

Evaluation of the Hybrid Multithreading Programming Model using Image Processing Transforms

Razali Jidin, David Andrews, Wesley Peck, Dan Chirpich, Kevin Stout, John Gauch
Information Technology and Telecommunications Center
Department of Electrical Engineering and Computer Science
University of Kansas
{dandrews,rjidin,peckw}@ittc.ku.edu}

Abstract

Hybrid chips containing both CPU's and FPGA components promise the potential of providing a unified platform for seamless implementation of hardware and software co-designed components. Realizing the potential of these new hybrid chips requires new high level programming model capabilities that support a far more integrated view of the CPU and FPGA components than is achievable with current methods. The KU Hybrid Threads project has been investigating extending the familiar multithreaded programming model across this CPU/FPGA boundary to support both FPGA based hardware and CPU based software threads. Adopting this generalized multithreaded model can lead to programming productivity improvement, while at the same time providing the benefit of customized hardware from within a familiar software programming model. In this paper we present an application study of our hybrid multithreaded model. We have implemented several image-processing functions in both hardware and software, but from within the common multithreaded programming model on a XILINX V2P7 FPGA. This example demonstrates hardware and software threads executing concurrently using standard multithreaded synchronization primitives transforming real-time images captured by a camera and display it on a workstation.

1. Introduction

Recently emerging hybrid chips containing both CPU and FPGA components are an exciting new development that promise COTS economies of scale, while also supporting significant hardware customization. Researchers are investigating new design languages, hardware/software specification environments, and development tools to tap full potential of these hybrid chips. Projects such as

Ptolemy [2], Rosetta [6], and System-C [3] are investigating system level specification capabilities that can drive software compilation and hardware synthesis. Other projects such as Spark [12], Streams-C and Handel C are focused on raising the level of abstraction at which FPGAs are programmed by augmenting the C language. System Verilog and a newly evolving VHDL standard are also now being designed to abstract away the distinction between the two sides of the traditional low level hardware/software interface into a system level perspective. Although these approaches differ in the scope of their objectives, they all share the common goal of raising the level of abstraction required to integrate hardware and software components.

The KU hybrid threads project has been focused on extending the familiar multithreaded programming model to abstract the FPGA/CPU components, bus structure, memory, and low-level peripheral protocols into a transparent system platform [8]. Our hybrid multithreaded programming model allows the specification of applications as sets of threads distributed flexibly across the system CPU and FPGA assets.

In next section, we provide an overview of our hardware hybrid thread synchronization primitives – spin locks, blocking locks and blocking counting semaphores. We then describe our image processing application using our synchronization primitives and provide performance comparisons with similar software implementations.

2. Overview of Hybrid Thread

General programming models form the definition of software components, and governing interactions among the components [1]. Achieving abstract programming capabilities across the FPGA/CPU boundary requires adaptation of a high-level programming model that abstracts the FPGA and CPU components, bus structure, memory, and low-level peripheral protocol into a transparent computational platform [8]. The KU hybrid

threads project has chosen a multithreaded framework as our model, supporting concurrent hybrid threads distributed flexibly across the systems CPU and FPGA assets. Within our model, all threads adhere to the policies of accepted shared memory synchronization protocols for exchanging data and synchronizing control. To support this generalized model across the FPGA, we have developed a Hardware Thread Interface that encapsulates mechanisms to support synchronization for FPGA based threads. Under this unified model, application programmers perform procedure calls from within both VHDL and C to access high level synchronization primitives between threads running across both hardware and software computations. The set of Hardware Thread Interface components as well as a standard software interface component form our system-level hybrid thread abstraction layer as shown in Figure 1.

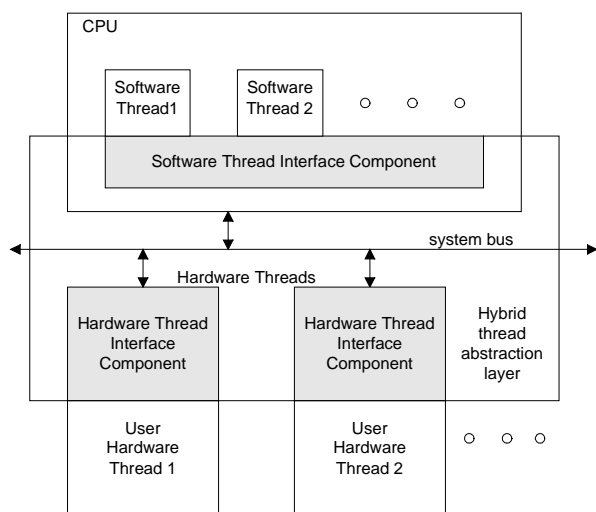


Figure 1. Hybrid thread abstraction layer

2.1. Hardware Thread Support

We provide our Hardware Thread Interface component as a library for inclusion within user-defined hardware threads. The hybrid thread abstraction layer implements all interactions between source hardware threads and other system components through the command and status registers. This capability is particularly useful for debugging and is used during runtime to interact with all other system components, such as our semaphore IP for thread blocking and wake-up. During debug, the status register is accessible from user programs allowing developers to monitor and control the execution state of a hardware thread.

We implement the Hardware Thread Interface component library as three subcomponents: a CPU

interface, a hardware-thread state controller, and a user hardware-computation interface. The user interface binds our device independent API's accessible from within the application program to platform-specific implementation methods. Thus the HTI component provides a general-purpose register set that supports platform independent user level semantics to promote thread migration across the system.

The controller state machine controls the execution of a user hardware thread, which will be in one of three states: idle, running, or waiting. Threads that have not yet started or have terminated are in the idle state. Threads that are currently in the run state transition into the wait state when a thread requests a semaphore, or continues to be blocked on a semaphore.

Each hardware thread's state is maintained in the status register. Dedicated hardware threads require no context switching when transitioning the thread into the wait state. Instead, the hardware thread simply idles. This allows the adoption of the same approach for both spinning and blocking semaphores. The hardware thread interface component, however, does perform different processing for spinning and blocking semaphores. For the spinning semaphore, the hardware thread interface transitions the thread state into the wait state, issues a single request for the semaphore, and return the thread to the running state when the request's status is returned in the status register. The thread then checks to see if it owns the semaphore or not. In contrast, for the blocking semaphore, the hardware thread interface transitions the thread to the wait state, issues the request for the blocking semaphore, and leave the thread in the wait state while semaphore is in use. Upon grant or release, the state machine will then transition the thread back into the run state.

User threads use standard API's to request system services through the user hardware-computation interfaces. The API writes an `op_code` into the operation register and waits for the controller to respond. The controller services requests initiated from user threads through the API's and updates the status register appropriately. The user thread request `op_code`, API and status register return codes are given in Table 1.

2.2. Hybrid Thread Synchronization

We implement efficient synchronization mechanisms that are CPU family independent, and require no additional control logic to interface into the system memory coherence protocol. As such, our new mechanisms are easily portable across shared and distributed memory multiprocessor configurations and are also available on CPU/FPGA systems with only software threads.

A single read operation is used to request a semaphore. We use the address lines to encode both the semaphore ID and thread ID during a normal bus read operation. The control structure within the semaphore IP then conditionally accepts or denies the request during a single read bus operation. We implement multiple semaphores within on chip BRAM, thus minimizing the number of CLB's that are traditionally used to implement the individual registers. The semaphore ID that is encoded within the address is directly decoded to select the semaphore in the BRAM.

Table 1. Hardware Thread API

Pseudo API	Opcode	Return Code
Read_data()	READ	READ_OK
Write_data()	WRITE	WRITE_OK
Sem_post()	WRITE	SEM_POST_OK
Sem_wait()	SEMWAIT	SEM_WAIT_OK
Mutex_lock	MTXLOCK	MTX_LOCK_OK
Mutex_Unlock	WRITE	MTX_UNLK_OK
Spin_lock()	SPINLOCK	SP_LOCK_OK
Spin_unlock()	WRITE	SP_UNLK_OK

2.2.1. Binary Spin Lock Semaphores

The block diagram for a (multiple) spin lock IP is shown in Figure 2. This single entity provides control for sixty-four spin locks. To request a semaphore, the *spin_lock()* API issues a single atomic read to an address formed by encoding the semaphore ID and thread ID as the least significant bytes of the base address. In response to this read operation, the spin lock controller decodes the address line and extracts both the semaphore and thread ID's. The extracted thread ID is then compared with the thread ID stored in the owner register to determine if the semaphore is empty or currently in used. If the owner register is free, it will be updated with the requested thread ID. If the lock is currently locked, then the control logic performs no update. After the check is performed, the controller places the appropriate thread ID from the current owner thread id onto the data bus and terminates the bus cycle. The controller takes eight cycles to complete the request. To release the semaphore, the API writes its thread ID to an appropriate address. The controller state machine then decodes the address lines and updates the selected owner register to non-owner status.

2.2.2. Blocking Synchronizations

Blocking synchronization allow threads that cannot gain access to the semaphore to be queued and

suspended thus enabling more efficient usage of the computing resources and decreasing congestion on the system bus. In many operating systems, each blocking semaphore resource is associated with a sleep or wait queue. As the total number of synchronization variables in a system may be quite large, implementing separate queues for each semaphore would require significant FPGA resources. Our approach is to create a single queue for all blocking semaphores. The queue size is an initial design parameter that is set to the total number of threads that can run concurrently within the system. Even though there will be many sub-queues associated with different semaphores, the combined lengths of all semaphore queues should not be greater than the total number of threads in the system as sleeping threads cannot make additional requests for other semaphores. Releasing a blocking semaphore triggers de-queuing of the semaphore's next owner. To manage the global queue efficiently, we created a single waiting queue that is divided into four tables - Queue Length, Next Owner Pointer, Last Request Pointer and Next Next-Owner.

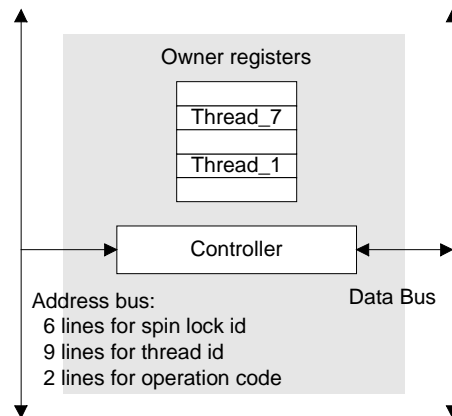


Figure 2. Multiple Spin Lock

The Queue Length Table maintains the length of each semaphores queue, and is accessed by indexing into the table with the semaphore ID. The Last Request Table contains a thread ID or pointer to the Next-next owner table. This table is also indexed by the semaphore ID. The table is used to point to the last semaphore request.

The Next Owner Table contains the next owner thread ID, which is also a pointer to Next-Next-Owner Table. When a semaphore is released, this pointer is used to provide next semaphore owner. Then it will be updated with new next owner by reading the next-next owner table. It is indexed semaphore ID. The Next-Next owner serves to provide a linked list between all the next owners of a given semaphore. It is indexed by a thread ID.

2.2.3. Blocking Binary Semaphores

The block diagram for a (multiple) block lock (MUTEX) IP is shown in Figure 3. This single entity carries sixty-four block locks (64 owner registers in BRAM). In addition to the owner registers, there is a wait queue to hold the thread ID's of blocked threads.

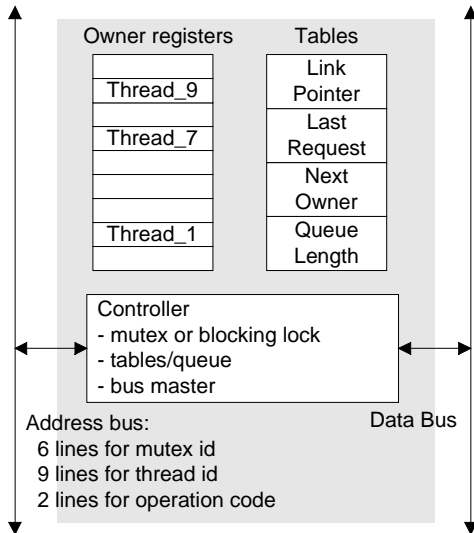


Figure 3. Blocking Binary Semaphore

This single queue holds all thread id's blocked on any of the sixty-four block locks. The controller responds similarly to the controller response for a spin lock. However, if the lock is not free, the appropriate queue length will be updated and the requested thread ID will be queued. The controller takes eight cycles to complete the request.

To release the semaphore, the API writes its thread ID to the release register. The controller will decode the address lines and update the selected owner register to non-owner status if the queue is empty. If the queue length of selected semaphore IDs is non zero, then the queue length is decremented and the next owner pointer will be read to de-queue a next owner thread ID, and the owner register will be updated with the next owner accordingly.

2.2.3. Blocking Counting Semaphores

The block diagram for a multiple blocking counting semaphore is shown in Figure 4. This entity carries sixty-four counting semaphores (64 counters in BRAM). In addition to the counters, there is a wait queue to hold the thread ID's of blocked threads. This single queue is designed to queue all threads blocked on all of the sixty-four semaphores.

To request for a semaphore, the *sem_wait()* API issues a read to an address formed by encoding the semaphore ID and thread ID as the least significant bytes of the base address. In response to this read operation, the controller decodes the address line and extracts both the semaphore and thread ID's. The controller reads the counter pointed by the extracted semaphore ID, places the read value on data bus, terminates the bus cycle, and perform additional operations depending on the value of the counters.

If the counter value is non-zero, the controller decrements the counter by one, otherwise extracted thread id is en-queued. If the returned value is zero, the API puts the thread to sleep.

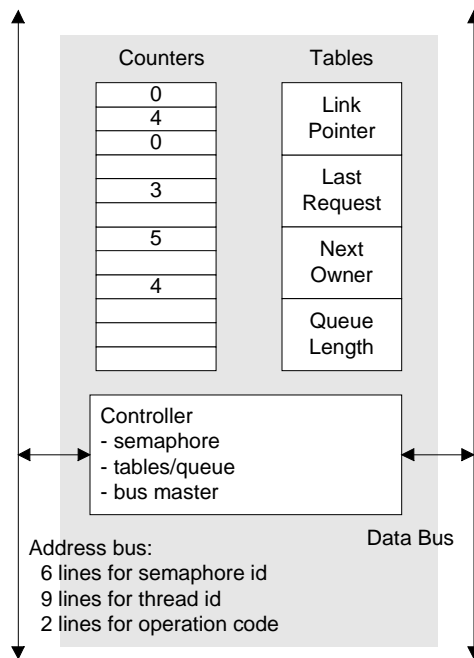


Figure 4. Blocking Counting Semaphore

The API pseudo code for blocking counting semaphore is shown in Figure 5. To release the semaphore, the *sem_post()* API write its thread ID to an appropriate address. The controller state machine then decodes the address lines and reads the selected counter.

If the selected counter is non-zero, it will incremented by one. Otherwise if the counter is zero, the controller proceeds checking the queue length of the selected semaphore. If the queue length is zero, the counter will be incremented. If the queue is not zero, the next semaphore owner will be de-queued and the counter will not be updated.

```

sem_wait(sema_id, thread_id) {
    address <= encode sema_id , thread_id;
    if location(address) == zero
        sleep( );
    else
        continue;
}

sem_post(&sema, thread_id) {
    address <= encode sema_id , thread_id;
    thread_id → location(address);
}

```

Figure 5. Blocking Counting Semaphore API

3. Image Processing

To verify our multithreaded model capability to support: concurrent execution of both hardware and software threads, communication and synchronization among these hybrid threads using the standard shared memory synchronization protocol, we have implemented several simple image-processing transforms using the experimental set-up as illustrated in Figure 6. A camera attached to a PC running LINUX is used to capture picture of moving objects. The image frames are then transferred from the PC to the V2P7 FPGA board via an Ethernet link. After the frame has been processed on the V2P7 boards, the modified image is then sent back to the PC. Both the original and modified images are then displayed on the PC real-time.

A software thread is created on the embedded PPC405 CPU to receive image frame data from the Ethernet and place it on the heap (in SDRAM). We use two counting semaphores to synchronize the CPU resident software and the FPGA resident hardware threads. Semaphore S1 synchronizes access to the shared image data not yet processed, while semaphore S2 serializes access to the processed images.

The software thread running on the PPC 405 starts with initialization routine. The C program listing is given in Figure 7. The initialization routine performs two memory allocations on the heap to get two pointers for storing image data, initializes the Ethernet link and calls a hardware thread create API. The hardware thread create API provides the two address pointers to the hardware thread through the hardware thread interface's argument registers, and also causes the transition of the hardware threads controlling state machine from the idle state to the run state. The software thread then waits for image to arrive from the Ethernet link. When a new image is available, the software thread transfers the received image into the

SDRAM location pointed by the first pointer. The software thread then performs a semaphore post on S1 to communicate to the hardware thread that the image to be processed is now available on the heap. The software thread then executes semaphore wait on semaphore S2.

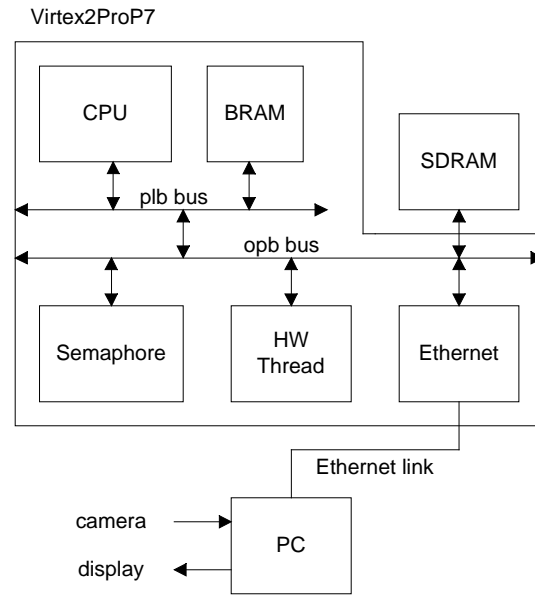


Figure 6. Hybrid Thread Image Processing

```

main( )
{
    // raw image data pointers
    addr1 = malloc(image_size)
    // processed image data pointer
    addr2 = malloc(image_size)

    img->in = addr1;
    img->out = addr2;

    //Hardware thread create API
    hw_thread_create(address1, address2, algorithm )

    while (1) {
        // Get image from ethernet
        receive(img->in, img_size)
        // Let hw thread know image data is available
        sem_post( &sema1 );
        //Wait for hw thread finish processing
        sem_wait( &sema2 );
        // Send processed image
        send(destination, img->out, img_size);
    }
}

```

Figure 7. CPU Software thread, part of C program

Upon receipt of the semaphore post (S1) operation from the software, the semaphore IP writes a wakes-up command to the blocking hardware thread (write it the hardware thread command register). The hardware thread awakes, reads the image from the heap, performs the image processing specified within the thread, and writes the processed image back into the heap at location provided by the second pointer variable. When the hardware thread finishes processing all the image pixels, it performs a post on semaphore S2 to signal the completion processing of the image frame to the software thread. The posting on semaphore S2 causes the sleeping software thread to wake and initiate transferring the processed image frame back to the workstation for display.

The hardware thread user component shown in Figure 6 is divided into two parts – a control unit and a data path. The control unit initiates requests to the hardware thread interface for accessing data and performing the synchronization operations. Addresses for accessing data from the SDRAM are generated by the control unit too. For several of our simple image processing functions such as gray scale inversion and threshold, the control unit is represented in the following five states: 1) semaphore wait, 2) read, 3) waiting for process to complete, 4) write and 5) semaphore post. The pseudo code for these five states is shown in Figure 8. The VHDL code for this hardware thread pseudo code that use API is given in Figure 9.

```

If command == run
  SW: sem_wait( &sema1 )
  RD: read data
      processing wait
      write data
      if count == image_size
        RD:
      else
        SP:
  SP: sem_post ( &sema2)
      branch SW

```

Figure 8: Hardware thread control code pseudo code

The data paths consist of logic for transforming data to the desired output. For the median filter algorithm, the data path is given in Figure 10, and consists of a frame buffer, a module to handle boundary conditions, and nine 8-bit comparators that produce medians of the nine pixels. The median filter operates at 100 MHz, and is capable of producing a new 8-bit pixel value every clock cycle.

We have implemented four separate image-processing algorithms in VHDL. As our FPGA hardware

resources are limited, we have implemented a minimal image- processing buffer size. For example for the median filter or the binomial filter, we have implemented a frame buffer that sizes up to about double of image width in byte $((\text{image width} * 2 + 3) * 8 \text{ bit})$.

```

when sem_wait =>
  semw1: sem_wait(base_addr, sema1, operation);
  if thread_status = SEM_WAIT_OK then
    reset_count <= '1';
    next state <= inv_read;
  end if;

when inv_read =>
  inv_rd: read(base_addr, addr1, operation);
  if thread_status = READ_OK then
    start_process <= '1';
    next state <= inv_filter;
  end if;

...

```

Figure 9. Hardware thread control (Part of VHDL code)

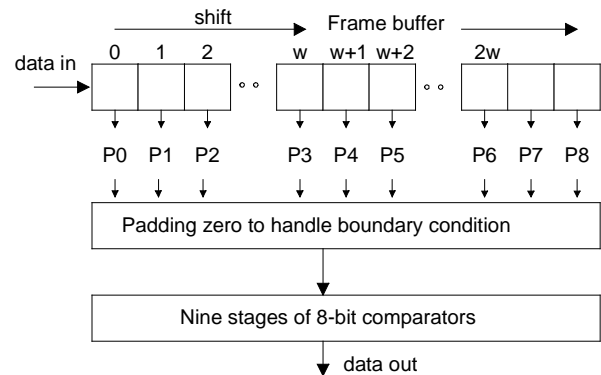


Figure 10. Data path - Median Filter in VHDL

```

img = (int*) malloc( width * height * sizeof(int) );
med = (int*) malloc( width * height * sizeof(int) );
k = (int*) malloc( 9 * sizeof(int) );

for (y = 0; y < height; y++ ) {
  for (x = 0; x < width; x++ ) {
    kc = 0;
    for(dx=max(0,x-1); dx<=min(width-1,x+1); dx++) {
      for(dy=max(0,y-1); dy<=min(height-1,y+1); dy++) {
        k[kc++] = img[dy*width + dx];
      }
    }
    sort(k, kc); // outer loop sort until (kc+1)/2
    med(y*width+x) = k[kc/2]
  }
}

```

Figure 11. Main part of Median Filter code in C

For performance comparisons, we have also implemented our targeted image algorithms in “C” language. Part of code for median filter running on software is given in Figure 11 above.

4. Results and Conclusion

Table 2 shows, for Virtex V2P7 FPGA, hardware resource required to implement hardware thread interface component that we have described in Section 2 earlier.

Table 2. Hardware Thread Interface Resources

Resource Type	# used	# total on chip	% used
Slices	366	4928	7.4
Flip-flop	332	9856	3.4
4 -input LUT	563	9856	5.7

The hardware cost to implement the four image transforms is given in Table 3. The cost includes the hardware thread interface component. The hardware thread interface contributes to about one fourth of total slices needed to implement the four image processing algorithms.

Table 3. Hardware Thread Resources

Resource Type	# used	# total on chip	% used
Slices	1429	4928	29
Flip-flop	1205	9856	12
4 -input LUT	2590	9856	26

The cost of FPGA hardware to implement sixty-four semaphores is given in Table 4. The semaphore entity supports *sem_wait*, *sem_trywait*, *sem_post*, and *sem_count_init* operations similar to POSIX API. The semaphore queue is sized to hold up to five hundreds and twelve sleeping threads (hardware or software threads).

Table 4. Hardware cost for 64 semaphores

Resource Type	# used	# total on chip	% used
Slices	229	4928	4.6
Flip-flop	186	9856	1.9
4 -input LUT	414	9856	4.2
BRAMs	2	44	4.5

The resource also includes the controller to dequeue and deliver the wake-up threads either to the scheduler queue or hardware threads. The result in Table 4 also indicated that the number of resource needed to implement each semaphore is about 3.6 slices only, and two BRAM for the sixty-four semaphores.

In this paper, we have presented an overview of our hybrid threads programming model. Both hardware and software threads synchronize their access to shared data through the standard the application program interface. Our hardware thread application interface enables application developers to write applications in VHDL without going into details of the system bus architecture. Therefore, our hybrid thread programming model can reduce development time, and opens the door for software engineers to access the reconfigurable logic through a familiar POSIX like generalized software multi threaded programming model.

To demonstrate the utility of our approach, we developed an image processing application that uses both software and hardware threads. The utility of providing programmers access to the potential of the FPGA through a familiar programming model can be seen by the performance comparisons given in Table 5. The results clearly show that the median filter implemented on hardware can handle about 90 frames per second based on an image size of 320 x 240 x 8 bits per frame. For comparison we operate the embedded CPU and reconfigurable logic at 100 MHz, the clock limit of current XILINX FPGA that we have. For the transforms that purely software based, we turn on and off the processor memory cache in order to study the effect of memory access on the CPU execution time. Further the four different transforms also indicated that CPU spends more time to perform image processing especially to calculate the median. Comparison on the execution times between the hardware based and software based transform implementations also reveal that the hardware thread enables us to exploit the nature of reconfigurable hardware architecture that it excels in doing repetitive tasks.

The four different hardware based transforms demonstrate that the executing times are dominated by the read and write of image from/to the system memory. This can be deduced from the same size of images (320 x 240 x 8 bits) processed by different transforms and their execution times vary slightly from 9.05ms to 11.2 ms. Even though latency of each hardware transform algorithm is small ranges from three to eleven clock cycles, and capable of producing average of nine pixel every clock cycle in the case of binomial and median transforms, the communication cost far overweigh image processing time as indicated by the results in the table.

Table 5. Image Transforms Execution Times

Image algorithms	HW image processing	SW image processing cache off	SW image processing cache on
Threshold	9.05 ms	140.7ms	19.7 ms
Negate	9 .05ms	133.9ms	17.5ms
Median	11.2 ms	2573ms	477ms
Binomial	10.6 ms	1282ms	320 ms

Our experiments also demonstrate that hybrid thread provides an ideal development environment for developing concurrent threads that execute within reconfigurable hardware. Our approach enables us to harness FPGA superiority in performing repetitive tasks and at the same time allows general-purpose processors to execute irregular or rarely repeated code within the familiar multithreading program model.

A more details description of KU hybrid thread project can be found at <http://www.ittc.ku.edu/hybridthreads/>.

5. Acknowledgement

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

6. Bibliography

[1] Lee, Edward, "Whats ahead for Embedded Software?", IEEE Computer, Sept 2000, pp.18-26

[2] Lee, Edward, Overview of the Ptolemy Project, Technical Memorandum, March 6, 2001, UCB/ERL M01/11 University of California

[3] Maya B. Gokhale and Janice M. Stone and Jeff Arnold and Mirek Kalinowski, "Stream-Oriented FPGA Computing in the Streams-C High Level Language", Proceedings of the Eight Annual IEEE Symposium on Filed-Programmable Custom Computing Machines (FCCM), April 2000

[4] P. Moisset, P. Diniz, J. Park, "Matching and Searching Analysis for Parallel Hardware Implementation on FPGAs", Int'l Symposium on Field Programmable Gate Arrays, FPGA'01, Monterey, California, USA, Feb 2001, pp.125-133

[5] G. Snider, B. Shackelford, R.J. Carter, "Attacking the Semantic Gap Between Application Programming Languages and Configuration Hardware", International Symposium on Field Programmable Gate Arrays, FPGA'01, Monterey, California, USA, Feb 2001, pp.115-124

[6] Perry Alexander and Cindy Kong, Rosetta: "Semantic Support for Model Centered Systems Level Design", IEEE Computer, November 2001

[7] F. Baloron, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, M. Chiodo, H. Hsieh, L. Lavagno "Hardware-Software co-design of embedded systems: the POLIS approach", Kluwer, 1997

[8] D.L. Andrews, D. Niehaus, P. Ashenden, " Programming Models for Hybrid FPGA/CPU Computational Components", IEEE Computer, January 2004

[9] D. L. Andrews, D. Niehaus, and R. Jidin, Implementing the Thread Programming Model on Hybrid FPGA/CPU Computational Components," Proc. 1st Workshop on Embedded Processor Arch, Proc. 10th Int'l Symp. High Performance Computer Architecture (HPCA 10), Feb 2004.

[10] R. Jidin, D. L. Andrews, and D. Niehaus, "Implementing Multithreaded system Support for Hybrid FPGA/CPU Computational Components," Proc. Int'l Conf. on Engineering of Reconfigurable System and Algorithms, CSREA Press, June 2004. pp. 116-122.

[11] D. L. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, P. Ashenden, "Programming Models for Hybrid FPGA-CPU Computation Components – A Missing Link", IEEE Micro, July/Aug 2004.

[12] S. Gupta, M.Luthra, N.D. Dutt, R. K. Gupta, A. Nicolau, " SPARK: A high -Level Synthesis Framework for Applying Parallelizing Compiler Transformations, Proceedings of the International Conference on VLSI Design, January, 2003

[13] M. Frisbee, D. Niehaus, V. Subramonian, C. Gill, "Group Scheduling in Systems Software Workshop on Parallel and Distributed Real-Time Systems", Proceedings of International Parallel and Distributed Processing Symposium, Santa Fe, NW, April 2004.

[14] J.Firgo, M.Gokhale, D. Lavenier, "Evaluation of the Stream-C C-to-FPGA Compiler: An Application Perspective, ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays, FPGA'01, Feb11-13, 2001, Monterey, California, USA.

[15] L. A. King, H. Quinn, M. Leeser, D. Galatopoulos, Manolakos, E., "Runtime Execution of Reconfigurable Hardware in a Java Environment" in the Proceedings of the IEEE International Conference on Computer Design (ICCD-01), 2001, pp. 380-385.

[16] K. A. Robbins, S. Robbins, Practical Unix Programming, A guide to Concurrency, Communication and Multithreading, Prentice Hall, 1996.