Achieving Programming Model Abstractions for Reconfigurable Computing

David Andrews, *Senior Member, IEEE*, Ron Sass, *Member, IEEE*, Erik Anderson, *Student Member, IEEE*, Jason Agron, *Student Member, IEEE*, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp

***Abstract*—This paper introduces hthreads, a unifying programming model for specifying application threads running within a hybrid computer processing unit (CPU)/field-programmable gate-array (FPGA) system. Presently accepted hybrid CPU/FPGA computational models—and access to these computational models via high level languages—focus on programming language extensions to increase accessibility and portability. However, this paper argues that new high-level programming models built on common software abstractions better address these goals. The hthreads system, in general, is unique within the reconfigurable computing community as it includes operating system and middleware layer abstractions that extend across the CPU/FPGA boundary. This enables all platform components to be abstracted into a unified multiprocessor architecture platform. Application programmers can then express their computations using threads specified from a single POSIX threads (pthreads) multithreaded application program and can then compile the threads to either run on the CPU or synthesize them to run within an FPGA. To enable this seamless framework, we have created the hardware thread interface (HWTI) component to provide an abstract, platform-independent compilation target for hardware-resident computations. The HWTI enables the use of standard thread communication and synchronization operations across the software/hardware boundary. Key operating system primitives have been mapped into hardware to provide threads running in both hardware and software uniform access to a set of sub-microsecond, minimal-jitter services. Migrating the operating system into hardware removes the potential bottleneck of routing all system service requests through a central CPU.**

***Index Terms*—Field-programmable gate arrays (FPGAs), operating systems, programming models, reconfigurable computing.**

I. 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 computer processing unit (CPU)/field-programmable gate array (FPGA) components with millions of free logic gates, as well as diffused intellectual property (IP) in the form of high-speed multipliers and SRAM blocks [1].

Manuscript received May 10, 2006; revised April 6, 2007. This work was supported by the National Science Foundation EHS under Contract CCR-0311599.

D. Andrews is with the Electrical Engineering and Computer Science Department, University of Kansas, Lawrence, KS 66045-7612 USA (e-mail: dandrews@ittc.ku.edu).

R. Sass is with the Department of Electrical and Computer Engineering, University of North Carolina, Charlotte, NC 28223 USA.

E. Anderson is with Information Science Institute, University of Southern California, Los Angeles, SC 22203 USA.

J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp are with the Information and Telecommunication Technology Center, University of Kansas, Lawrence, KS 66045-7612 USA.

Digital Object Identifier 10.1109/TVLSI.2007.912106

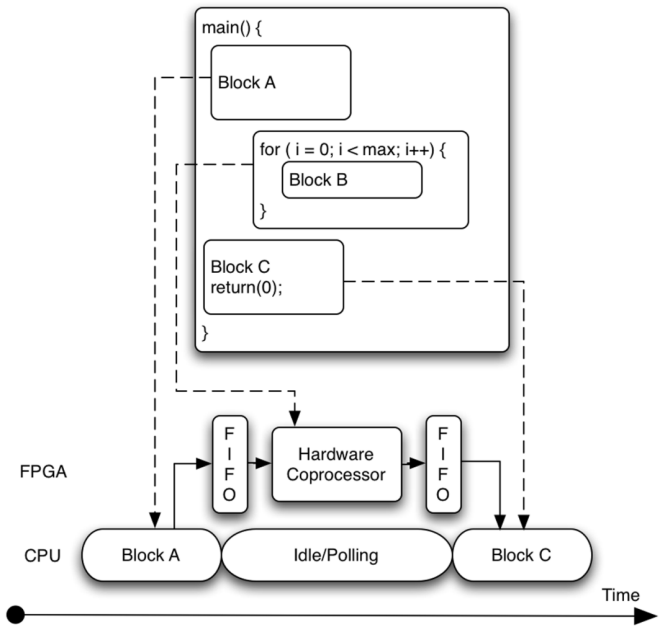


Fig. 1. Traditional FPGA coprocessor model.

Unfortunately, researchers have thus far struggled to develop tools and programming environments that allow *programmers and system designers*—not just hardware designers—to tap the full potential of the new reconfigurable chips. This deficiency is in part due to the absence of modern operating system and middleware services that extend across the CPU/FPGA boundary. These layers, along with a high-level language, form an abstract computational model of a virtual machine. Importantly, these layers provide the concurrency and synchronization mechanisms used within modern software concurrency models such as asynchronous threads.

In the absence of these concurrency mechanisms, research has focused on implicit automated compiler extraction of loop-level parallelism, and explicit augmentations to sequential programming languages to achieve parallelism. As shown in Fig. 1, these approaches treat the FPGA as a coprocessor on which low-level instruction- and data-level parallelism can be mapped as an extension of the execution stream running on the CPU. These approaches are 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, operating systems, 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

to adopt a modern virtual machine approach and associated abstract computational model, that hides the platform specific CPU/FPGA distinctions from the programmer. We then present hthreads, an abstract computational model composed of hardware (HW)/software (SW) codesigned operating system and middleware services that supports the multithreaded programming model. Hthreads allows programmers to work from within the standard software concurrency model of asynchronously running threads communicating through shared memory. The hthreads compiler and run-time libraries allows programmers to write multithreaded programs in C and standard pthreads air position indicators (APIs). The hthreads operating system and middleware services provide the mechanisms that allow the threads to run on either the CPU, or within a custom circuit on the FPGA. For applications where maximal data level parallelism is required, individual threads specifically targeted for execution within a custom circuit can also be written in VHDL and linked into the run-time framework. Thus, hthreads represents a framework that can support both multiple instruction-multiple data (MIMD) and classic single instruction multiple data model (SIMD) parallelism within the asynchronous software concurrency model.

A. Computational Models

Computational models provide a description of a machines primitive capabilities [2]. The basic attributes of a computational model were categorized by Brown [3] as: 1) a machine's primitive units; 2) control and data mechanisms; 3) communication capabilities; and 4) synchronization mechanisms. Computational models can be used to describe the primitive operations of the physical machine as well as the primitive operations of a higher level virtual machine. Tanenbaum [4] provided the following hierarchy of computational models. We have updated Tanenbaum's original hierarchy to also include more modern middleware services shown as follows:

1) *algorithm* or high-level plan of attack;
2) *high-level language* in which the user writes a program for the computer;
3) **middleware services* which provides an abstraction layer for the high-level language;
4) *operating system*, a program that allocates the resources of th system, using its own abstract picture of the system;
5) *physical machine* architecture, represented by its instruction set.

For a general purpose CPU, the instruction set architecture (ISA) is the computational model of the physical machine. This simple model, called the Von Neumann model in honor of John Von Neumann and his associates, is categorized within Flynn's [5] taxonomy of machine organizations as a single instruction stream—single data stream (SISD) computer. Historical work within reconfigurable computing has primarily focused on bringing the FPGA under this organization to exploit instruction- and data-level parallelism within the FPGA. The FPGA has been brought within a single CPU's SISD machine organization by replacing a sequence of general purpose assembler instructions with a coprocessor instruction that initiates execution of a customized circuit on the FPGA. This is similar to the organization of historical complex instruction set computers (CISC) but with gates replacing traditional microcode. The SISD machine organization has also been extended to the SIMD machine organization using the FPGA as a slave array of coprocessors. To program this organization, data parallelism must be exposed either implicitly within the compiler by unrolling loops, or explicitly within the source program through augmentations and library routines. In both cases, the CPU serves as a front-end instruction sequencer, while the FPGA serves as a set of slave coprocessors.

Computational models for both SISD and SIMD machine organizations are largely formed through the addition of control and data selection instructions added into an existing sequential instruction set. Programmers then deal directly with the physical computational model. In contrast to SISD and SIMD machine organizations, the multiple instruction multiple data (MIMD) organization has evolved as the platform of choice for achieving parallelism. MIMD organizations, such as clusters, are composed of multiple processing elements connected across interconnection networks. Programmers do not deal with the physical computational model of an MIMD machine directly. Instead, programmers interact with a virtual computational model formed through a composition of services provided by the operating system, middleware, and high-level languages. Different attributes of the composite model can be provided through a variety of combinations of the high-level language, operating system, and middleware. Targeting this virtual computational model brings portability across specific physical computational models. Unfortunately, the absence of such a virtual computational model extending across the CPU/FPGA boundary has eliminated the ability to create asynchronous threads within the FPGA that independently run and communicate with other threads running within the system. This has led to the misconception that FPGAs are not suitable for supporting common software concurrency models such as asynchronous threads [6]. This misconception is also reinforced by the cursory observation that the physical circuit structure of an FPGA is composed of programmable logic blocks that are fundamentally synchronous. Clearly, the general purpose CPUs within an MIMD organization also represent devices with physically synchronous computational models. The operating system and middleware provides the framework that enables standard software asynchronous concurrency models.

B. Abstract Programming Models

Programming models have evolved over the last decade as abstract frameworks within which programmers can define system components and their interactions. Within modern software engineering practices, abstract programming models can bring portability through separating policy from mechanism within the framework. The multithreaded programming model defines the policies that enable the creation and running of asynchronous threads that communicate through shared memory. The pthreads libraries represent specific implementations of the multithreaded programming model. Current software practices for high-performance cluster computing combine the use of unaltered languages, such as C, and middleware libraries that encapsulate all machine-specific code. These are then linked with the operating system to form a framework

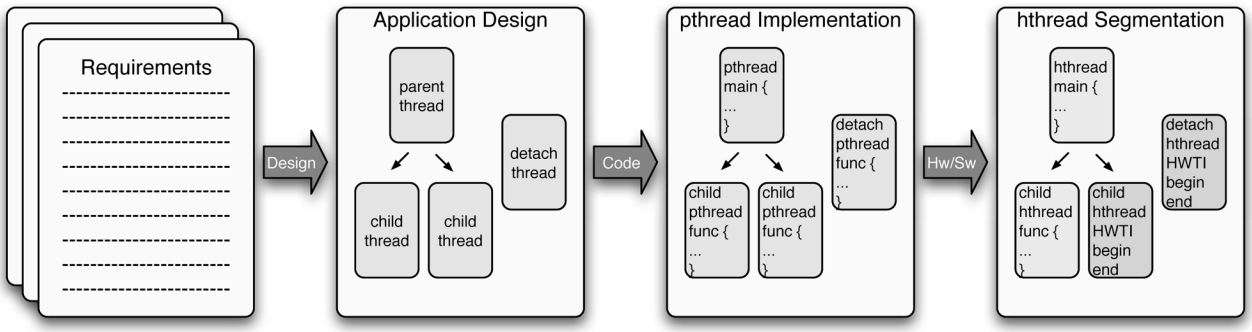


Fig. 2. Hthreads design flow.

which fully supports the policies established within the abstract programming model. As such, the programming model is implemented on the computational model of the virtual and not the physical machine. Therefore, enabling the asynchronous multithreaded programming model across the CPU/FPGA boundary requires bringing the FPGA under both the virtual and physical computational models. Pragmatically, this requires extending operating system and middleware services across the CPU/FPGA boundary.

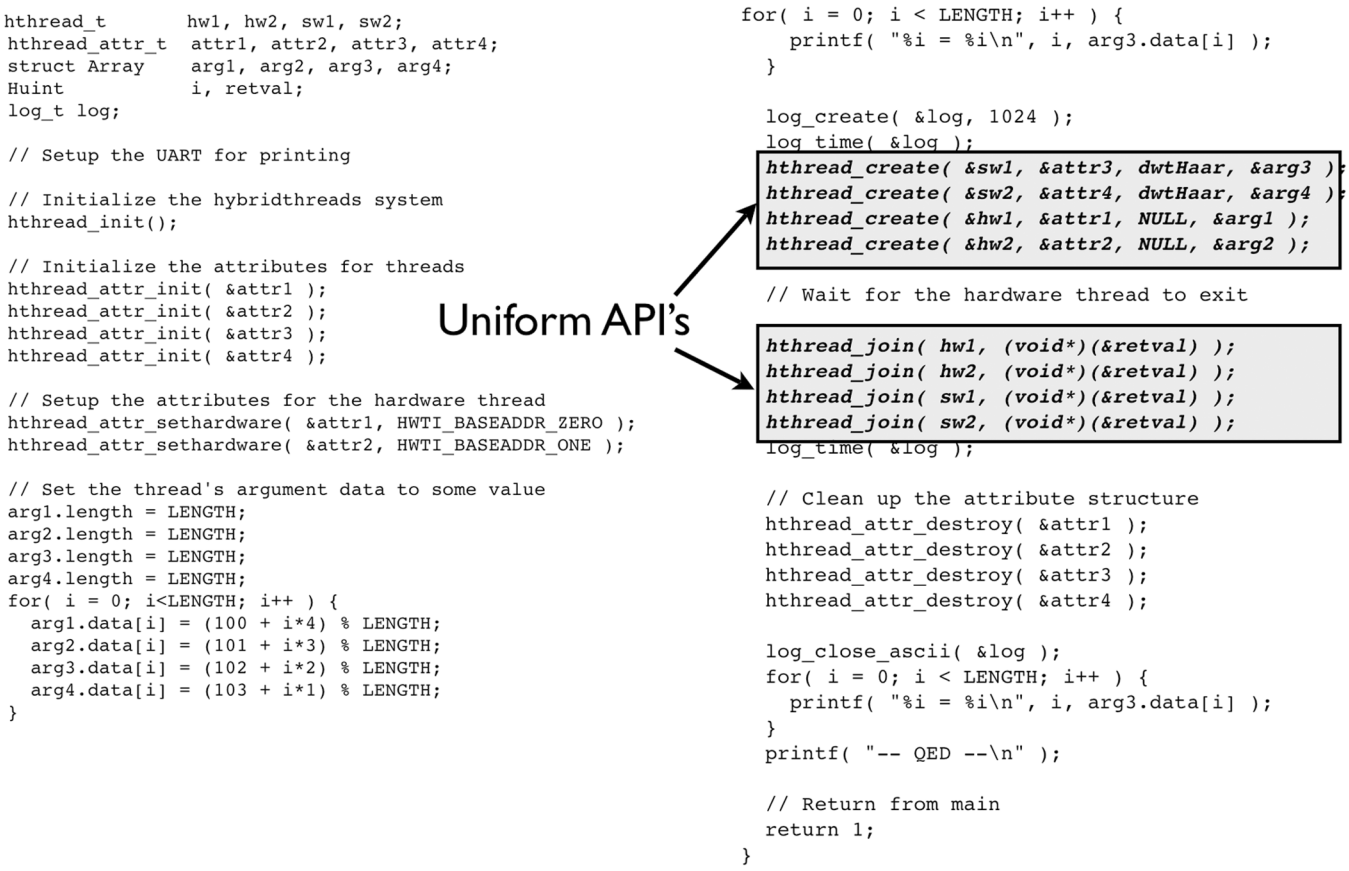
The absence of this virtual computational model, and specifically the ability to support modern asynchronous threads through operating system and middleware support has reinforced the continued exploration of how to configure the FPGA as a coprocessor within the controlling CPU's SISD and extended SIMD machine organizations. Specific attributes of the physical computational model of these machines are either reflected back into the high-level language or automatically extracted within a parallelizing compiler [7]–[18]. Leading FPGA fabrication houses are also working on tools that allow interface extensions and custom circuits to be specified from a high-level language. Altera has developed a Hardware (C2H) compiler [19] for their NIOS-II processor. 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 and CPU to FPGA interface to create a coprocessor. The C2H compiler has a strong advantage in that a user is not required to have prior knowledge of a hardware description language (HDL) 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 exceptions).

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 nontrivial. The C language by itself only provides a thin abstraction of the Von Neumann architecture. Interestingly, modern asynchronous multithreaded programming models use C to describe the computations within threads. However, the language itself must be used with the operating system and middleware services to create the virtual computational model on which the asynchronous programming model runs. Thus, we feel that C can be used as the programming language for reconfigurable computing, but only if the requisite operating system and middleware services are also provided to the asynchronous threads.

II. Hthreads: A Multithreaded Programming Model

The high-level design flow for hthreads, our multithreaded programming model for hybrid CPU/FPGA architectures, is shown in Fig. 2. In the hthreads design flow, programmers express their system computations using traditional pthreads semantics. The high-level program can be written and functionally verified using a standard workstation running Linux prior to synthesis and hardware design. Hthreads' APIs are fully compatible with pthreads APIs and API wrappers between the two are provided to allow a user to seamlessly switch system service libraries from pthreads to hthreads and vice versa. After initial debugging on a standard workstation, the multithreaded application can be profiled or run through on-the-board testing, allowing the developer to identify which threads should be mapped into the reconfigurable fabric of the FPGA.

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 Fig. 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. Fig. 4 shows 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 3.7× speedup compared to the software version. The next two timing results highlight the benefits of exploiting coarse-grained parallelism within the FPGA. For two software threads running on a single CPU, each thread must be time-multiplexed, in a pseudo-concurrent fashion, 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, real parallelism is achieved. Two hardware threads running in parallel show 7.1× speedup when compared to two software threads time-multiplexing on the CPU. Additional experiments were conducted with varying numbers of threads mapped be-




```
hthread_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;
}
```

Fig. 3. Application-level code for creating hybrid threads.

tween software and hardware. With this example, we were limited to hosting only two DWT hardware threads due to size limitations of the FPGA being used. The speedup shown in the last two examples, is the speedup over using all software threads. This simple example illustrates the benefit of enabling coarse-grained 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 implementations of syscall run-time services for hardware-resident threads. In effect, the APIs provide consistent policies for threads running on both the CPU and within the FPGA. This includes the ability to support standard function call invocations, and the ability to create and pass abstract data types and pointers between threads in accordance with the semantics of each of the pthread APIs.

Fig. 5 shows a high-level description of our integrated compilation/synthesis tool flow. As shown in Fig. 5, we have augmented the standard GCC tool chain to produce a new hardware intermediate form (HIF) from which we then generate a VHDL implementation of a user-defined thread. The HIF is similar to standard static-single-assignment (SSA) intermediate forms, but in a slightly modified and controlled format to better serve as an intermediate target for VHDL generation.

The second challenge in extending hthreads across the HW/SW boundary was to create new hardware versions of system service libraries in support of standard threaded operations, but for hardware threads. To support our shared-memory threaded model, we created services to support the creation, control, and scheduling of threads executing in hardware. Additionally, we created services that allowed the independently executing hardware threads to synchronize and communicate with all other threads in the system by providing standard semaphore operations, as well as the ability to independently access global and local data stores. All services are invoked, even within hardware threads, from the original calls to hthreads APIs from the unmodified user-defined source program. Thus, our hthreads APIs eliminate the need to create unique interfaces for threads to interact across the CPU/FPGA interface, or hardware/software boundary. This allows the high-level application code to be portable between software and hardware boundaries, as well as different platforms as shown by our high-level design flow shown in Fig. 2.

III. Implementing the Hthreads Computational Model

Fig. 6 shows the hthreads run-time system components implemented in hardware. Hthreads migrates a thread manager, scheduler, mutex manager, and a CPU bypass interrupt scheduler (CBIS) into the hardware. This creates a virtual machine that abstracts the CPU/FPGA boundary and is accessible from either the FPGA or CPU. We have implemented the hthread run-time system on both the Xilinx Virtex-II Pro 7 and Virtex II-Pro 30 platform FPGAs. An embedded PowerPC 405 core and off-

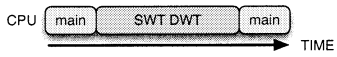
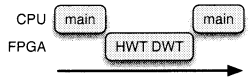
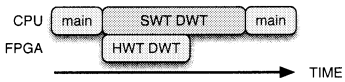
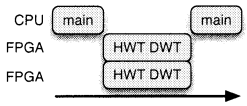
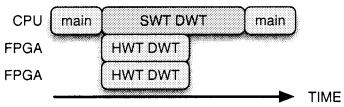
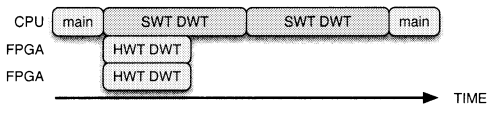
Threads	Thread Types	Total Time(ms)	Speed Up	Graphic
1	1 software	15.6		
	1 hardware	4.19	3.7x	
2	2 software	31.1		
	1 software, 1 hardware	16.5	1.9x	
	2 hardware	4.40	7.1x	
3	1 software, 2 hardware	16.6	2.8x	
4	2 software, 2 hardware	32.1	1.9x	

Fig. 4. DWT example run-time performance.

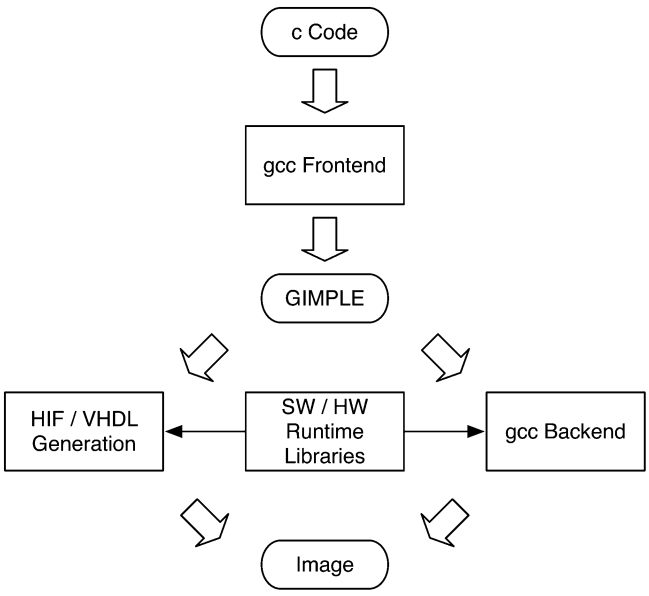


Fig. 5. Compilation/synthesis tool flow.

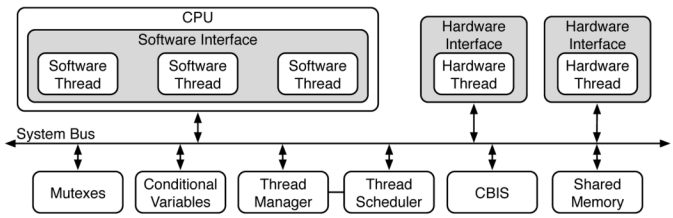


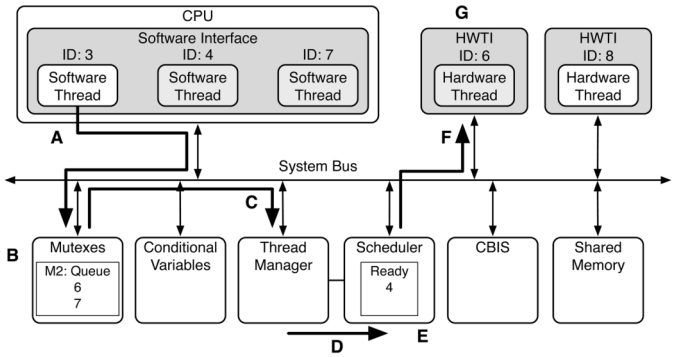
Fig. 6. Hthreads system block diagram.

chip memory are configured for the processor local bus (PLB) while all hthread cores and hardware threads are located on the on-chip peripheral bus (OPB) with a bridge in between each bus. All hthread cores and hardware threads use the vendor supplied IPIF interconnect to communicate with the OPB. The processor operates at 300 MHz and all other buses, cores, memory, and hardware threads operate at 100 MHz.

Migrating these OS and middleware services into hardware is required to create efficient mechanisms that are equally accessible from traditional system software as well as equivalent FPGA resident finite-state machines (FSMs). This has the additional benefit of bringing significant performance advantages to software threads through more efficient invocation and processing mechanisms [20], [21]. 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, Fig. 7 shows comparative timings

	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

Fig. 7. Hthreads performance summary.

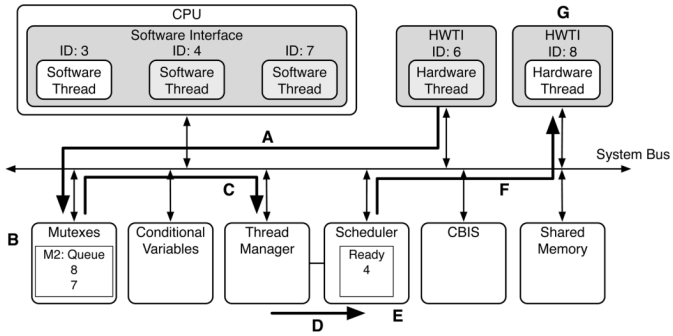


A: Software thread 3 unlocks mutex M2 by calling hthread_mutex_unlock(M2), which sends signal to Mutex Manager.
B: Mutex Manager inspects M2's queue and decides ID 6 will own mutex next.
C: Mutex Manager sends add_thread(6) to Thread Manager.
D: Thread Manager gives ID 6 to Scheduler to add to Ready to Run queue.
E: Scheduler inspects sched_param of thread ID 6 to find it is a HW thread, therefore does not add ID 6 to queue.
F: Scheduler instead sends RUN command to ID 6's HWTI.
G: Hardware thread 6 resumes execution, now owner of mutex M2.

Fig. 8. Hthreads Mutex unlock sequence (SW to HW).

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 O(1) scheduler is 24 clock cycles; a constant delay, independent of the number of threads in the ready-to-run queue [22]. The small amount of jitter seen in Fig. 7 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 during the unblocking of a thread due to the release of a mutex). Currently, the scheduler supports a single PowerPC core and zero to many hardware threads. We are currently modifying the original scheduler to support multiple processors including multiple embedded PowerPC cores and soft processors, such as the MicroBlaze, instantiated within the FPGA logic.

As a more complete example, the mutex_unlock() operation, illustrated in Figs. 8 and 9, shows the processing steps the hthreads system performs to release a mutex, make a scheduling decision, and resume the execution of a thread. In a traditional operating system, steps A–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



A: Hardware thread 6 (ID = 6) unlocks mutex M2 by calling hthread_mutex_unlock(M2), which sends signal to Mutex Manager.
B: Mutex Manager inspects M2's queue and decides ID 8 will own mutex next.
C: Mutex Manager sends add_thread(8) to Thread Manager.
D: Thread Manager gives ID 8 to Scheduler to add to Ready to Run queue.
E: Scheduler inspects sched_param of thread ID 8 to find it is a HW thread, therefore does not add ID 8 to queue.
F: Scheduler instead sends RUN command to ID 8's HWTI.
G: Hardware thread 8 resumes execution, now owner of mutex M2.
(**Note:** CPU remains uninterrupted by the hardware thread wakeup process)

Fig. 9. Hthreads Mutex unlock sequence (HW to HW).

if a new scheduling decision is required based on the queuing of a blocked thread. In hthreads, steps B–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 prescheduler speculative processing for hardware threads being unblocked. An example of how the hthreads system is capable of avoiding CPU-based speculative processing for operations that are solely HW-based is shown in Fig. 9 for completeness. In contrast, [23], [24] a multithreaded capability is reported that supports the creation and control of both hardware and software threads through Linux running on the CPU. This approach was taken to allow hardware threads to access data through Linux's existing virtual memory address space. Although convenient, this approach requires additional complexity within the hardware thread to maintain virtual address translation tables, and invokes the memory manager running on the CPU for page swapping through external interrupts, thus introducing jitter and overhead.

A. Hardware Thread Interface (HWTI) Abstract Interface

The HWTI makes the policies of the virtual machines computational model accessible to hardware threads. The HWTI implements the same functionality in HW-based state machines as the hthreads.h binary libraries for traditional software threads. An important detail to note is that every instantiated hardware thread has its own copy of the HWTI. This enables each hardware thread to execute autonomously and in parallel. This is in contrast with software threads that may only run pseudo-concurrently. A block diagram depicting the HWTI user and system interfaces, state machines, and internal registers is shown in Fig. 10. As shown in Fig. 11, the HWTI entity is linked in with the VHDL implementation of the user's code in a similar fashion to traditional linking of software library routines. The user VHDL code can be automatically generated from our C to VHDL translator, or written by a user as a core and linked with the HWTI. The HWTI component contains two interfaces: the

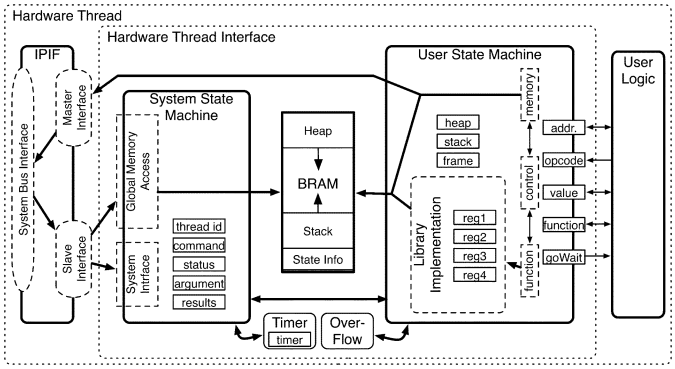


Fig. 10. HWTI block diagram.

```
entity user_logic_hwtul is
  port (
    clock : in std_logic;
    intrfc2thrd_address : in std_logic_vector(0 to 31);
    intrfc2thrd_value : in std_logic_vector(0 to 31);
    intrfc2thrd_function : in std_logic_vector(0 to 15);
    intrfc2thrd_goWait : in std_logic;

    thrd2intrfc_address : out std_logic_vector(0 to 31);
    thrd2intrfc_value : out std_logic_vector(0 to 31);
    thrd2intrfc_function : out std_logic_vector(0 to 15);
    thrd2intrfc_opcode : out std_logic_vector(0 to 5)
  );
end entity user_logic_hwtul;

ARCHITECTURE beh OF simple_thread IS
  ... constant & variable declaration ...
  HWTUL_STATE_PROCESS : process (clock, intrfc2thrd_goWait) is
  begin
    ... state transitions ...
  end process HWTUL_STATE_PROCESS;

  HWTUL_STATE_MACHINE : process (clock) is
  begin
    ... Default register assignments ...

    case current_state is
      when FUNCTION_RESET =>
        --Set default values
        thrd2intrfc_opcode <= OPCODE_NOOP;
        thrd2intrfc_address <= Z32;
        thrd2intrfc_value <= Z32;
        thrd2intrfc_function <= U_FUNCTION_START;

      when FUNCTION_START =>
        -- Push a return value
        thrd2intrfc_value <= x"ABCDEF01";
        thrd2intrfc_opcode <= OPCODE_PUSH;
        next_state <= WAIT_STATE;
        return_state_next <= _EXIT;

      when FUNCTION_EXIT =>
        --Immediatly exit
        thrd2intrfc_function <= FUNCTION_HTHREAD_EXIT;
        thrd2intrfc_value <= Z32(0 to 15) & U_FUNCTION_RESET;
        thrd2intrfc_opcode <= OPCODE_CALL;
        next_state <= WAIT_STATE;

      when WAIT_STATE =>
        next_state <= return_state;

    end case;
  end process HWTUL_STATE_MACHINE;
END beh;
```

Fig. 11. VHDL example code for exit thread.

HWTI system interface, used for interacting with other hthreads service components, and the HWTI user interface. The HWTI user interface serves as both an abstraction for system service calls and also provides indirect support for high-level language

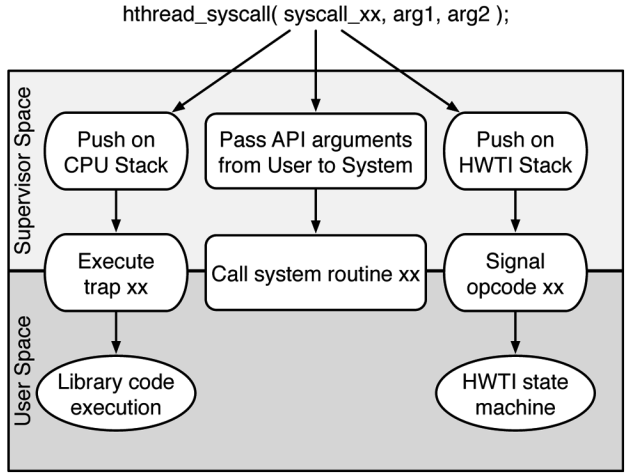


Fig. 12. System service policy and mechanisms.

constructs typically difficult to achieve in hardware. Furthermore, the user interface is designed such that the programmer does not have to be concerned with detailed information of the intended FPGA chip. The user-thread is thus portable between chips without modifications.

Fig. 12 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 Fig. 12, standard policy for invoking run-time services is achieved in 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 in order to invoke the run-time service. To achieve an analogous mechanism for hardware threads, the HWTI provides its own function call stack. In this manner, the user-thread pushes parameters to the HWTI and then calls the appropriate function by driving the appropriate opcode to the `function` register. Once received, the HWTI interprets the arguments and performs the system call on behalf of the user. The run-time services are provided as state machines in the HWTI instead of traditional binary software libraries. In a few exceptions when a system service cannot be implemented within the HWTI due to complexity, the HWTI signals a special system software thread to perform the operation on its behalf. The HWTI's remote procedure call (RPC) mechanism is similar to the callbacks described in [24]. Currently, remote procedure calls are used for allowing hardware threads to create and join other threads (either hardware or software). In addition to syscall services, the HWTI provides an abstraction for accessing and modifying memory in the forms of LOADs and STOREs. This policy allows the programmer to ignore the complexities of the bus-interconnect protocol. The LOAD and STORE operations work analogously to pointer dereferencing. For instance, during a LOAD operation, the user-thread drives the `address` register, with the address it wants to read, and the HWTI responds, after executing a bus transaction on behalf of the user-thread, with the value stored at that address in the `value` register.

An important and enabling feature that has been added to the HWTI is a globally distributed local memory. The globally distributed local memory, or "local memory" for short, is accessible both by the user-thread as well as all hthreads system cores. User-threads access the local memory using the same LOAD and STORE operations as for global memory with the HWTI accessing the appropriate memory bank based on the address range. The hthreads cores, including other hardware threads and the CPU, can access the local memory blocks using standard bus read/write transactions, since the local memory is made available as part of the HWTI's address range. On the Xilinx Virtex-II Pro FPGA, the mechanism used to implement these local memory blocks are the embedded dual-ported block RAMs (BRAMs) distributed throughout the reconfigurable fabric. One BRAM port is used by the system interface for system-core references, the second BRAM port is used by the user interface for user-thread references.

The immediate advantage of the local memory is that the user-thread has access to a fast memory without having to contend for a shared bus. As shown in Table III, the access time for both LOAD and STORE is an order of magnitude faster for local memory compared with global memory. Importantly, the local memory enables the HWTI to provide a function call stack for the user-thread. The HWTI's function call stack works analogously to software function call stacks. The HWTI maintains registers for a stack and frame pointer, pointing to its local memory instead of traditional global memory. During a call, the HWTI pushes the stack and frame pointer values onto the stack, the pointers are then appropriately incremented for the new function. All function parameters are passed by pushing the values onto the stack and retrieved by popping the values off the stack. The HWTI supports a PUSH and POP operation for this purpose. Finally, this being a key difference, instead of saving the contents of the program counter during a function call, as done on CPUs, the HWTI pushes the thread's return state onto the stack. The user-thread 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-thread passes a 16-bit variable representing the state of the FSM to return to after the function call is complete. The user-thread is responsible for mapping this variable to its return state when control is returned to the caller function. When the user-thread makes a CALL operation it specifies the function it wants to call through a 16-bit function code. Purposefully similar to the return state, this function code represents either the initial state of the called function within the user logic, or it represents a system library function supported by the HWTI. Although the HWTI does not dictate how the user-thread is designed, we have found that it is convenient to write the user-thread as a FSM. Indeed, this is the output of our C-to-VHDL translator. Consequently, the concept of an instruction address in a software program is encapsulated as a state within the user-thread's FSM.

The HWTI's function call stack has enabled analogous software function call convention support not only for function invocation but also variable declaration. 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 then increments its stack pointer the specified number of words. To read or write to the declared memory addresses the user-thread uses the READ and WRITE operations 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. The index number is added to the stack pointer to access the appropriate memory location within the BRAM. Since each declared variable is stored within the HWTI's local memory they each have an address within global memory. Using the ADDRESSOF operator, the user-thread may obtain a variables address. The HWTI calculates a variable's address using the index number, frame pointer, and base address of the hardware thread.

Finally, the HWTI's local memory has enabled support for dynamic memory allocation. The user-thread may call a light version of malloc and free, part of C's standard library, just as it would call an hthread system service. 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 (the heap) is preallocated in 8B, 32B, and 1024B segments at the top of the local memory address range. These sizes were selected to assist with the dynamic creation of mutexes, condition variables, and threads, common structures within the hthreads programming model. By preallocating memory the HWTI avoids implementing a defragmentation routine. When the user-thread 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-thread may request only a single segment of memory larger than 1024B. If the user-thread requests a memory space larger than the HWTI has available, the HWTI returns a null pointer. When the user-thread calls free, the HWTI marks the appropriate malloc state table entry as unused.

B. Hthread Compiler

To enable programmers to develop a HW/SW codesign without any knowledge of the targeted platform, programmers need a tool that will automatically and correctly translate their design from a standard high-level programming language to hardware threads that can be synthesized for the targeted chip. To create such a tool we are developing the HybridThreads compiler (HTC). The prototype implementation of HTC can successfully generate a custom hardware thread implemented in VHDL from a software thread specified in C. The resulting threads can then be integrated into the hthreads synthesis process and executed on an FPGA. The goal of HTC is to be able to translate *any* C thread conforming to the hthreads API into an independent and parallel hardware thread. Because HTC targets the HWTI, and the HWTI abstracts the system details from the user-thread, HTC may translate the C source code

without any concern for low-level hardware communication details. Using unmodified C and the hthreads API allows for a thread to easily migrate across the hardware/software boundary and requires no special effort for the developer other than simple compilation.

At a high level, HTC works by converting sequential C code into a state machine specified in behavioral VHDL. The generated state machine handles the user-thread's control flow and communication protocol with the HWTI. The HTC is partitioned into two independent modules: HIFGEN and HIF2VHDL. HIFGEN converts the programmer's C code into our new hardware intermediate form (HIF). HIFGEN starts with GIMPLE, GCC's intermediate form [25], and generates HIF. This allows us to take advantage of GCC's existing front end parsing and architecture independent optimizations. HIF is similar to traditional RISC type instructions, but modified to better serve as an intermediate form for VHDL translation. HIF also serves as an object file to allow for separate compilation of hardware threads. HIF2VHDL, which is a separate process from GCC, reads in the generated HIF files, analyzes the control flow graph of each function in the call graph of the hardware thread, and outputs a VHDL entity that implements the hardware thread. Because the HWTI already provides operating system, communication, memory management, and call stack services, HIF2VHDL does not have to generate these items specifically for the user-thread.

The current HTC supports a broad subset of the ANSI C standard including pointers, arrays, function calls, recursion, structs, all C control flow constructs including variable bounded while loops, as well as support for communication and synchronization with other threads in the system. The current HTC is limited to using 32-bit signed integers as the only primitive data type, and does not support function pointers, floating point operations, or the bulk of C's standard libraries. These limitations are associated with the current HTC being an early prototype and not due to the hardware thread model. We are currently developing a second version of HTC that will support all ANSI C functionality. However, in our current prototype hardware threads are not optimized during compilation from HIF to VHDL and, therefore, suffer from known inefficiencies introduced during the compilation process. Once we complete support for ANSI C, we will focus on optimizing the hardware thread compilation process using both traditional compiler optimizations and by researching new optimizations that are specific to our hardware thread model. The long-term goal of HTC is to create a system that allows for C to be compiled into a hardware thread as easily and efficiently as it can be compiled into software.

IV. Results

To validate hthread system call functionality, relevant conformance, and stress tests from the open POSIX test suite [26] were adapted for hthreads and run in both software and hardware. These tests are designed to ensure compliance with the pthread model as well as detect any long-term use errors. Both interfaces, the software library and the HWTI respectively, passed all adapted test cases. Successfully completing these tests was important, as it showed the hardware/software boundary may be abstracted through a shared memory multithreaded programming model.

TABLE I
Performance, Total Resources, and Speed up Equivalent Software Implementation, for Selected Algorithms Hand Translated From C to VHDL. Data Set in Each Algorithm was an Array of 1000 Integers

Algorithm	Slices	Execution Time	Speedup over CPU
quicksort	2770	8.89ms	3.9
Harr DWT	1385	.918ms	3.5
Huffman encoding	2479	10.5ms	3.9
IDEA encryption	2616	4.06ms	11.6

TABLE II
Performance and Size of Selected HWTI System Calls

Function Call (hthread_)	Execution Time	Size (slice count)
create	160μs	401
join	130μs	356
self	.01μs	13
equal	.05μs	69
exit	.02μs	0
mutex_lock	.36μs	73
mutex_trylock	.36μs	66
mutex_unlock	.36μs	54
cond_signal	.41μs	115
cond_broadcast	.41μs	117
cond_wait	1.02μs	197

To demonstrate the HWTI's ability to support HLL semantics, a number of common application algorithms were translated, by hand, from C to VHDL, targeting the HWTI. The algorithms, quicksort, Harr discrete wavelet transformation (DWT), Huffman encoding, and the IDEA encryption algorithm, are listed in Table I, along with their implementation size, performance, and speedup. Speedup compares the hardware core with the original software code compiled to run on the embedded PowerPC core. In each instance, the hardware thread was operating on data from global memory, and using its local memory for temporary variable declaration. Note especially that quicksort, a recursive algorithm, successfully used the HWTI's function call stack to correctly maintain its state between recursive calls. In general, hardware threads individually achieve about a 3.5$\times$ or more speedup. Additional speedups may be obtain with multiple threads are concurrently working.

The base HWTI is implemented in 1019 slices. This includes logic for the standard vendor supplied IP bus interface (IPIF), 32 kB of BRAM for the local memory, a function call stack, and a minimal user logic thread that immediately exits following a RUN command (abbreviated in Fig. 11). The 1019 slices represent 7% of all slices on our Xilinx Virtex-II Pro FPGA (XC2VP30). Size and performance results for selected hthread system calls, implemented in the HWTI are shown in Table II. The size of each system call is an estimate based on the difference in size between a hardware thread that immediately exits and a hardware thread that calls the selected function and then exits. Note by this definition, the size of `hthread_exit` is 0 slices. The performance of most HWTI supported system calls is largely dependent on the number of bus operations the HWTI has to perform to complete the call. `hthread_mutex_lock`, `hthread_mutex_trylock`, `hthread_mutex_unlock`, `hthread_cond_signal`, and

TABLE III
PERFORMANCE OF HWTI OPERATIONS

Operation	Execution Time
POP	.05μs
PUSH	.01μs
LOAD (local memory)	.03μs
LOAD (global memory)	.51μs
STORE (local memory)	.01μs
STORE (global memory)	.28μs
DECLARE	.01μs
READ	.03μs
WRITE	.01μs
ADDRESSOF	.01μs
CALL (user defined)	.03μs
CALL (system function)	varies
RETURN	.07μs

`hthread_cond_broadcast` require only one bus operation, and thus the execution time of each is around 40 clock cycles. `hthread_cond_wait` requires three bus operations and consequently takes roughly 3× as long. `hthread_self`, `hthread_equal`, and `hthread_exit` do not require any bus operations. `hthread_create` and `hthread_join` are not directly implemented within the HWTI but instead implemented using the remote procedure call (RPC). Although the RPC mechanism allows the HWTI to support any function, their performance is comparatively poor. Finally, timing results for HWTI operations are listed in Table III.

V. 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 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. Importantly, our ability to allow multiple execution threads to exist within the FPGA provides a new mechanism to exploit the full potential of the FPGA. Hthreads, as described in this paper, is a usable OS that has been fully implemented in synthesizable form, and all experiments described within this paper have been executed on Xilinx Virtex-II Pro FPGAs. More information on hthreads can be accessed at www.ittc.ku.edu/hybridthreads.

REFERENCES

[1] Xilinx, San Jose, CA, "Programmable logic devices," Sept. 10, 2007 [Online]. Available: http://www.xilinx.com/

[2] G. Almasi and A. Gottlieb, *Highly Parallel Computing*. Redwood City, CA: Benjamin Cummings, 1994.

[3] J. C. Browne, "Parallel architectures for computer systems," *IEEE Computer*, vol. 37, no. 5, pp. 83–87, May 1984.

[4] A. S. Tanenbaum, *Structured Computer Organization*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[5] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.

[6] S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Des. Test Comput.*, vol. 23, no. 5, pp. 375–386, Sep. 2006.

[7] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proc. Int. Conf. VLSI Des.*, 2003, pp. 461–466.

[8] Celoxica, Abbingdon, Oxfordshire, U.K., "Celoxica homepage," [Online]. Available: www.celoxica.com

[9] J. Park, P. C. Diniz, and K. S. Shayee, "Performance and area modeling of complete FPGA designs in the presence of loop transformations," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1420–1435, Nov. 2004.

[10] B. So, M. Hall, and P. Diniz, "A compiler approach to design space exploration in FPGA-based sysems," in *Proc. ACM Conf. Program. Lang. Des. Implementation*, 2002, pp. 165–176.

[11] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Bev-eridge, M. Chawathe, and C. Ross, "High-level language abstraction for reconfigurable computing," *IEEE Computer*, vol. 36, no. 8, pp. 63–69, Aug. 2003.

[12] T. Callahan, "Automatic compilation of C for hybrid reconfigurable architectures" Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. California Berkeley, Berkeley, 2002 [Online]. Available: http://brass.cs.berkeley.edu/documents/tjc.thesis.htrnl

[13] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh, "PRISM-II compiler and architecture," in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.*, 1993, pp. 9–16.

[14] P. Athanas and H. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11–18, Mar. 1993.

[15] M. Gokhale and R. Minnich, "FPGA computing in a data parallel C," in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.*, 1994, pp. 94–101.

[16] J. Hauser and J. Wawrzynek, "GARP: A MIPS processor with a reconfigurable coprocessor," in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.*, 1997, pp. 12–21.

[17] ImpulseC, Kirkland, WA, "ImpulseC homepage," (2007). [Online]. Available: www.impulsec.com

[18] SystemC, "SystemC homepage," [Online]. Available: www.systemc.org

[19] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access," in *Proc. 14th Ann. Conf. Field-Program. Custom Comput. Mach.*, 2006, pp. 45–56.

[20] W. Peck, J. Agron, D. Andrews, M. Finley, and E. Komp, "Hardware/software co-design of operating systems for thread management and scheduling," in *Proc. 25th IEEE Int. Real-Time Syst. Symp., Works Progress Session (RTSS WIP)*, 2004, pp. 48–51.

[21] J. Agron, D. Andrews, M. Finley, E. Komp, and W. Peck, "FPGA implementation of a priority scheduler module," in *Proc. 25th IEEE Int. Real-Time Syst. Symp., Works Progress Session (RTSS WIP)*, 2004, pp. 116–119.

[22] 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 *Proc. 10th IEEE Int. Conf. Emerging Technol. Factory Autom.*, 2005, p. 8.

[23] M. Vuletic, L. Possi, and P. Ienne, "Virtual memory window for application-specifc reconfigurable coprocessors," in *Proc. 41st Ann. Conf. Des. Autom.*, 2004, pp. 948–953.

[24] M. Vuletic, L. Pozzi, and P. Ienne, "Seamless hardware software integration in reconfigurable computing systems," *IEEE Des. Test Comput.*, vol. 22, no. 2, pp. 102–113, Mar. 2005.

[25] J. Merrill, "GENERIC and GIMPLE: A new tree representation for entire functions," in *Proc. GCC Developers Summit*, 2003, pp. 171–180.

[26] POSIX, "Open POSIX Test Suite," [Online]. Available: http://posixtest.sourceforge.net/

David Andrews (SM'01) received the B.S.E.E. and M.S.E.E. degrees from the University of Missouri-Columbia, Columbia, and the Ph.D. in computer science from Syracuse University, Syracuse, NY, in 1992.

He is a Professor with the Electrical Engineering and Computer Science Department, University of Kansas, Lawrence. Prior to joining the faculty at the University of Kansas in 2000, he was with the faculty of the University of Arkansas, Fayetteville, and was a Senior Systems Engineer with Underwater Systems Division, General Electric, Syracuse, NY, and a System Architect with the Electronics Laboratory and the Advanced Technologies Laboratory. His research interests include embedded systems architectures, parallel, and reconfigurable computing.

Ron Sass (M'99) received the B.S. degree in computer science engineering from the University of Toledo, Toledo, OH, in 1989, and the M.S. and Ph.D. degrees in computer science from Michigan State University, Lansing, in 1992 and 1999, respectively.

He is an Associate Professor with the Department of Electrical and Computer Engineering, University of North Carolina, Charlotte (UNC-Charlotte). Prior to joining the faculty of UNC-Charlotte in 2006, he was with the faculty of the University of Kansas, Lawrence, from 2004 to 2006, and with Clemson University, Clemson, SC, from 1997 to 2004. His research interests include reconfigurable computing, high-performance computing, and networking.

Dr. Sass is a member of the Association for Computing Machinery (ACM) and the American Association for the Advancement of Science (AAAS).

Erik Anderson (S'07) received the B.S. degree in computer science from the University of Kentucky, Lexington, in 1997, and the Ph.D. degree in electrical engineering from the University of Kansas, Lawrence, in 2007.

He is a Research Scientist with the Information Science Institute, University of Southern California, Los Angeles. His research interests include reconfigurable computing and hardware/software codesign.

Jason Agron (S'08) received the B.S. and M.S. degrees in computer engineering from the University of Kansas, Lawrence, in 2004 and 2006, respectively, where he is currently pursuing the Ph.D. degree in computer science.

He is a Research Assistant with the Information and Telecommunication Technology Center, University of Kansas. His primary research interests include hardware/software codesign, real-time operating systems, and parallel programming models.

Wesley Peck received the B.S. in computer science from the University of Kansas, Lawrence, in 2003, where he is currently pursuing the Ph.D. degree in computer science.

He is a Research Assistant with the Information and Telecommunication Technology Center, University of Kansas. His primary research interests include algorithm design, hardware/software codesign, and real-time operating systems.

Jim Stevens received the B.S. degree in computer engineering from the University of Kansas, Lawrence, in 2006, where he is currently pursuing the Ph.D. degree in computer science.

He is a Research Assistant with the Information and Telecommunication Technology Center, University of Kansas. His primary research interests include compilation for reconfigurable architectures and hardware/software codesign.

Fabrice Baijot received the B.S. degree in computer engineering from the University of Kansas, Lawrence, in 2006, where he is currently pursuing the Ph.D. degree in computer science.

He is a Research Assistant with the Information and Telecommunication Technology Center, University of Kansas. His primary research interests include compiler optimizations, hardware/software codesign, and software to hardware translation.

Ed Komp received the B.A. degree in mathematics and the M.S. degree in computer science from the University of Kansas, Lawrence, in 1976 and 1979, respectively.

After over 15 years designing, implementing, and managing commercial software development, he became a Research Engineer with the Information and Telecommunications Technology Center, University of Kansas. His primary research interests include specialized computer language design for application specific domains, functional programming, software development environments, and networking.