

The Case for High Level Programming Models for Reconfigurable Computers

David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck,
Jim Stevens, Fabrice Bajjot, and Ed Komp

Information and Telecommunication Technology Center

University of Kansas

2335 Irving Hill Road, Lawrence, KS

{dandrews,rsass,eanderso,jagron,peckw,jstevens,bricefab,komp}@ittc.ku.edu

Abstract

In this paper we first outline and discuss the issues of currently accepted computational models for hybrid CPU/FPGA systems. Then, we discuss the need for researchers to develop new high-level programming models, and not just focus on extensions to programming languages, for enabling accessibility and portability of standard high level applications across the CPU/FPGA boundary. We then present hthreads, a unifying programming model for specifying application threads running within a hybrid CPU/FPGA system. Threads are specified from a single pthreads (POSIX threads) multithreaded application program and compiled to run on the CPU or synthesized to run on the FPGA. The hthreads system, in general, is unique within the reconfigurable computing community as it abstracts the CPU/FPGA components into a unified custom threaded multiprocessor architecture platform. A hardware thread interface (HWTI) component has been developed that provides an abstract, platform-independent compilation target. Thus, the HWTI enables the use of standard thread communication and synchronization operations across the software/hardware boundary.

1 Introduction

Reconfigurable computing (or RC), as a discipline, has now been in existence for well over a decade. During this time, significant strides have been made in fabrication that are now providing hybrid CPU/FPGA components with millions of free logic gates, as well as diffused IP in the form of high-speed multipliers and SRAM blocks [25]. Unfortunately, researchers have thus far struggled

to develop tools and programming environments that allow *programmers and system designers*, and not just hardware designers to tap the full potential of the new reconfigurable chips. This deficiency is in part due to the widespread acceptance of the use of the FPGA as a simple co-processor accelerator, and the subsequent absence of a modern parallel programming model that supports both accessibility and portability for parallel computations across the CPU/FPGA boundary. This should be concerning for the reconfigurable computing community, as lessons learned from past parallel processing efforts clearly indicate the need to provide portable, parallel programming models composed of unaltered high-level languages and middleware libraries. In this paper we first outline and discuss the issues of currently accepted computational models for hybrid CPU/FPGA systems. We then discuss the need for researchers to develop new high-level programming models, and not just focusing on the extension of programming languages to include machine-specific pragmas. Finally, we outline hthreads, a multithreaded programming model and run-time system for achieving seamless interaction of threads running across the CPU/FPGA boundary.

By far, the most commonly accepted computational model within the RC community treats the FPGA as an instruction level *hardware-accelerator* for the CPU. Figure 1 outlines this basic computational model. After traditional software development of a single threaded application, the code is then profiled to find the portion of the application that would provide the most performance benefit from custom hardware acceleration. Once identified, the target code section can be replaced with the custom hardware core. Platform specific interfaces are then created to allow the application software to communicate with and control the hardware core. Typically the software ap-

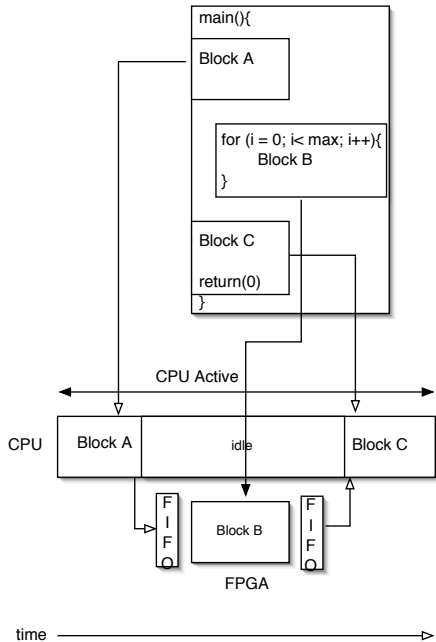


Figure 1: Traditional FPGA Co-Processor Model

plication interfaces with the hardware core through low level component structures such as FIFO queues: one for feeding input data into the hardware core, and a second for streaming output data back to the software application. During execution, the application software executes as normal until it reaches the critical portion, or kernel, of the code and invokes the hardware core. This is accomplished by the CPU first transferring data into the hardware core, and then idling until the hardware core finishes execution. Once the hardware core signals completion, the CPU reads the output data back from the hardware core and resumes normal execution. On small scales this hardware acceleration model is effective in exploiting instruction-level parallelism (ILP) that can be exposed through loop unrolling [13]. 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. As an example, consider the mathematically intensive realm of molecular dynamics applications. In [12], the authors successfully were able to port a molecular dynamics application, called NAMD, to an SRC-6 platform (consisting of both Intel microprocessors and FPGA technologies), using the hardware-acceleration model to achieve a 3.0x speedup.

From a historical perspective, the hardware-

acceleration model is similar to the model assumed within CISC processor architectures. In CISC architectures the CPU executes a series of microcoded instructions for each CISC instruction in its instruction set architecture (ISA). In the hardware-acceleration model, CISC-like instructions are instead implemented as parallelized circuits within the FPGA instead of microcode. Both of these methods strive to achieve maximum performance through low-level instruction customization. Whereas traditional CISC architectures are limited by the microcode ISA and associated fixed structure of available ALUs and datapaths; hardware-acceleration-based reconfigurable architectures are only limited by the number of free gates within an FPGA. These approaches are both similar in their instruction issue semantics as they both require the CPU to stop fetching instructions while the multi-cycle instruction, whether implemented in microcode or within the FPGA, is being executed.

In support of the FPGA as a co-processor model, researchers have been investigating augmentations to existing languages and hardware compilation techniques to allow programmers and system designers to specify their custom accelerators through high-level programming languages [9, 21, 15, 17, 14, 5, 20, 3, 7, 10, 22, 24]. The acceptance of the hardware-acceleration model is further seen through the reinforcement of design tools offered by leading FPGA fabrication houses. Altera has adopted the model for use in their NIOS-II processor along with their C to Hardware (C2H) compiler [6]. The objective of C2H is to isolate a section of code that the programmer has determined as being a good candidate for hardware-acceleration, and then automatically create a hardware core to execute in its place. The C2H compiler has a strong advantage in that a user is not required to have prior knowledge of a hardware description language (HDL) in order to take advantage of being able to implement critical portions of code in hardware. Additionally, the C2H compiler is able to support a large portion of the full ANSI C standard (recursion and floating-point arithmetic being the exception).

Considering the challenge of producing parallel circuits from a purely sequential language, these efforts are quite impressive. The task of translating C to an HDL in a way that increases program performance is non-trivial. The C language provides a thin abstraction of the Von Neumann architecture that reinforces sequential instruction execution and the architecture's associated CPU/memory bottleneck. Furthermore, certain programming constructs and techniques such as pointers and recursion, which are commonly found in software-based systems, are difficult to replicate in hardware. These tools show some of the

results of the great effort that has thus far been put forth in providing software programmers access to the reconfigurable fabric of modern FPGA devices.

In addition to language translation issues, standard abstract communication methods for interfacing the software and hardware portions of an application have also yet to be defined. The need for such capabilities are outlined in [11]. Jerraya [8] and Wolf [26] outline the technical challenges and propose approaches for automating the creation of abstract hardware/software interfaces for embedded systems and multiprocessor systems-on-chip (SoC). Interestingly, the data transfer times across the hardware/software boundary can be significant and occasionally even greater than the original software execution time. Additional consistency problems can also be introduced for modern memory hierarchies utilizing multiple levels of cache. Efforts such as [17] have explored optimization techniques for balancing the number of loops unrolled into parallel circuits to match the bandwidth capabilities of the system bus. Work such as GARP [10] attempted to circumvent the memory bottleneck by proposing a new CPU architecture with an integrated reconfigurable fabric. Unfortunately, [5] showed that applications with limited spatial parallelism within loop bodies coupled with size limitations of the embedded reconfigurable fabric can result in performance degradations when compared to compiler optimized software implementations. Finally, models that seek to exploit low-level parallelism from within a limited sized code fragment in a single execution stream ignore the available user-specified concurrency exposed within modern coarse-grained multithreaded and multitasking models. As Moore's law continues to improve the size and density of FPGAs, new models are required that can translate more coarse levels of parallelism into parallel concurrent circuits within the FPGA.

2 Abstract Programming Models

Conceptually, the approaches for providing access to the hardware acceleration computational model are following a development path similar to that which occurred in the 1980s and 1990s for SIMD and systolic arrays from the parallel and signal processing domains. In these approaches higher-level languages are augmented with specific pragmas for exploiting fine-grained, arithmetic and instruction-level parallelism in a form that matches the underlying machine's computational model. Although these approaches bring advancements in *accessibility*, they do so at the cost of *portability* by promoting detailed knowledge of the underlying architec-

ture directly into the source language. This is concerning for reconfigurable computing as lessons learned from the parallel processing domain clearly show the importance of decoupling machine-specific attributes from the high-level language to achieve portability and the importance of exploiting coarser-grained, thread-level parallelism. Current practices as evidenced from software developed for high-performance cluster computing, encapsulate machine-specific code into middle-ware libraries that can be linked in with unaltered source code to form an abstract programming model.

Informally, abstract programming models provide a framework of system software components and their interactions that are platform independent and portable. Portability is achieved by separating policy from mechanisms within the framework. The policy is specified through a common high-level language and set of system service APIs. Modern programming models achieve portability by adopting unmodified high-level languages and middle-ware service routines. Recently, the multithreaded programming model has gained in popularity within the embedded systems domain with its ability to represent time and event-triggered reactive processes typical of this domain. The multithreaded programming model is also familiar within the realm of general purpose computing; providing a convenient framework for processing concurrent client requests on a common server. Support for the multithreaded programming model is now provided through the pthreads (POSIX threads [4]) library released with Linux, Unix, and Windows. Additional low-level hardware support for multithreading is also standard within modern CPUs as evidenced by Intel's hyperthreading technology [23].

3 hthreads: A Multithreaded Programming Model

The high-level design flow for hthreads, our multithreaded programming model for hybrid CPU/FPGA architectures, is shown in Figure 2. In the hthreads design flow, programmers can express their system computations using familiar pthreads semantics, based on a set of application requirements. The functional flow of the threaded high level program can be written and verified using a standard workstation running Linux and pthreads prior to synthesis and hardware design. Hthreads' APIs are compatible with pthreads APIs, and wrappers going from pthreads to hthreads as well as hthreads to pthreads are available. After initial debugging on a standard workstation, the multithreaded application can be profiled on the workstation

or run through on-the-board testing, allowing the developer to identify which threads should be mapped into the reconfigurable fabric.

As an example of the ease in which hthreads supports seamless creation of threads for execution in either hardware or software, consider the code seen in Figure 3. This example shows the application-level code for the parent thread creating four child threads; two within software and two within hardware. In this example, each child thread implements a complete Discrete Wavelet Transform (DWT). The parent thread can create multiple child threads to run *in parallel* within the hardware, and synchronize with the threads as if they were traditional software threads. Figure 4 show the execution times of several configurations of the DWT child threads. The first two timings are for a single child thread, representing a classic single instruction stream FPGA accelerator model. Not surprisingly, the hardware implementation shows 7.5x speedup compared to the software version. The next two timings highlight the benefits of parallelism. For two software threads running on a single CPU, each thread must be time multiplexed with the total execution time being the summation of the independent execution times of each thread, plus operating system overhead. However, when one of the threads is mapped into hardware, parallelism is achieved. Two hardware threads running in parallel show 11x speedup when compared to two software threads time multiplexing on the CPU. This simple example illustrates the benefit of enabling coarse grained multiple instruction streams, multiple data (MIMD) parallelism within the FPGA.

Although conceptually simple, extending hthreads across a *reconfigurable system* faced two key challenges. First, our design flow was modified to support the synthesis of the application program code, and linking of the (APIs) to state machine versions of syscall run time services. In effect, the API's provide consistent policies for threads running on both the CPU and within the FPGA. This includes the ability to support standard programming function invocations, and creating and passing abstract data types and pointers in accordance with the semantics of each pthread API.

Figure 6 shows a high level description of our integrated compilation/synthesis tool flow. As shown in Figure 6 we have augmented the standard gcc tool chain to produce and output a new hardware intermediate form (HIF) from which we then generate VHDL. The HIF is similar to standard single assignment intermediate forms, but in a slightly modified and controlled format to better serve as the target for VHDL generation.

The second challenge was to create new hardware ver-

sions of system service libraries to support abstract API operations from and to hardware threads. To support our shared-memory thread model, we created services to support the creation, control, and scheduling of child threads executing in hardware. Additionally, we created services that allowed the independent hardware threads to synchronize with all other threads using standard semaphore operations, and independently access global and local data. All services are invoked, even within hardware threads, from the original user specified hthreads APIs of the source program. Thus, our hthreads API's eliminate the need to create unique interfaces for threads to interact across the CPU/FPGA, or hardware/software boundary. This allows the high level code to be portable between software and hardware, and different platforms as verified by our high level design flow.

3.1 HWTI Abstract Interface

We created the hardware thread interface (HWTI) to encapsulate our system service mechanisms for hardware threads. A block diagram of the hardware thread interface component is shown in Figure 7. As shown in Figure 8, the HWTI entity is linked in with the automatically generated VHDL architecture version of the user code in a similar fashion to traditional system call software routines. The HWTI component contains two interfaces; the HWTI system interface for interacting with other hthread service components, and the HWTI user interface for supporting system service calls from the user thread. The HWTI is thus a target that can either be automatically linked in with our HLL to VDHL synthesizer, or directly included by developers wishing to hand write threads in VHDL.

Figure 5 shows the analogous nature of our common run time system services available to both hardware and software threads. As shown in the center column of Figure 5 standard policy for invoking run time services is based on two steps. First, arguments are passed from the application program to the run-time services, and second the run-time service routine is invoked. Within traditional software methods, this policy is achieved by pushing the arguments onto the stack, and then executing a specific trap instruction to the run-time service. To achieve an analogous policy for threads running in hardware, the hardware interface provides registers that replace the stack, and a command register that replaces a trap. As shown in Figure 5 the mechanism used to provide the passing of arguments is to simply drive the inputs to the HWTI registers.

Specific syscall functionality is specified by a unique opcode shown in Table 1. For example when the

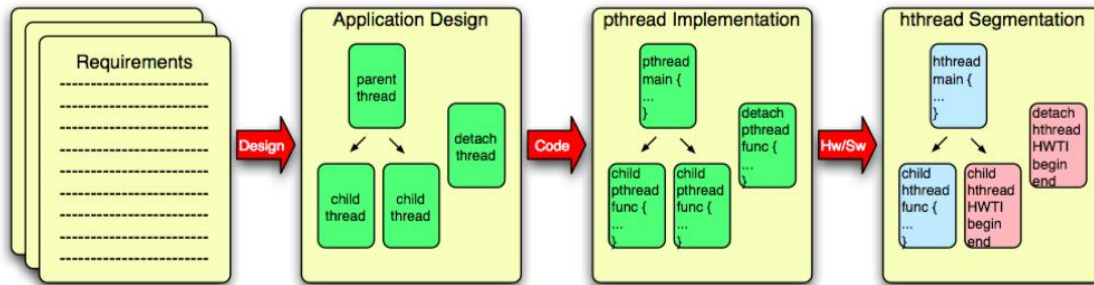


Figure 2: hthread Design Flow

```

pthread_t      hw1, hw2, sw1, sw2;
hthread_attr_t attr1, attr2, attr3, attr4;
struct Array  arg1, arg2, arg3, arg4;
Huint        i, retval;
log_t log;

// Setup the UART for printing

// Initialize the hybridthreads system
hthread_init();

// Initialize the attributes for threads
hthread_attr_init( &attr1 );
hthread_attr_init( &attr2 );
hthread_attr_init( &attr3 );
hthread_attr_init( &attr4 );

// Setup the attributes for the hardware thread
hthread_attr_sethardware( &attr1, HWTI_BASEADDR_ZERO );
hthread_attr_sethardware( &attr2, HWTI_BASEADDR_ONE );

// Set the thread's argument data to some value
arg1.length = LENGTH;
arg2.length = LENGTH;
arg3.length = LENGTH;
arg4.length = LENGTH;
for( i = 0; i<LENGTH; i++ ) {
    arg1.data[i] = (100 + i*4) % LENGTH;
    arg2.data[i] = (101 + i*3) % LENGTH;
    arg3.data[i] = (102 + i*2) % LENGTH;
    arg4.data[i] = (103 + i*1) % LENGTH;
}

for( i = 0; i < LENGTH; i++ ) {
    printf( "%i = %i\n", i, arg3.data[i] );
}

log_create( &log, 1024 );
log_time( &log );
hthread_create( &sw1, &attr3, dwtHaar, &arg3 );
hthread_create( &sw2, &attr4, dwtHaar, &arg4 );
hthread_create( &hw1, &attr1, NULL, &arg1 );
hthread_create( &hw2, &attr2, NULL, &arg2 );

// Wait for the hardware thread to exit

hthread_join( hw1, (void*)&retval );
hthread_join( hw2, (void*)&retval );
hthread_join( sw1, (void*)&retval );
hthread_join( sw2, (void*)&retval );
log_time( &log );

// Clean up the attribute structure
hthread_attr_destroy( &attr1 );
hthread_attr_destroy( &attr2 );
hthread_attr_destroy( &attr3 );
hthread_attr_destroy( &attr4 );

log_close_ascii( &log );
for( i = 0; i < LENGTH; i++ ) {
    printf( "%i = %i\n", i, arg3.data[i] );
}
printf( "-- QED --\n" );

// Return from main
return 1;
}

```

Uniform API's

Figure 3: Application-Level Code for Creating Hybrid Threads

Number of Threads	Type of Threads	Total Time (ms)	Speedup	Graphic
1	1 software	26.2		
	1 hardware	3.5	7.51	
2	2 software	52.1		
	1 software, 1 hardware	26.7	1.95	
	2 hardware	4.7	11.01	
3	1 software, 3 hardware	27.4		???
4	2 software, 2 hardware	53.8		

Figure 4: DWT Example Run Time Performance

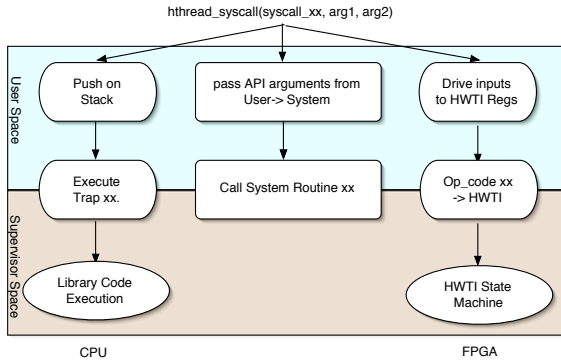


Figure 5: System Service Policy and Mechanisms

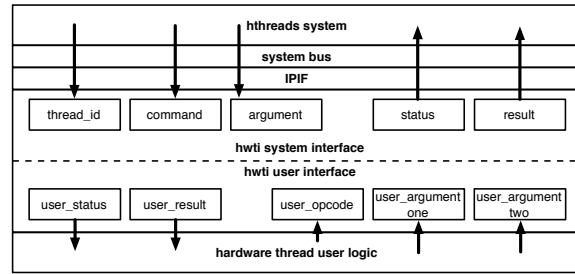


Figure 7: HWTI Block Diagram

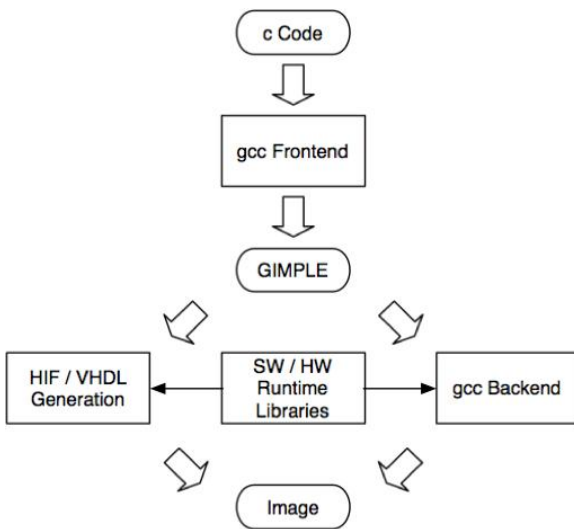


Figure 6: Compilation/Synthesis Tool Flow

user thread has completed and executes a `hthread_exit()` API, the opcode `HTHREAD_EXIT` is placed on the `thrd2intrfc_opcode` line and latched in the `user_opcode` register. This can be seen through the VHDL example of Figure 8. In accordance with the standard `pthread_exit()` system call, the user thread may pass one parameter back to the parent thread using the HWTI's `user_argument_one` register. The `thrd2intrfc_argument` registers are also used to access global memory, in combination with the `LOAD` and `STORE` opcodes.

The HWTI is implemented in 404 slices. This number includes logic for the standard vendor supplied bus interface (IPIF) and a minimal user logic thread that immediately exits following a `RUN` command. The 404

```

ENTITY simple_thread IS
  port (
    clk : in std_logic;
    intrfc2thrd_status : in std_logic_vector(0 to 3);
    intrfc2thrd_result : in std_logic_vector(0 to 31);
    thrd2intrfc_opcode : out std_logic_vector(0 to 7);
    thrd2intrfc_argument_one : out std_logic_vector(0 to 31);
    thrd2intrfc_argument_two : out std_logic_vector(0 to 31)
  );
END ENTITY simple_thread;

ARCHITECTURE beh OF simple_thread IS
  ... constant & variable declaration ...
BEGIN
  update : PROCESS
  BEGIN
    wait until rising_edge(clk);
    IF intrfc2thrd_status = USER_STATUS_RESET THEN
      ... reset variables ...
    ELSE
      IF ((intrfc2thrd_status = USER_STATUS_RUN)
        OR (intrfc2thrd_status = USER_STATUS_ACK)) THEN
        CASE current IS
          WHEN N0 =>
            IF intrfc2thrd_status = USER_STATUS_RUN THEN
              current <= N8;
            ELSE
              current <= N0;
            END IF;
          ... additional state machine logic ...
        END CASE;
      END IF;
    END IF;
  END PROCESS;
END beh;

```

Figure 8: VHDL Generated Code for Simple Thread

Table 1: HWTI System Calls

Syscall	Description
NOOP	An operation is not being requested
HTHREAD_EXIT	User thread finished executing and returns results in argument register
LOAD	User thread requesting to read from memory
STORE	User thread requesting write to memory
HTHREAD_SELF	Returns the thread_id
HTHREAD_YIELD	No meaning for a hardware thread, resumes execution immediately
HTRHEAD_MUTEX_LOCK	User thread requesting to lock a mutex
HTHREAD_MUTEX_UNLOCK	User thread requesting to unlock a mutex

slices represent 2% of all slices on our Xilinx Virtex II Pro FPGA (XC2VP30). Timing results for each HWTI operation are listed in Table 2. With the exception of LOAD and STORE, these results were recorded using a cycle accurate simulation of the entire HybridThreads system. LOAD and STORE were recorded using timings from on-chip execution, with the hardware thread instantiated on the OPB bus and DRAM on the PLB bus.

3.2 hthreads Hw/Sw Run Time Kernel

Although at first glance mirroring traditional run-time software thread system services within hardware components appears daunting, it instead provided motivation to redesign a complete set of more efficient globally accessible shared services for both hardware and software threads.

Figure 9 shows the hthreads run-time system components implemented in hardware. Hthreads migrates a Thread Manager, Scheduler, Mutex Manager, and a new CPU Bypass Interrupt Scheduler (CBIS) into hardware. Migrating these services into hardware brings significant performance benefits to software threads through more efficient invocation and processing mechanisms [16, 1]. First, invocation mechanisms for accessing the system services are no longer based on inefficient traversal of hierarchical software protocol stacks, but instead are achieved through lightweight atomic load and store operations. Second, speculative and variable execution performed within key system services such as the scheduler are eliminated. As an example, Figure 10 shows comparative timings for executing typical scheduler services for a system with 2 and 250 active software threads running within hthreads. The overhead for making a scheduling decision is now constant, with negligible jitter. The actual overhead for selecting the next thread to be run within the hardware-based scheduler is 240 clock cycles; a constant delay, independent of the number of threads in the ready-to-run queue [2]. The small amount of jitter seen in

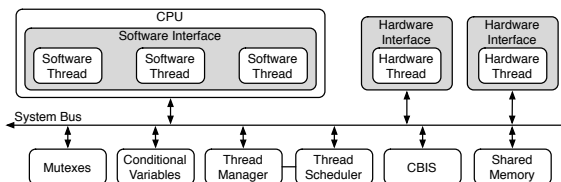


Figure 9: hthread System Block Diagram

Figure 10 is solely due to cache misses during the swapping of thread contexts on the CPU. The Scheduler makes all scheduling decisions a priori, in parallel to application programs running on the CPU. The CPU is only interrupted when a thread entering the ready-to-run queue has a higher priority than the thread running on the CPU as well as any other threads within the ready-to-run queue. In contrast, existing software schedulers must be invoked via an interrupt to the CPU just to consider if an event may or may not trigger a true scheduling decision (such as the release of a mutex).

As a more complete example, the mutex_unlock() operation illustrated in Figure 11 shows the processing steps the hthread system performs to release a mutex, make a scheduling decision, and resume the execution of a thread. In a traditional operating system steps A through E are performed completely in software on the CPU. These steps would require a context switch from the application thread to the system services, and must be performed before the scheduler considers if a new scheduling decision is required based on the queuing of a blocked thread. In hthreads, steps B through G are performed in hardware, allowing the CPU to continue executing the application thread. For systems with both hardware and software threads, migrating this processing off the CPU is critical as significant overhead and jitter can be introduced if the CPU must perform this pre-scheduler speculative processing for hardware threads being unblocked. In [18, 19] a multithreaded capability is reported that supports the cre-

Table 2: Timing Results for HWTI Operations

Command	Clock Cycles	Comment
Write to thread_id register	5	Time from receiving the thread id to the time the system status changes to USED
Write RUN into command register	5	Time from receiving RUN command to time user_status register changes to RUN
Write RESET to command register	4	Time from receiving RESET command to time user_status register changed to UNUSED
LOAD	60	Time user thread issues LOAD opcode to time HWTI returns user_status to RUN including bus transaction
STORE	32	Time user thread issues STORE opcode to time HWTI returns user_status to RUN including bus transaction time
HTHREAD_YIELD	5	Time from user thread issuing opcode to time HWTI returns user_status to RUN
HTHREAD_SELF	5	Time from user thread issuing opcode to time HWTI returns user_status to RUN
HTHREAD_MUTEX_LOCK	20	Time from user thread issuing opcode to time HWTI returns user_status to RUN, including bus transaction time and Mutex Manager time
HTHREAD_MUTEX_UNLOCK	20	Time from user thread issuing opcode to time HWTI returns user_status to RUN, including bus transaction time and Mutex Manager time
HTHREAD_EXIT	20	Time from user thread issuing opcode to time HWTI ends bus transaction with Thread Manager and system status changes to EXIT

	2 Running Software Threads			250 Running Software Threads		
	Min (μs)	Mean (μs)	Max (μs)	Min (μs)	Mean (μs)	Max (μs)
Scheduling Decision	1.750	1.751	2.140	1.910	1.975	3.380
Mutex Lock	.750	.750	.750	.750	.750	.750
Interrupt Handler Determination	.760	.760	.760	.760	.796	1.530

Figure 10: hthread Performance Summary

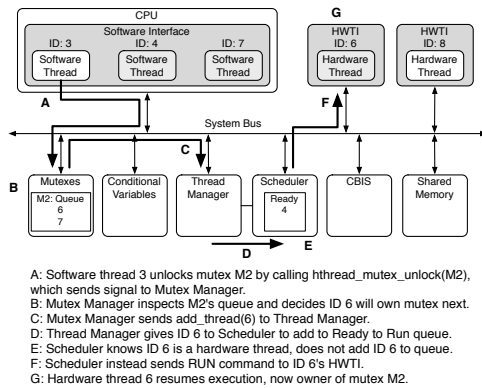


Figure 11: hthread Mutex Unlock Sequence

ation and control of both hardware and software threads through Linux. This approach was taken to allow hardware threads to access data through Linux's existing virtual memory address space. Although convenient, this approach requires additional complexity within the hardware thread to maintain virtual address translation tables, and invokes the memory manager running on the CPU for page swapping through external interrupts, thus introducing jitter and overhead.

4 Conclusion

In this paper, we have discussed existing computational models for hybrid CPU/FPGA systems, and the need for the creation of standard parallel programming models. We then presented hthreads, a unifying multithreaded programming model for controlling hardware and software threads running across the CPU/FPGA boundary. Hthreads provides system service libraries that encapsulate platform specific operations under pthreads compatible APIs. This allows threads specified from a single pthreads multithreaded application program to be compiled to run on the CPU or synthesized to run on the FPGA. To support the abstraction of the CPU/FPGA component boundary, we have created the hardware thread interface (HWTI) component that frees the designer from having to specify and embed platform specific instructions to form customized hardware/software interactions. Instead, the hardware thread interface supports the generalized pthreads API semantics. This approach follows accepted practices within the high performance computing community that can bring both accessibility and portability to the reconfigurable computing domain. Our ability to allow multiple execution threads to exist within the FPGA

also provides a new mechanism to exploit the full potential of the FPGA.

Acknowledgment

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

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) 2004*, 2004.
- [2] 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*, September 19–22, 2005.
- [3] P. Athanas and H. Silverman. Processor Reconfiguration Through Instruction-set Metamorphosis. In *IEEE Computer*, volume 26, pages 11–18, 1993.
- [4] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [5] T. Callahan. Automatic Compilation of C for Hybrid Reconfigurable Architectures. *Ph.D. Dissertation at the University of California Berkeley*.
- [6] O. P. David Lau and P. Molson. Automated Generation of Hardware Accelerators with Direct Memory Access. In *Proceedings of the 14th Annual Conference on Field-Programmable Custom Computing Machines*, 2006.
- [7] M. Gokhale and R. Minnich. FPGA Computing in a Data Parallel C. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–101, 1994.
- [8] A. Grasset, F. Rousseau, and A. A. Jerraya. Automatic generation of component wrappers by composition of hardware library elements starting from communication service specification, 2005.
- [9] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *International Conference on VLSI Design*, January 2003.
- [10] J. Hauser and J. Wawrzynek. GARP: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, 1997.
- [11] A. A. Jerraya and W. Wolf. Hardware/software interface co-design for embedded systems, February 2005.
- [12] V. Kindratenko and D. Pointer. Automated Generation of Hardware Accelerators with Direct Memory Access. In *Proceedings of the 14th Annual Conference on Field-Programmable Custom Computing Machines*, 2006.
- [13] R. Lysecky and F. Vahid. A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. In *Design Automation and Test in Europe (DATE)*, volume 01, pages 18–23, Munich, Germany, 2005.
- [14] 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, August 2003.
- [15] J. Park, P. C. Diniz, and K. S. Shayee. Performance and Area Modeling of Complete FPGA Designs in the Presence of Loop Transformations. *IEEE Transactions on Computers*, 53(11):1420 thru 1435, November 2004.
- [16] W. Peck, J. Agron, D. Andrews, M. Finley, and E. Komp. Hardware/Software Co-Design of Operating Systems for Thread Management and Scheduling. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium, Works in Progress Session (RTSS WIP) 2004*, 2004.
- [17] B. So, M. Hall, and P. Diniz. A Compiler Approach to Design Space Exploration in FPGA-Based Systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.
- [18] M. Vuletic, L. Possi, and P. Ienne. Virtual Memory Window for Application-Specific Reconfigurable Coprocessors. In *Proceeding of the 41st Annual Conference on Design Automation*, ACM Press, pages 948–953, 2004.
- [19] M. Vuletic, L. Pozzi, and P. Ienne. Seamless Hardware Software Integration in Reconfigurable Computing Systems. *IEEE Design and Test of Computers*, pages 102–113, 2005.
- [20] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, 1993.
- [21] www.celoxica.com. Celoxica.
- [22] www.impulsec.com. ImpulseC.
- [23] www.intel.com/technology/hyperthread. Intel Hyper-Threading Technology.
- [24] www.systemc.org. SystemC.
- [25] Xilinx. Programmable logic devices. <http://www.xilinx.com/>. Last accessed May 6, 2006.
- [26] T.-Y. Yen and W. Wolf. Communication synthesis for distributed embedded systems, 1995.