Enabling a Uniform Programming Model Across the Software/Hardware Boundary

Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, Ed Komp, Ron Sass, David Andrews
Information and Telecommunication Technology Center - University of Kansas
2335 Irving Hill Road, Lawrence, KS
{eanderso,jagron,peckw,jstevens,bricefab,komp,rsass,dandrews}@ittc.ku.edu

Abstract

In this paper, we present hthreads, a unifying programming model for specifying application threads running within a hybrid CPU/FPGA system. Threads are specified from a single pthreads 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. 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, and allows passing of abstract data types between hardware and software threads. Thus the hardware thread interface provides an abstract, platform independent compilation target that enables thread and instruction-level parallelism across the software/hardware boundary.

1 Introduction

In this paper, we present hthreads, a unifying programming model for specifying application threads running within an hybrid CPU/FPGA system. Hthreads provides unique capabilities within the reconfigurable computing community by enabling concurrent execution of threads specified through a set of pthreads compatible API library routines to be automatically compiled, synthesized, and seamlessly executed on a CPU/FPGA hybrid chip. The hthreads programming model extends the pthreads high-level programming policies to span the hardware/software boundary. This is accomplished by extending standard shared-memory synchronization primitives and system services to hardware threads, and by providing the same abstract-interface support to all threads regardless of their location in the system. A description of the hthreads hardware/software co-designed system services can be found in [2, 3].

In the hthreads working model, programmers specify their application as a set of concurrent threads using the pthreads model semantics. In much the same way as a programmer does not care which processor an individual pthread may be scheduled, the distinction as to where an hthread is running (either hardware or software) is also irrelevant. One subtle but important capability the hthreads system offers is the ability to execute each thread in parallel within the FPGA. As pointed out in [8], this true parallelism can provide significant performance increases compared to the pseudo-concurrency that is achieved by time-multiplexing multiple threads on a shared CPU.

The hthreads system provides portability through library system services that encapsulate and link unique mechanisms to our pthreads compatible API's. The hthreads libraries hide all low-level platform-specific implementation details from the user, similar to the encapsulation of processor specific code within standard pthreads libraries. The hthreads system also opens the possibility of creating custom thread SMP architectures within the FPGA to software programmers without requiring any hardware design skills or knowledge of platform-specific implementation details. Furthermore, hthreads does so without any additional overhead or interrupts to start, maintain, or end threads running on the FPGA. In past hybrid architectures [7, 25, 11] the notion of a model with a single-thread of execution forces hardware computations to be slaves under control of the CPU: therefore not providing a truly uniform view of hybrid computations. In hthreads, all threads (hardware and software based) are autonomous computations: they are free to synchronize and communicate with any other thread in the system through the use of standard high-level APIs that are called using uniform interfaces.

The remainder of this paper is outlined as follows. In the next section we discuss the need for a complete abstract programming model for reconfigurable computing. The following section then discusses the implementation and operation of the hardware thread interface (HWTI) and how it provides seamless interaction and portability between hardware and software threads within our hthreads multithreaded programming systems. Our prototype hthreads design flow is then presented, followed by an example illustrating the use of our hthreads APIs. We conclude with a discussion of our future work.

2 Abstract Programming Models

Reconfigurable computing as a discipline has now been in existence for over a decade. Since its inception, researchers have been investigating augmentations to existing languages and hardware compilation techniques to allow programmers and system designers to access the potential of the reconfigurable fabric [10, 26, 17, 20, 16, 6, 24, 5, 9, 11]. In most instances, these approaches assume a computational model consisting of a single control thread hosted on the CPU that controls the computations that occur within the FPGA. Conceptually, these approaches are following a development path similar to that which occurred in the 1980's and 1990's 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. Parallelism is also exploited through loop analysis. Although these approaches bring advancements in *accessibility*, they do so at the cost of *portablilty* promoting detailed knowledge of the underlying architecture 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 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. The multithreaded programming model is also familiar within 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 library released with Linux and Unix. Additional low-level hardware support for multithreading is also standard within CPU's as evidenced by Intel's hyperthreading technology [28].

3 Targetting an Abstract Hardware Thread Interface

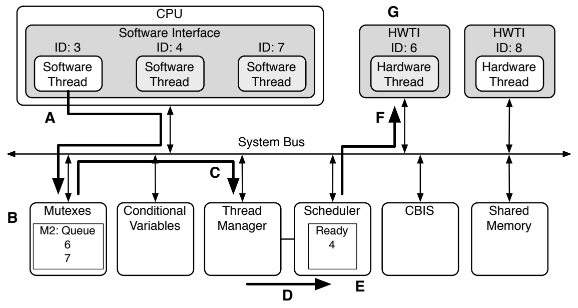
Although conceptually simple, extending high-level programming models across *reconfigurable systems* faces two key challenges. First, design flows must be modified to support the generation of the API interfaces specified within the programming model to provide consistent policies between components running on both the CPU and within the FPGA. This includes sharing abstract data types, and pointers in accordance with the semantics of each API. Although impressive, existing high-level synthesis tools such as Handel-C [26], ImpulseC [27], and SystemC [29] still require user inserted low-level interface ports that represent the system bus for passing data and control into and out of hardware components. These low-level interface ports additionally require the user to recast any abstract data types created within the original C program into a series of bit-width specified integers.

Second, as no standard system service routines or register sets exist within the reconfigurable fabric, new hardware component system service libraries must be created to support abstract API operations from and to hardware threads. In the case of the shared-memory thread model, this includes the ability to create, control, and schedule threads executing in either hardware or software as independent executing components. The independent components should be able to synchronize using standard semaphore operations, and independently access global data. Although at first glance mirroring traditional run-time software thread system services within hardware components appears daunting, it instead provides new opportunities to create more efficient globally accessible hardware/software co-designed shared services for both hardware and software resident threads. Migrating classic software run-time services into hardware can significantly reduce operating system overhead by eliminating the need to traverse vertically through software protocol stacks. Instead of vertical protocol stacks, the system services can be efficiently implemented within concurrent finite state machines accessible with single load and store operations.

In hthreads, we have migrated key system services including 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 [18, 1]. First, invocation mechanisms for accessing the system services are no longer based on inefficient traversal of a hierarchical protocol stack on the CPU, but instead are achieved through lightweight atomic load and store operations. Second, speculative and variable execution performed within such system services such as the scheduler are eliminated. As an example, Figure 1 shows comparative timings for executing typical scheduler services for between 2 and 250 active software threads running on 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 is 13 clock cycles independent of the number of threads in the ready-to-run queue [4]. The small amount of jitter seen in Figure 1 is due to cache misses when swapping 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 in the ready-to-run queue is higher priority than the thread running on the CPU. This is in contrast to existing software schedulers that must be invoked and ran to consider if an event may or may not trigger a true scheduling decision such as the release of a mutex.

	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 1. hthread 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 knows ID 6 is a hardware thread, 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.

Figure 2. hthread Mutex Unlock Sequence

As a more complete example, the mutex_unlock() operation illustrated in Figure 2 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 queueing 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 [22, 23] a multithreaded capability is reported that supports the creation 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.

Although unique in their standalone abilities to provide significant performance advantages to traditional software based threads, our hardware/software co-designed system services were developed to support the seamless interaction of application threads running within and across the CPU/FPGA boundary. To enable this abstraction, we created the hardware thread abstraction layer, and specifically the hardware thread interface (HWTI). The HWTI is a hardware thread's version of a software syscall interface layer but using the more efficient load/store mechanisms in place of hierarchical protocol stacks. As such, the HWTI interfaces into the co-designed system services using the same invocation protocol running on the CPU. Further, the HWTI provides the same hthread library calls available to software threads. This enables seamless interactions for all threads running in either hardware or software. The HWTI is thus a target that can be accessed through an analogous syscall interface for hardware threads, whether automatically synthesized through HLL to HDL compilation, or by developers wishing to hand write threads in VHDL.

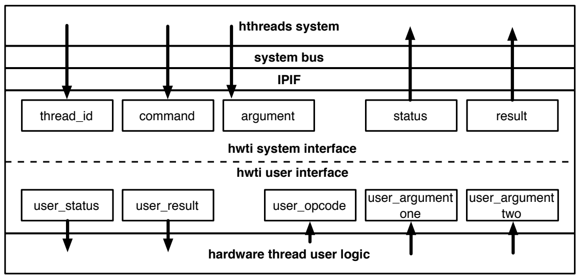


Figure 3. HWTI Block Diagram

3.1 Hardware Thread Interface (HWTI)

A block diagram of the hardware thread interface component is shown in Figure 3. The HWTI component is written in VHDL as an entity and is linked in with the user code in a similar fashion to system call software routines. The HWTI component contains two interfaces; the HWTI system interface for interacting with other hthreads service components, and the HWTI user interface for supporting system service calls from the user thread. The HWTI also contains three state machines not shown in Figure 3. The first two state machines control and monitor the system and user interfaces. The third state machine controls the interactions between the first two state machines and provides the system call mechanism.

3.1.1 System Interface

The system interface shown in the top of Figure 3 is composed of address-mapped registers accessible across the bus. This interface is hidden from the user threads and is only accessed by other hthread system service components. This interface is composed of five memory mapped registers: `thread_id`, `command`, `status`, `argument`, and `result`. These registers form the equivalent of thread context for a hardware thread similar to the data registers, program pointer, and stack pointer for software threads. The `thread_id` register is written with the thread ID assigned by the system when the create_thread() API is executed in software. This allows a statically synthesized and mapped thread to be created and terminated, and then re-created dynamically from within a parent thread in accordance with the semantics of software threads. The user thread may obtain its own thread ID by calling the hthread_self() API, similar to the pthread's pthread_self().

The `command` register is used by the scheduler to control the execution of the thread. The allowable commands are RUN and RESET. Writing a RUN into the `command` register is similar to loading the program counter with an address. Whereas in a microprocessor loading the program counter initiates the fetching, decoding, and execution of the instruction specified at the address, issuing a RUN command causes the controlling state machine to transition the execution of the user thread from an idle state into an active state. The scheduler will issue a RUN command for either the initial starting of the thread, or in response to the unblocking of a semaphore. In most cases, the HWTI will immediately transition the user logic into the RUN state, by issuing a similar RUN command into the user thread's visible `user_status` register. A RESET command returns the HWTI and the user logic to their initial unused states.

The `status` register is readable by the system and used primarily for debugging purposes to monitor a thread's current execution state. It reports to the system one of five states, NOT_USED, USED, RUNNING, BLOCKED, or EXITED. NOT_USED is the state after system start up, or a RESET command. USED, refers to the state after the thread id has been assigned. The HWTI is RUNNING after a RUN command is issued to the `command` register. BLOCKED is a specific case of RUNNING, namely when the HWTI is waiting to own a mutex. Finally, EXITED is when the user logic is done executing.

The `argument` register is used to allow the system to pass one argument into the thread when the thread_create() API is executed from a SW thread. This register provides consistency with the pthreads protocol, which only allows a single argument to be passed into a thread. The `argument` register can hold either a data value or more traditionally a pointer to a structure in global memory. When a RUN command is issued to start the execution of the thread, the value contained in the `argument` register is passed into the `user_result_register` for the user thread before the `user_status` is changed to RUN.

The `result` register allows the user thread to return results when a joinable thread executes the thread_exit() API. For consistency with pthreads, the result value should be a pointer, however this is not required.

3.1.2 User Interface

The user interface registers, `user_status`, `user_result`, `user_opcode`, `user_argument_one`, and `user_argument_two`, represented at the bottom of Figure 3 are visible only to the hardware thread's user logic. These registers are not memory mapped or accessible by other hthreads system services. The operation of these registers form the functional equivalence of executing a syscall for software threads running on a standard CPU. General system services are requested through the `user_opcode` register with parameter data passed in using the `user_argument` registers. The analogy between executing a syscall and interfacing the `user_opcode` register is intentional and encapsulates the differences

```
void * simple_thread( void * arg ) {
  int bound = (int) arg;
  int x=0;
  int i;

  for ( i=0; i<bound; i++ ) {
    x++;
  }

  return x;
}
```

Figure 4. Simple Thread Function Source Code

between software and hardware interfaces within a syscall. Encapsulating the platform specific differences entirely within the syscall allows portability of the source code, any differences are contained within the platform specific HWTI code generation.

As an example, Figure 4 shows the C code for a simple thread. The functional equivalent, in VHDL, targeting the HWTI is partially shown in Figure 5. Note that this example is kept simple for demonstrative purposes. The HWTI contains the pre-defined interface signals `intrfc2thrd_status`, `intrfc2_thrd_result`, `thrd2intrfc_opcode`, `thrd2intrfc_argument1`, `thrd2intrfc_argument2`. These signals link the HWTI user registers into the user thread's VHDL code.

The `user_status` register is updated by the HWTI and controls the execution of the user thread. The allowable status values are RESET, RUN, and ACK. As seen in the simple_thread example, Figure 5, the state machine generated to control the user thread inputs the `intrfc2thrd_status` signal from the `command` register to sequence the execution of the thread. The state machine controller waits in the reset state until a RUN command is issued. Once issued, the user thread begins autonomous execution.

The `user_opcode`, and `user_argument` registers provide the syscall interface for the user thread. Specific syscall functionality is specified by a unique opcode shown in Table 1. For example when the 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. 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 `user_result` register has two uses; first to pass initial arguments into the user thread and second to return values back in response to a service request. As an example, on a LOAD operation from global DRAM, the data results

```
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 ...

                when final0 =>
                    -- call exit
                    thrd2intrfc_opcode <= OPCODE_HTHREAD_EXIT;
                    current <= final1;
                when final1 =>
                    -- wait for exit ack
                    IF intrfc2thrd_status = USER_STATUS_ACK THEN
                        thrd2intrfc_opcode <= OPCODE_NOOP;
                        current <= final2;
                    ELSE
                        current <= final1;
                    END IF;
                when final2 =>
                    -- idle until reset event
                    current <= final2;
                END CASE;
        END IF;
    END IF;
END PROCESS;
END beh;
```

Figure 5. Automatically Generated VHDL Code For Simple Thread

will be returned to the thread in this register.

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 slices represent 2% of all slices on our Virtex II Pro (XC2VP30). Timing for each operation is listed in Table 2 and was recorded based on a cycle accurate simulation.

4 Hthreads Design Flow

Figure 6 shows a high level overview of the hthread's design flow. A detailed description of the complete process is beyond the scope of this paper. However, we provide a short description of key points appropriate for generating hardware and software threads, and linking in the appropriate system services. As seen in Figure 6 the complete application program is first translated into an architecture independent intermediate form. Application threads that are to be resident in hardware are then translated into a new hardware intermediate form (HIF). The HIF, while not shown

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

Table 1. HWTI System Calls

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	14	Time user thread issues LOAD opcode to time HWTI returns `user_status` to RUN including bus transaction
STORE	33	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

Table 2. HWTI Operation Timing

in this high-level design flow, is a more suitable intermediate representation for us to work with in performing certain finite-state machine analysis and optimizations of user applications before being translated into VHDL. The VHDL representation of a thread, or its user logic, is then linked with other HW libraries in order to include system call and other service component functionality. Once the inclusion of HW libraries is complete, the newly created HW threads are able to run through vendor-supplied synthesis tool flows.

The traditional software tool flow can execute in parallel with the automatic generation of hardware-based threads. In this flow, threads are linked in with the hthreads libraries. Once the SW libraries have been included with the application threads, traditional software compilation can occur to produced an executable to run on the PowerPC processor embedded within the Virtex II Pro FPGA. All of an application's threads are passed through the traditional software compilation flow to allow for threads to be created and implemented in either software or hardware.

The software and hardware tool flows operate in the exact same manner; only their targets differ. The software tool flow compiles to an assembly language target, and uses the hthreads libraries (written in C) as system service mechanisms. The hardware tool flow compiles to a VHDL target, and uses the HW libraries which include system component descriptions and the HWTI (written in VHDL) as system service mechanisms.

Once all threads, both hardware and software, have been compiled, the entire system can be integrated by composing the standard hthread's hardware/software co-designed system cores, the individual hardware thread implementations, hardware and software library files, as well as the software thread implementations into a single Xilinx Platform Studio (XPS) system. This system is built using the standard Xilinx EDK toolset, which is used to create both a full bitstream and executable file to be downloaded into the recon-

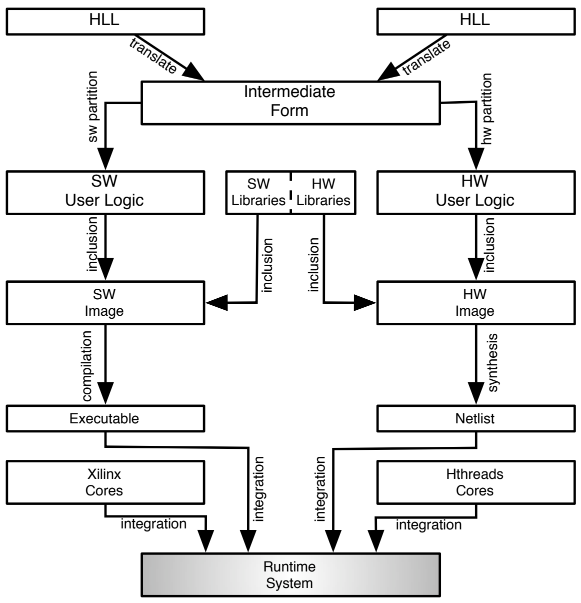


Figure 6. HW/SW Compilation

figurable platform for execution.

High-level synthesis (HLS) techniques used in the transformation and partitioning of HLLs to hardware circuits has been a hot research topic for many years. The hthreads compiler is not being designed in order to provide unparalleled performance of hardware implementations of computations described using HLLs. It is being developed in order to closely target the interfaces provided by the hthreads runtime system so that multithreaded programs can take greatest advantage of the resources that hybrid CPU/FPGA systems offer. This allows the programmer to have a truly uniform view of computations in the hybrid system, allowing the user to focus on how to express their program in a multithreaded environment rather than focusing on interfaces between hardware and software computations. Additionally, the hthreads compiler will not focus on elaborate exploitation of instruction-level parallelism (ILP) within a thread because the hthreads system itself allows for the exploitation of coarse-grained parallelism through actual concurrent execution of threads and OS services. The hthreads runtime system provides uniform, low overhead and low jitter OS services to threads in a hybrid environment [4], while the hthreads compiler enables the translation of threads expressed in HLLs to become autonomous circuits able to execute in parallel within the run-time system. Together, the run-time system and compiler provide the ability to exploit both thread-level and instruction-level parallelism in a hybrid CPU/FPGA system while abstracting the interface between hardware and software computations through the use of a uniform view of threads.

```
struct matrix {
  Huint size;
  Huint index;
  Huint * X;
  Huint * Y;
  Huint * Z;
};

hthread_mutex_t matrixMutex;

int main( int argc, char *argv[] ) {
  hthread_t       hwThread, swThread;
  hthread_attr_t  hwAttr, swAttr;
  struct matrix   matrixData;
  Huint i;

  // Initialize the hybrid threads system
  hthread_init();
  hthread_mutex_init( &matrixMutex, NULL );

  // Initialize the attributes for the threads
  hthread_attr_init( &hwAttr );
  hthread_attr_init( &swAttr );

  // Hardware thread attributes need the HWT’s base address
  hthread_attr_sethardware( &hwAttr, HWT_BASEADDR );
  // Software thread uses default attributes.

  // Initialize matrixData
  initializeData( &matrixData );

  hthread_create( &hwThread, &hwAttr, NULL, (void*)(&matrixData) );
  hthread_create( &swThread, &swAttr, matrixAdd, (void*)(&matrixData) );

  // Wait for the threads to exit
  hthread_join( hwThread, NULL );
  hthread_join( swThread, NULL );

  // Clean up the attribute structure
  hthread_attr_destroy( &hwAttr );
  hthread_attr_destroy( &swAttr );

  //Matrix add is now complete
  return 1;
}
```

Figure 7. Matrix Add main Function

5 Hthreads Hardware/Software Thread Example

To illustrate the use of our multithreaded programming model, consider the matrix add process shown in Figure 7. This example is kept simple to demonstrate two threads, one hardware and one software, that work together, using equivalent synchronization policies, to perform a matrix add. The complete software thread source code is show in Figure 8. The VHDL for this example is not shown as it is over 400 lines of source code. The VHDL code for this example was written by hand and not generated by our C to VHDL translator (see Future Works section for status of our application code translator) as this example was initially created to test the HWTI entity interfaces. The VHDL code contains the same system calls and interfaces as the software version shown in Figure 8. At the start of execution, each thread reads the passed in struct matrix, which contains information for the size and index of the arrays, as well as the address location of the X, Y, and Z arrays. A global mutex is used to protect the index variable, which identifies the element within the X and Y arrays that the thread is to operate on.

The main thread first initializes the hthread system by calling hthread_init(), followed by the initialization of matrixMutex and the attributes for hwThread and swThread.

In our current version, the user software code must designate a thread as a hardware thread by specifying the hardware threads's base address. The base address is assigned manually through Xilinx's EDK tool. Once the main thread initializes the matrixData, both threads are created with the main thread passing a pointer to matrixData.

To create the hwThread, the main thread calls the Thread Manager's hthread_create function in order to create a joinable thread. All Thread Manager functions are implemented as LOAD operations over the system bus. Upon return, the system sets the hwThread ID by writing to the HWTI's `thread_id` register, sets the address of matrixData to the HWTI's `argument` register, and passes the base address of hwThread to the Scheduler. The system then starts hwThread by calling the Thread Manager's add_thread() function. This in turn adds the hwThread ID to the schedulers ready-to-run queue. The Scheduler, recognizing it as a hardware thread, issues a RUN command to the hwThread's HWTI `command` register. The HWTI in turn, updates the `user_result` register to the value of the `argument` register, which is the address of matrixData previously set by the system. Finally, the HWTI allows the hardware thread's user logic to begin execution by changing the status of the `user_status` register to RUN from RESET.

The software thread, swThread, is started in similar fashion. The system once again calls the Thread Manager's hthread_create function to create a joinable thread. Upon return, the system initializes the new thread's stack, and passes the thread's priority to the Scheduler. The system then readies the software thread to run by again calling the Thread Manager's add_thread() call. This in turn add's the swThread's ID to the ready-to-run queue, and forces the Scheduler to re-evaluate the next_thread scheduling decision. When the next context-switch occurs, the running thread is context-switched out in favor of the `next_thread`. In this example's case, the context-switch occurs in conjunction with the main thread calling hthread_join. The swThread is then chosen as the highest priority thread in the ready-to-run queue and will begin executing on the CPU.

The join operation, from a system's perspective is identical, for both a hardware and software thread. To join on the hwThread, the main thread calls the Thread Manager's join_thread() function, encoding the hwThread ID, in the address lines. The Thread Manager checks to see if the child thread has exited, if not (as is likely the case), it returns a signal to the CPU to tell it to perform a context-switch. When the hwThread is done executing it sends an exit_thread() call to the Thread Manager. The Thread Manager responds by adding the main thread ID, the parent thread to hwThread, to the ready-to-run queue in the Scheduler. When the main thread is running on the CPU again, an almost identical process happens for joining on the swThread. Since the swThread and hwThread operate on the same data, they will end at nearly the same time. After the main thread joins on both, the sum of the X and Y matrix is stored in the Z matrix.

This example illustrates how the operations for creating, joining communicating, and exiting are either identical or analogous between threads running in software and threads running in hardware. Figure 9 shows the timing reports for this example. For comparative purposes we performed the matrix add operations with combinations of between one to four hardware and software threads. Figure 9 shows the performances advantages gained by the hybrid thread system. First, not surprisingly, we see that the hardware thread outperforms the software thread running on the general purpose CPU. For an array size of 100, the execution times for one hardware thread and one software thread are 1.43 msec and 5.02 msec respectively, yielding a speedup factor of 3.5. For an array size of 25,000 the speedup increases to 9.4.

The difference between two threads being time sliced and suffering context-switching time overhead on a CPU, and the time for two parallel hardware threads running in the hardware can also be seen by comparing the execution times and speedups for two software threads and two hardware threads with the previous cases of one hardware and one software thread. For an array size of 100, execution times of 1.5 msec and 8.43 msec for two hardware and two software threads respectively, yield a speedup of 5.6. For an array size of 25,000 this speedup increases to 19. Since hardware threads do not share CPU resources and are not time-sliced, hthread_yield() simply is ignored during the VHDL generation process and thus does not invoke context switching overhead typically incurred on the CPU. Thus these speedups increase when compared to the speedups obtained using only a single thread.

Finally, an interesting comparison can be seen from the execution times for runs with mixed numbers of hardware and software threads. For all array sizes the execution times using two hardware threads and two software threads actually increase when compared to using just two hardware threads. This results from allowing a slower software thread to lock a mutex thereby blocking one or both hardware threads from continuing processing at faster rates. These simple results indicate that Moore's law applied to FPGAs produces larger and faster devices, performance increases can scale with the number of threads, This is in contrast to software based threaded systems, where the creation of additional threads can result in degraded performance due to additional time slicing and context switching overhead. This example, when implemented with two hardware mutexAdd threads, for the Xilinx Virtex II Pro (XC2VP30), utilizes 11,658 slices (85% of available slices) and 34 block RAMs (25% of available block RAMs). This includes all supporting system IP components, such as the IPIF bus

```
void * matrixAdd( void * inputData ) {
  struct matrix * inputMatrix;
  Huint size, index, x, y;

  inputMatrix = (struct matrix *) inputData;
  size = inputMatrix->size;
  index = 0;

  while ( index < size ) {
    //Increment matrix.index, to tell the next thread what element to add
    hthread_mutex_lock( &matrixMutex );
    index = inputMatrix->index;
    inputMatrix->index = index+1;
    hthread_mutex_unlock( &matrixMutex );

    if ( index < size ) {
      x = inputMatrix->X[index];
      y = inputMatrix->Y[index];
      inputMatrix->Z[index] = x + y;
      hthread_yield();
    }
  }
}
```

Figure 8. Software Thread source code for matrixAdd

interface logic. The ELF file, compiled with the hthread kernel, main function and mutexAdd software thread is 233KB.

6 Status and Future work

Within our hthreads system, the first version of the HWTI included support for the most common system calls. However the HWTI does not yet support the creation of new threads from within a hardware thread. Thus, all threads, whether resident in either hardware or software, must be created by a software thread. When the HWTI is fully completed, we envision the hardware compilation process determining which system calls a thread makes, and only include the VHDL functionality in the HWTI for those calls. Currently all system calls are included within the HWTI when the user thread is instantiated.

We are currently working on a second version of a C to VHDL compiler, to enable a full end-to-end synthesis capability from C and pthreads. In our initial prototype, we compiled the C application code to C– [14] as an intermediate form. Our objective was to then translate C– to PowerPC code and, for threads selected to run in the FPGA, to translate C– to VHDL. Although we were successful in developing a C– to VHDL translation capability [12], we felt that C– did not possess all the attributes we desired in a true architecture independent form. We began investigating other available intermediate forms such as GIMPLE [15] and SUIF [21]. We have since began work on a new translator from GIMPLE, gcc's internal intermediate form, to replace C–. However, GIMPLE like most intermediate forms [15, 21, 14] assumes a CPU compilation target, and is, by its self, not perfectly suited for VHDL translation. To overcome this problem we are creating a new intermediate form called HIF, or Hardware Intermediate Form generated directly from GIMPLE. HIF gives us the ability to optimize the data and control flow graphs specifically for customized circuits in VHDL [19]. Furthermore a HIF to VHDL translator will be HWTI-aware and can import any needed library calls, into the HWTI. A more detailed description of the HIF grammar and additional examples may be found in [13].

7 Conclusion

In this paper, we have 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 API's. 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.

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

[3] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbee, J. Ortiz, E. Komp, and P. Ashenden. Programming Models for Hybrid FPGA-CPU Computatoinal Components: A Missing Link. *IEEE Micro*, 24(4):42–53, 2004.

[4] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, September 19-22, 2005.

Array Size		1 HW 0 SW	2 HW 0 SW	0 HW 1 SW	1 HW 1 SW	2 HW 1 SW	0 HW 2 SW	1 HW 2 SW	2 HW 2 SW
100	Speed	1.43	1.51	5.02	2.28	2.88	8.43	3.11	2.97
	Percent	0	0	100	0	0	100	0	0
5,000	Speed	15.8	13.9	142	15.0	15.5	269	16.6	19.6
	Percent	0	0	100	13	7.8	100	4.2	4.8
10,000	Speed	30.2	27.9	285	295	37.9	536	323	30.8
	Percent	0	0	100	15	10	100	4.6	4.1
25,000	Speed	75.4	69.1	709	71.9	76.3	1330	78.1	74.5
	Percent	0	0	100	15	8.9	100	4.8	4.4

Speed, measured in milliseconds, is time from first hthread_create(), to last hthread_join()
Percent, is the percentage of array elements performed by a software thread.

Figure 9. Matrix Add Example Timings

[5] P. Athanas and H. Silverman. Processor Reconfiguration Through Instruction-set Metamorphosis. In *IEEE Computer*, volume 26, pages 11–18, 1993.

[6] T. Callahan. Automatic Compilation of C for Hybrid Reconfigurable Architectures. *Ph.D. Dissertation at the University of California Berkeley*.

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

[8] K. Compton and W. Fu. An Execution Environment for Reconfigurable Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) 2005.*, 2005.

[9] 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.

[10] 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.

[11] 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.

[12] http://wiki.ittc.ku.edu/hybridthread/HybridThreads_Compiler. C2VHDL Compiler Documentation.

[13] http://wiki.ittc.ku.edu/hybridthread/Main_Page. KU Hybrid Threads Project.

[14] S. P. Jones, N. Ramsey, and F. Reig. C–: A Portable Assembly Language that Supports Garbage Collection. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP)*, October 1999.

[15] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *GCC Developers Summit*, pages 171–180, 2003.

[16] 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.

[17] 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.

[18] 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.

[19] R. Sharp and A. Mycroft. The FLaSH Compiler: Efficient Circuits from Functional Specifications. June 2003.

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

[21] suif.stanford.edu/suif/NCI/suif.html. Stanford University Intermediate Form.

[22] M. Vuletic, L. Possi, and P. Ienne. Virtual Memory Window for Application-Specifc Reconfigurable Coprocessors. In *Proceeding of the 41st Annual Conference on Design Automation, ACM Press*, pages 948–953, 2004.

[23] 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.

[24] 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.

[25] M. J. Wirthlin and B. L. Hutchings. DISC: The Dynamic Instruction Set Computer. In *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, volume SPIE 2607, pages 92–103, 1995.

[26] www.celoxica.com. Celoxica.

[27] www.impulsec.com. ImpulseC.

[28] www.intel.com/technology/hyperthread. Intel Hyper-Threading Technology.

[29] www.systemc.org. SystemC.