Extending the Thread Programming Model Across Hybrid FPGA/CPU Architectures

Dissertation Defense
by
Razali Jidin

Advisor: Dr. David Andrews

Information Technology and Telecommunications Center (ITTC)
University of Kansas

April 15, 2005
Thank you

• Committee members
  – Dr. David Andrews
  – Dr. Douglas Niehaus
  – Dr. Perry Alexander
  – Dr. Jerry James
  – Dr. Carl E. Locke Jr.

• Research members
  – Wesley Peck
  – Jason Agron, Ed Komp, Mitchel Trope, Mike Finley, Jorge Ortiz, Swetha Rao, …….
Presentation Outline

- Problem Statement & Motivation
- Background – Previous works
- Research Objectives
- Hybrid Synchronization Mechanisms
- Hardware Thread
- Performance Results
- Evaluation of Hybrid Thread - Image Processing
- Conclusion & Future Works
Contributions

• Status: Completed HW/SW co-design of multithreading programming model, stable and running.
• Publications:
• Further impacts:
  Inquiries received from universities world wide and Cray Research
Problem Statement

• FPGAs serve as computing platforms?
  – History: serve as prototyping and glue logics devices
  – Becoming more denser and complex devices
  – Hybrid devices: embedded CPUs + other resources
  – Require better tools to handle new complexities

• Current FPGA programming practices
  – Require hardware architecture knowledge – not familiar to the software engineers
  – Have to deal with timing issues, propagation delays, fan-out, etc.
  – Hardware and software components interaction using low level communication mechanisms
Motivation: Hybrid CPU/FPGA Architectures

- Embedded PPC 405 CPU + Sea of free FPGA Gates (CLB’s) + …
- BRAM provides efficient storage to save system “states”.
- System components provided as libraries or soft IPs:
  - System buses (PLB, OPB)
  - Interrupt controllers, UARTs
- Migration of system services from CPU into FPGA can provide new capabilities to meet system timing performance. New services are provided in the form of soft IPs

Source: IBM T.J Watson Research Center
Motivation: Higher Abstraction Level

- Need to use high level of abstraction to increase productivity
  - Focus on applications not on hardware details.
  - Reduce the gap between HW and SW designs, to enable programmer access to hybrid devices.
- Hybrid Thread Abstraction Layer
  - abstract out hardware architectures such as buses structure, low level peripheral protocols, CPU/FPGA components, etc.
Previous Works

• Research efforts to bring High Level Languages (HLL) into hardware domain

• Streams-C [Los Alamos]
  – Supplements C with annotations for assigning resources on FPGA
  – Suitable for systolic based computations, compiler based on SUIF
  – Hardware/software communication using FIFO based streams
  – Programming productivity versus device area size

• Handel C [Oxford]
  – Subset of C + additional constructs for specifying FPGA circuits
  – Compile Handel C programs into synchronous machines
  – Hardware/software interactions using low level communication mechanisms

• System level [System C, Rosetta]
  – Attempt to remove hardware/software boundaries
  – High level integration between hardware & software components?
Research Objectives

• Goal: Create an environment where programmers can express system computations using familiar parallel thread model
  – standard thread semantics across CPU/FPGA boundaries
  – threads represented such that they can exist on both components
  – enable threads to share data with synchronization mechanisms

• Issues of interest:
  – FPGA based Thread Control and Context:
    • initiating, terminating, synchronizing threads
    • computational models (threads over FSM’s)
    • new definition of thread context
  – Synchronization Mechanisms for CPU/FPGA based Threads
    • Semaphore, lock (mutex) or condition variables
  – API and Operating System (OS) Support
    • User Application Program Application (API) Library Functions
    • System services adaptation and migration
      – Ex. thread scheduling
Current Thread Programming Model (TPM)

- An application can be broken into executable units called threads (a sequence of instruction).
- Threads execute concurrently on CPUs (M threads map to N CPUs)
- Threads interleave on a single CPU to create an illusion of concurrency
- Accesses to shared data are serialized with the aid of synchronization mechanisms.
Current Synchronization Mechanisms

• Current synchronization mechanisms
  – Depend on atomic operations provided by HW or CPU
  – SW: CPU instructions Test and Set
    • Set variable to one, return old value to indicate prior set
  – HW: Snoopy cache on multiprocessors

• Challenges
  – Current methods do not extend well to the HW based Thread
  – Do not want to increase overhead on CPU

• New methods
  – FPGAs provide new capabilities to create more efficient mechanisms to support semaphores
  – No special instruction, no modification to processor core
  – New FPGA based synchronization mechanism provided as IP cores
Achieving Atomic Operations with FPGA

• Atomic transaction controller on FPGA
  – Read acknowledgement is delayed
  – Hardware operation completes within this delay
  – Use lower order address lines to encode necessary information such thread ID and lock ID
  – Controller returns status & grant to the application program interface (API) request on data bus

• Issues on cost of FPGA resources when the number of synchronization variables in a system is large
  – Implement all the synchronization variables within a single entity.
  – Use a single controller to manage multiple synchronization variables.
  – Use on chip block memory (BRAM) instead of LUT to save the state of each individual variables
  – Example our multiple (64) spin locks core
Multiple Spin Locks Core

- **APIs**
  - Spin_lock
  - Spin_unlock

- **Lock BRAM**
  - 64 Recursive counters
  - 64 Lock Owner register

- **Controllers**
  - Common controllers for multiple locks
  - Access to Lock BRAM
  - Atomic read transaction
  - Recursive error
  - Reset all locks
Blocking Type Synchronization

• Spin vs. blocking type synchronization
  – Blocking reduces bus activities and does not tie CPU
  – Blocking requires queues to hold the sleeping threads

• Mapping of synchronization variables to sleep queues
  – Provides a separate queue for each blocking semaphore is costly when many semaphore variables are needed on a system

• Global Queue
  – Creates multiple semaphores with a single global queue
  – Efficient queuing operation but not at the expense of hardware resources

• Wakeup mechanism & delivery of unblocked threads
  – De-queue operation of unblocked threads
  – Delivery of unblocked threads either to the scheduler queue or individual hardware threads (bus master capability)
Hybrid Thread System

- Moves Mutexes + queues + wake-up into FPGA from memory
- Provides synchronization services to FPGA & CPU threads
Blocking Synchronization Core Design

- **Global Queue**
  - Conceptually configured as multiple sub-queue associated with different semaphores
  - Combined lengths of all sub-queues will not be greater than the number of total threads in the system as a blocked thread cannot make another request
  - For efficient operation, the global queue is divided into four tables:
    - Queue Length Table contains an array of queue lengths
    - Next owner Pointer Table contains an array of lock next owners
    - Last Request Pointer Table contains an array of last requesters
    - Next Next Owner Table contains link pointers
Global Queue & Lock Owner Registers

Queue Length Table
- Queue length = 0
- Queue length = 3
- Queue length = 8
- ....

Address
- lock owner S0 = 00
- lock owner S1 = 00
- lock owner S2 = 99
- lock owner S3 = 00
- ....
- lock owner S26 = 00
- lock owner S27 = 00
- ....
- lock owner S40 = 00
- ....
- lock owner S63 = 01

Queue length = 0
Queue length = 3
Queue length = 8
....

Address + 64

Lock owner registers

Link Pointer Table

- Next next owner = 09
- Next next owner = 11

Address

Last Request Table
- Last Request = 04
- Last Request = 11
- ....
- Last Request = 05

Address

Next owner Pointer Table
- Next owner = 08
- Next owner = 07
- ....
- Next owner = 20

Address

Next Owner Pointer Table

Address

Indexed by lock id

Address

Indexed by lock id

Address

Indexed by thread id

Address
Multiple Recursive Mutexes Core

- Provide exclusive accesses to shared data & allow threads to block
- Operations: `mutex_lock` (recursive), `unlock` and `trylock`

**mutex_lock()**

if thread ID = OWNER
   lock selected mutex
   cnt = cnt + 1
else
   queue thread ID

**mutex_unlock()**

cnt = cnt – 1
releases the mutex
when its cnt reaches 0
Multiple Semaphores Core

- **sem_wait(sm)**
  - if $C \geq 1$ then $C = C - 1$
  - else queues thread ID
- **sem_post(sm)**
  - if blocked thread, dequeues
  - else $C = C + 1$
- **sem_trywait(sm)**
  - non blocking
A Condition Variable

- Implements sleep/wakeup semantics using condition variables
- Useful for event notification
- Associated with a predicate which is protected by a mutex or spin lock
- Wakeup one or all sleeping threads
- Up to 3 or more mutexes are typically required:
  - one for the predicate
  - one for the sleep queue (or CV list)
  - one or more for the scheduler queue (context_switch)
- New approach requires one mutex (predicate)
Condition Variable APIs

```c
void wait (cv *c, mutex *m)
{
    lock (&c->qlistlock);
    add thread to queue
    unlock (&c->qlistlock);
    unlock (m); //release mutex
    context_switch ( );
    /* when wakes-up */
    lock (m); //acquire mutex
    return;
}

void signal (cv *c)
{
    lock (&c->qlistlock);
    remove a thread from list
    unlock (&c->qlistlock);
    if thread, make runnable;
    return;
}

void broadcast (cv *c)
{
    lock (&c->qlistlock);
    while (qlist is nonempty) {
        remove a thread
        make it runnable
    }
    unlock (&c->qlistlock);
    return;
}
```

Source: VAHALIA, UNIX Internals
Multiple Condition Variables Core

- **cond_wait(cv, mutex)**
  - Queuing of thread IDs
- **cond_signal(cv)**
  - De-queuing of a thread ID
- **cond_broadcast(cv)**
  - De-queuing & delivery of all blocked threads
  - Return busy status to new requests if delivery is not complete yet.
**Bus Master Interface** (IPIF MASTER)

1. Request Handlers
2. Bus Mastering
   - reader
   - writer

**Bus Slave Interface** (IPIF SLAVE)

1. Determine next owner:
   - HW or SW thread
2. Generate read or write to Bus Master
3. Calculate next owner address

**Comparator**

1. Determine next owner:
   - HW or SW thread
2. Generate read or write to Bus Master
3. Calculate next owner address

**Controller for multiple mutexes**

1. Manage recursive mutexes
2. Update owner register
   - with new owner if free
   - with next owner (deque)
3. Gen enque if lock not free
4. Gen deque if lock release
5. Soft Reset all own registers

**Shader Controller**

1. Request Handlers
2. Bus Mastering
   - reader
   - writer

**Atomic transaction**

- control owner register
- read req ack delay

**API return status**
- Status busy/OK
- Xfer status betw regs

**Operation mode**

- Decode address & read
- Determine lock/unlock

**Mutex ID register**

**Thread ID register**

**Queue with 4 tables**

- Link Pointers
- Last Request
- Next Owners
- Queue Lengths

**Queue Controller**

1. Enqueue blocking thread
2. Dequeue next lock owner
   - signals E to update owner register
   - signals D to via F to deliver next owner
3. Manage queue/4 tables
4. Soft Reset, clear all the table

**Next Owner Address Generator**

Parameters:
- HW thread base address
- HW Thread size
- SW thread Manager address
Hardware Thread Architecture

Bus Interface (Architectural dependent + independent components)

Hardware Thread Interface Component

Command

State Machines:
- Thread state scheduler
- Status process
- Command process
- Bus Slave Handshake

State Machines:
- Bus Master Handshake
- Address Generator
- Bus Writer/Reader
- Data in/out
- Synchronization tests, Busy wait

Address

Parameter1

Parameter2

Read data

User Hardware Thread

Control Unit uses API (operation=mutex, mutex_id=xx, parameter=thread_id)

Data and data processing such as image processing algorithm like median filter
Hardware Thread Interface Core (HWTI) RTL level description.
Hardware Thread States (Contexts)

1. Moves to RUN if receives cmd_run
2. Moves to WAIT while in the process of obtaining mutex or semaphore
3. Moves to RUN state if mutex is obtained.
4. If mutex is not available, block waits in WAIT state until wake-up command is received from the mutex core
5. Thread state visible via status register
6. User computation decides when it is appropriate to check status register and control its own operation
Hardware Thread APIs

• **HW_Thread_Create API on CPU**
  – CPU loads arguments to registers
  – CPU writes “code” into command register to start/stop

• **HW APIs on HW Thread**
  – Synchronization APIs
    • Mutex: blocking lock, unlock
    • Semaphore: wait, post
    • Spin lock: lock, unlock
  – Memory read/write accesses APIs
  – APIs write operation codes into the operation register, and status register provides feedback to the user
HW/SW Threads Spin Lock Access Ratios

- Baseline performance HW and SW thread run individually to own and release a spin lock, hw faster by a 6:1 ratio.
- Allow both Hardware/Software Hybrid Threads to compete:
Timing Performance

Blocking Mutex (Data Cache On)

- Mean: 597 ns
- Std Dev: 30
- Minimum: 520 ns
- Maximum: 790 ns

Time to Lock Mutex in Nanoseconds

Number of Events

250 Threads | 128 Threads | 64 Threads | 32 Threads | 16 Threads | 2 Threads
Timing Performance

Blocking Mutex (Data Cache Off)

- Mean: 750 ns
- Std Dev: 0
- Minimum: 750 ns
- Maximum: 750 ns

Time to Lock Mutex in Nanoseconds

- 250 Threads
- 128 Threads
- 64 Threads
- 32 Threads
- 16 Threads
- 2 Threads
# Synchronization Hardware Cost

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Total slices for 64 synchronization variable</th>
<th>Number of slices per synchronization variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spin Lock</td>
<td>123</td>
<td>1.9</td>
</tr>
<tr>
<td>Mutex</td>
<td>189</td>
<td>3</td>
</tr>
<tr>
<td>Semaphore</td>
<td>229</td>
<td>3.6</td>
</tr>
<tr>
<td>Condition Variable</td>
<td>137</td>
<td>2.1</td>
</tr>
</tbody>
</table>

## Hardware Resources for 64 MUTEXES (excluding bus interface)

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Resources Used</th>
<th>Total Resources On-chip</th>
<th>% Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-input LUT</td>
<td>328</td>
<td>9856</td>
<td>3.3%</td>
</tr>
<tr>
<td>Flip-flop</td>
<td>134</td>
<td>9856</td>
<td>1.4%</td>
</tr>
<tr>
<td>Slices</td>
<td>189</td>
<td>4928</td>
<td>3.8%</td>
</tr>
<tr>
<td>BRAMs</td>
<td>2</td>
<td>44</td>
<td>4.5%</td>
</tr>
</tbody>
</table>
## Synchronization Access Time

<table>
<thead>
<tr>
<th>Synchronization APIs</th>
<th>internal operation (clk cycles)</th>
<th>bus transaction after internal operation start (clk cycles)*</th>
<th>Total clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>spin_lock</td>
<td>8</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>spin_unlock</td>
<td>8</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>mutex_lock</td>
<td>8</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>mutex_trylock</td>
<td>8</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>mutex_unlock</td>
<td>13</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>sem_post</td>
<td>9</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>sem_wait</td>
<td>6</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>sem_trywait</td>
<td>6</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>sem_init</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>sem_read</td>
<td>6</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>cond_signal</td>
<td>11</td>
<td>10</td>
<td>21</td>
</tr>
<tr>
<td>cond_wait</td>
<td>10</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>cond_broadcast</td>
<td>6n</td>
<td>10n</td>
<td>16n</td>
</tr>
</tbody>
</table>
Hybrid Threads: Image Processing

Virtex2ProP7

CPU

BRAM

Controller

Semaphores

HW Thread

SDRAM

Controller

Ethernet

Camera: USBVISION
Image Display: SDL
O/S: Linux

IBM Compatible
Image Processing Flow Diagram

1. CPU init( )
   - ether_init()
   - a1 = malloc()
   - a2 = malloc()
   - hw_create( a1, a2)

2. CPU loads image into memory at address a1
   - receive image
   - recv(a1, img_size)

3. sema s1
   - sem_post(s1)
   - sem_wait(s1)

4. HW thread interface*
   - get image from memory
   - read ( a1 )

5. HW thread interface*
   - store processed image in memory
   - write ( a2 )

6. HW thread image processing* (Filter)
   - hw image process
     - 3x3 win median
     - invert
     - threshold
     - 3x3 win binomial

7. sema s2
   - sem_post(s2)
   - sem_wait(s2)

8. CPU reads memory a2 and send processed image
   - send image out
   - send(a2, img_size)

Note* VHDL
PART OF SOFTWARE (CPU):
addr1 = malloc(image_size) //raw image ptr
addr2 = malloc(image size) //proc image ptr
//Hardware thread create API
hw_thread_create(addr1, addr2, function )
while (1) {
    //Get image from Ethernet
    receive(src, addr1, 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(dest, addr2, img_size); }

PART HARDWARE (FPGA):
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:
Frame buffer & Parallel Median Filter

**Frame Buffer**
- Size: \((2W+3) \times 8\) bits
- Output: 3x3 window or 9 pixels
- Image size: \(W \times H \times N\)

**Boundary condition:**
- top left, top side, top right, right side, etc

**Pipelined Median filter**
- 9 stages, 8 bit comparators
- Calculate median of 9 pixels

**Image**
- 4 byte / pixels
- Padding zeroes to handle boundary conditions

**8 x 8-bit shift register**
- 4 byte outputs / 4 medians
HW vs. SW Image Processing

- Image frame size 240 x 320 x 8 bits
- FPGA & CPU clocked at 100 MHz
- For median transform, FPGA can process 100 frames/sec, speed-up about 40x, consistence with [12, 27]
- Execution time dominated by communication

<table>
<thead>
<tr>
<th>Image Algorithms</th>
<th>HW Image Processing</th>
<th>SW Image Processing Cache OFF</th>
<th>SW Image Processing Cache ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold</td>
<td>9.05 ms</td>
<td>140.7 ms</td>
<td>19.7 ms</td>
</tr>
<tr>
<td>Negate</td>
<td>9.05 ms</td>
<td>133.9 ms</td>
<td>17.5 ms</td>
</tr>
<tr>
<td>Median</td>
<td>11.2 ms</td>
<td>2573 ms</td>
<td>477 ms</td>
</tr>
<tr>
<td>Binomial</td>
<td>10.6 ms</td>
<td>1084 ms</td>
<td>320 ms</td>
</tr>
</tbody>
</table>
Conclusion & Future Works

- Extend thread programming model across CPU/FPGA
- Our synchronizations cores provides services similar to POSIX thread.
  - Test program uses our CVs and mutex produced similar result when port it to desktop running with Pthread.
  - Semaphores used in the image transform evaluations.
- Effective synchronization mechanism, improve system performance & reduce memory requirements.
- Improve programming productivity, while at the same time providing the benefit of customized hardware from within a familiar software programming model
- Hardware thread can be used as a base to implement other computations into hardware.
- High level language compiler that can translate applications into hybrid hardware and software components
Thank You!