exploiting multilevel parallelism. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 121–130, Washington, DC, USA, 2006. IEEE Computer Society.

[22] celoxica. Handelc. www.celoxica.com.

[23] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *International Symposium on Computer Architecture*, Boston, MA, USA, June 2006.

[24] S. Chappell and C. Sullivan. Handel-C for co-processing and co-design of Field Programmable System on Chip. In *Jornadas de Computacion Reconfigurable y Aplicaciones*, September 2002.

[25] D. Computers. Product Information. http://www.drccomputer.com/pages/-products.html.

[26] A. Corporation. Flash FPGAs in the Value-Based Market White Paper. http://www.actel.com/documents/ValueFPGA_WP.pdf. Technical Report, January 2005.

[27] H. Deitel and P. Deitel. *C Sharp for Programmers Second Edition.* Prentice Hall, Inc, Upper Saddle River, New Jersey, 2006.

[28] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for content filtering. In *Symposium and Architectures for Networking and Communications Systems*, Princeton, NJ, October 2005.

[29] J. Dongarra and A. V. D. Steen. Overview of Recent Supercomputers. http://www.netlib.org/utk/papers/advanced-computers/overview.html.

[30] N. Dorairaj, E. Shiflet, and MarkGoosman. PlanAhead Software as a Platform for Partial Reconfiguration. *Xcell Journal Online*, (4):68–71, 2005.

[31] S. A. Edwards. The Challenges of Hardware Synthesis from C-like Languages. In *Proceedings of the International Workshop on Logic and Synthesis*, Temecula, California, June 2004.

[32] G. Estrin. Organization of Computer Systems–The Fixed Plus Variable Structure Computer. In *Proceedings of Western Join Conference*, pages 33–40, New York, 1960.

[33] G. Estrin. Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer. *IEEE Annals of the History of Computing*, pages 3–8, February 2002.

[34] S. G. et al. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1999.

[35] M. Finley, E. Anderson, and J. Agron. Thread Manager Design Document. http://wiki.ittc.ku.edu/hybridthread/Image:ThreadManager.pdf. Technical Report.

[36] M. Forum. Message passing interface forum. http://www.mpi-forum.org/.

[37] J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the streams-c c-to-fpga compiler: an applications perspective. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 134–140, New York, NY, USA, 2001. ACM Press.

[38] E. Fu, J. Fleischer, L. Yu, and R. Selbak. Open POSIX Test Suite. http://posixtest.sourceforge.net/.

[39] D. Galloway. The transmogrifier c hardware description language and compiler for fpgas. In *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, page 136, Washington, DC, USA, 1995. IEEE Computer Society.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patters: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1995.

[41] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems*, 2006:1–19, 2006.

[42] J. Gauch. Ku image processing library. http://www.ittc.ku.edu/ jgauch/teaching/742.s07/hw/kuim.tar.

[43] M. B. Gokhale and J. M. Stone. Napa c: Compiling for a hybrid risc/fpga architecture. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 126, Washington, DC, USA, 1998. IEEE Computer Society.

[44] M. B. Gokhale and J. M. Stone. Automatic allocation of arrays to memories in fpga processors with multiple memory banks. In *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 63, Washington, DC, USA, 1999. IEEE Computer Society.

[45] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[46] R. C. Gonzalez and R. E. Woods. *Digital Image Processing, Second Edition*. Prentice Hall, Inc., Upper Saddle River, New Jersey, 2002.

[47] M. Graphics. Modelsim. http://www.model.com/.

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

[49] A. Haar. Zur Theorie der Orthogonalen Funktionensysteme. *Math. Annal.*, 69:331–371, 1910.

[50] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2003.

[51] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of the Institure of Radio Engineers*, volume 40, pages 1098–1101, September 1952.

[52] Impulse Accelerated Technologies. Impulse c. http://www.impulsec.com/.

[53] S. C. Inc. Src computers inc. http://www.srccomp.com/.

[54] A. A. Jerraya and W. Walf. *Multiprocessor Systems On Chip*. Morgan Kaufmann, Sanfrancisco, CA, 2005.

[55] A. A. Jerraya and W. Wolf. Hardware/software interface codesign for embedded systems. *Computer*, 38(2):63–69, 2005.

[56] R. Jidin. *Extending the Thread Programming Model Across CPU and FPGA Hybrid Architectures*. PhD thesis, University of Kansas, Apr. 2006.

[57] R. Jidin, D. Andrews, and D. Neihaus. Implementing Multi Threaded System Support for Hybrid FPGA/CPU Computational Components. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 116–122, 2004.

[58] Joint Photographic Experts Group. Jpeg 2000. http://www.jpeg.org/jpeg2000/.

[59] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[60] V. Kindratenko and D. Pointer. A Case Study in Porting a Production Scientific Supercomputing Application to a Reconfigurable Computer. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–22, Napa, California, Apr. 2006.

[61] KU HybridThreads. Hybridthread Subversion Control. http://wiki.ittc.ku.edu/hybridthread/Hybridthread_Version_Control.

[62] S. Kumar, A. Jantsch, M. Millberg, J. Oberg, J.-P. Soininen, M. Forsell, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. *isvlsi*, 00:0117, 2002.

[63] X. Lai and J. L. Massey. A Proposal for a New Block Encryption Standard. In *Proceedings of the Workshop on the Theory and Appiation of Cryptographic Techniques on Adnaces in Cryptology*, pages 389–404, 1991.

[64] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *ISCA*, pages 46–57, 1992.

[65] D. Lau, O. Pritchard, and P. Molson. Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, California, Apr. 2006.

[66] G. D. Micheli. Hardware Synthesis from C/C++ Models. In *Proceedings of Design, Automation and Test in Europe*, pages 382–283, Munich, Germany, March 1999.

[67] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu. Tartan: evaluating spatial computation for whole program execution. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 163–174, New York, NY, USA, 2006. ACM Press.

[68] G. R. Morris, V. K. Prasanna, and R. D. Anderson. A Hybrid Approach for Mapping Conjugate Gradient onto a FPGA Augmented Reconfigurable Supercomputer. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, Napa, California, Apr. 2006.

[69] T. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, pages 52–58, April 2001.

[70] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. In *IEEE Computer*, pages 63–69. IEEE Computer, August 2003.

[71] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1994.

[72] V. Paxson, K. Asanovic, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In *USENIX First Workshop on Hot Topics in Security*, Vancouver, British Columbia, July 2006.

[73] W. Peck, J. Agron, D. Andrews, M. Finley, and E. Komp. Hardware/software co-design of operating system services for threaded management and scheduling. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium, Works in Progress Session (RTSS WIP)*, Lisbon, Portugal, December 2004.

[74] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews. Hthreads: A computational model for reconfigurable devices. In *16th International Conference on Field Programmable Logic and Applications*, Madrid, Spain, August 2006.

[75] F. Petrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in noc based shared memory multiprocessor soc architectures. In *DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 53–60, Washington DC, USA, 2006. IEEE Computer Society.

[76] M. Scarpino. Implementing the Message Passing Interface (MPI) with FPGAs. In *9th Annual International MAPLD Conference*, Washington D.C., September 2006.

[77] A. Schmidt. Quantifying Effective Memory Bandwidth in Platform FPGAs. Master's thesis, The University of Kansas, Lawrence, KS, May 2007.

[78] A. Schmidt and R. Sass. Quantifying Effective Memory Bandwidth in Platform FPGAs. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, April 2007.

[79] B. Schneier. *Applied Cryptography Second Edition: Protocols, Algorithms, and Source Code in C.* John Wiley and Sons, Inc., New York, NY, 1996.

[80] L. R. Scott, T. Clark, and B. Bagheri. *Scientific Parallel Computing.* Princeton University Press, Princeton, New Jersey, 2005.

[81] L. Semeria, K. Sato, and G. D. Micheli. Resolution of dynamic memory allocation and pointers for the behavioral synthesis form c. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 312–319, New York, NY, USA, 2000. ACM Press.

[82] D. Skillicorn. *Foundations of Parallel Programming (Cambridge International Series on Parallel Computation).* Cambridge University Press, New York, NY, USA, 2005.

[83] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[84] D. L. Slotnick, W. C. Borck, and R. C. McReynolds. The Solomon Compuer. pages 97–107, Philadelphia, PA, December 1962.

[85] W. H. M. Smith, B. Hutchins, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prassana, and H. A. E. Spannenburg. Seeking Solutions in Configurable Computing. *IEEE Computer*, 30(12):38–43, December 1997.

[86] B. So, M. Hall, and P.Diniz. A Compiler Approach to Design Space Exploration in FPGA-Based Sysems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.

[87] D. Soderman and Y. Panchul. Implementing c algorithms in reconfigurable hardware using c2verilog. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 339, Washington, DC, USA, 1998. IEEE Computer Society.

[88] D. Soderman and Y. Panchul. Implementing C Designs in Hardware: A Full-Featured ANSI C to RTL Verilog Compiler in Action. In *IVC-VIUF '98: Proceedings of the International Verilog HDL Conference and VHDL International Users Forum*, page 22, Washington, DC, USA, 1998. IEEE Computer Society.

[89] I. Sommerville. *Software Engineering, 6th Edition.* Addison-Wesley, Harlow, England, 2001.

[90] Standord Compiler Group. Stanford university intermediate form. http://suif.stanford.edu/suif/NCI/suif.html.

[91] J. Stevens and F. Biajot. Hybridthreads compiler. http://www.ittc.ku.edu/hybrdithreads/downloads. Technical Report.

[92] C. E. Stroud, R. R. Munoz, and D. A. Pierce. Behavioral model synthesis with cones. *IEEE Design and Test of Computers*, 5(3):22–30, 1988.

[93] SystemC. Systemc. http://www.systemc.org.

[94] S. H. Unger. A Computer Oriented Towards Spatial Problems. In *Proceedings of the Institue of Radio Engineers*, volume 46, pages 1744–1750, October 1958.

[95] University of California, San Diego, Microelectronic Embedded Systems Laboratory. Spark project. http://mesl.ucsd.edu/spark/.

[96] M. Vuletic. *Unifying Software and Hardware of Multithreaded Reconfigurable Applications within Operating System Processes.* PhD thesis, Ecole Polytechnique Federale de Lausanne, July 2006.

[97] M. Vuletic, L. Pozzi, and P. Ienne. Seamless hardware-software integration in reconfigurable computing systems. *IEEE Design and Test*, 22(2):102–113, 2005.

[98] M. Vuletic, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):910–915, August 2006.

[99] R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin, and C. Kitchen. An overview of FPGAs and FPGA programming; Initial experiences at Daresbury. http://www.cse.clrc.ac.uk/disco/publications/FPGA_overview.pdf. Daresbury Laboratory Technical Report, June 2006.

[100] D. W. Wall. Limits of instruction-level parallelism. *SIGARCH Comput. Archit. News*, 19(2):176–188, 1991.

[101] P. Wilson, M. Johnson, and D. Boles. Dynamic Storage Allocation, A Survey and Critical Review. In *International Workshop Memory Management*, Sept. 1995.

[102] W. Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38–43, 2003.

[103] Xilinx. Ml310 documentation and tutorials. http://www.xilinx.com/products/boards/ml310/current/index.html.

[104] Xilinx. Opb ipif. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_ipif.pdf. Product Specification.

[105] Xilinx. Virtex-5 Family Overview LX, LXT, SXT Platforms. Xilinx Documentation. http://direct.xilinx.com/bvdocs/publications/ds100.pdf.

[106] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. Xilinx Documentation. http://direct.xilinx.com/bvdocs/publications/ds083.pdf.

[107] Xilinx. Xilinx. http://www.xilinx.com/.

[108] Xilinx. Xst user guide. http://toolbox.xilinx.com/docsan/xilinx7/books/docs/-xst/xst.pdf.

[109] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf, September 2004. Xilinx Application Note 290.

[110] T.-Y. Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer Aided Design*, pages 288–294, 1995.

[111] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *International Symposium on Computer Architecture*, Madison, WI, USA, June 2005.

[112] L. Zhuo and V. K. Prasanna. High-Performance and Parameterized Matrix Factorization. In *16th International Conference on Field Programmable Logic and Applications*, Madrid, Spain, August 2006.