FARM: A Prototyping Environment for Tightly-Coupled, Heterogeneous Architectures

Tayo Oguntebi, Sungpack Hong, Jared Casper, Nathan Bronson

Christos Kozyrakis, Kunle Olukotun
Outline

- Motivation
- The Stanford FARM
- Using FARM
Motivation

- **FARM**: Flexible Architecture Research Machine
- A high-performance flexible vehicle for exploring new tightly-coupled computer architectures

- New heterogeneous architectures have unique requirements for prototyping
  - Mimic heterogeneous structures and communication patterns
- Communication among prototype components must be efficient...
Motivational Examples

- Prototype a hardware memory watchdog using an FPGA
  - FPGA should know about system-level memory requests
  - FPGA must be placed closely enough to CPUs to monitor memory accesses
- An intelligent memory profiler
- Hardware race detection
- Transactional memory accelerator
- Other fine-grained, tightly-coupled coupled coprocessors...
Motivation

- CPUs + FPGAs: Sweet spot for prototypes
  - Speed + Flexibility
  - New, exotic computer architectures are being introduced: need high performing prototypes

- Natural fit for hardware acceleration
  - Explore new functionalities
  - Low-volume production

- “Coherent” FPGAs
  - Prototype architectures featuring rapid, fine-grained communication between elements
Motivation: The Coherent FPGA

**Why coherence?**
- Low latency coherent polling
- FPGA knows about system off-chip accesses
  - Intelligent memory configurations, memory profiling
- FPGA can “own” memory
  - Memory access indirection: security, encryption, etc.

**What’s required for coherence?**
- Logic for coherent actions: snoop handler, etc.
- Properly configure system registers
- Coherent interconnect protocol (proprietary)
- Perhaps a cache
Outline

- Motivation
- The Stanford FARM
- Using FARM
The Stanford FARM

- FARM (Flexible Architecture Research Machine)
- A scalable fast-prototyping environment
  - “Explore your HW idea with a real system.”
  - Commodity full-speed CPUs, memory, I/O
  - Rich SW support (OS, compiler, debugger ... )
  - Real applications and realistic input data sets
  - Scalable
  - Minimal design effort
The Stanford FARM: Single Node

- Multiple *units* connected by high-speed memory fabric
- CPU (or GPU) units give state-of-the-art computing power
  - OS and other SW support
- FPGA units provide flexibility
- Communication is done by the (coherent) memory protocol
  - Single node scalability is limited by the memory protocol

An example of a single FARM node
The Stanford FARM: Multi-Node

- Multiple FARM nodes connected by a scalable interconnect
  - Infiniband, ethernet, PCIe...
- A small cluster of your own

An example of a multi-node FARM configuration
The Stanford FARM: Procyon System

- Initial platform for single FARM node
- Built by A&D Technology, Inc.
- CPU Unit (x2)
  - AMD Opteron Socket
  - DDR2 DIMMs x 2
- FPGA Unit (x1)
  - Altera Stratix II, SRA
  - Debug ports, LEDs, etc.
- Each unit is a board
- All units connected via cHT
  - Coherent HyperTransport
  - We implemented cHT in FPGA unit (next slide)
The Stanford FARM: Procyon System

- Initial platform for single FARM node
- Built by A&D Technology, Inc.
- CPU Unit (x2)
  - AMD Opteron Socket F (Barcelona)
  - DDR2 DIMMs x 2
- FPGA Unit (x1)
  - Altera Stratix II, SRAM, DDR
  - Debug ports, LEDs, etc.
- Each unit is a board
- All units connected via HT
  - Coherent HyperTransport (version
  - We implemented cHT compatibility for FPGA unit (next slide)
The Stanford FARM: Procyon System

- Initial platform for sparsely connected networks
- Built by A&D Technology
- CPU Unit (x2)
  - AMD Opteron Server grade CPUs
  - DDR2 DIMMs x 2
- FPGA Unit (x1)
  - Altera Stratix II, SRAM, DDR
  - Debug ports, LEDs, etc.
- Each unit is a board
- All units connected via cHT backplane
  - Coherent HyperTransport (version 2)
  - We implemented cHT compatibility for FPGA unit (next slide)
The Stanford FARM: Procyon System

- Initial platform for single FARM
- Built by A&D Technology, Inc.
- CPU Unit (x2)
  - AMD Opteron Socket F (Base)
  - DDR2 DIMMs x 2
- FPGA Unit (x1)
  - Altera Stratix II, SRAM, DRAM
  - Debug ports, LEDs, etc.
- Each unit is a board
- All units connected via cHT backplane
  - Coherent HyperTransport (version 2)
  - We implemented cHT compatibility for FPGA unit (next slide)
The Stanford FARM: Base FARM Components

- Block diagram of FARM on Procyon system
- Three interfaces for user application
  - Coherent cache interface
  - Data stream interface
  - Memory mapped register interface

*chHTCore was created by the University of Manheim
The Stanford FARM: Base FARM Components

- Block diagram of FARM on Procyon system
- Three interfaces for user application
  - Coherent cache interface
  - Data stream interface
  - Memory mapped register (MMR) interface
- FPGA Unit: communication logic + user application
The Stanford FARM: Data Transfer Engine

- Ensures protocol-level correctness of cHT transactions
  - e.g. Drop stale data packets when multiple response packets arrive
- Handles snoop requests (pull data from the cache or respond negative)
- Traffic handler: memory controller for reads/writes to FARM memory
  - MMR loads/stores also handled here
The Stanford FARM: Coherent Cache

- Coherently stores system memory for use by application
- Write buffer: stores evicted cache lines until write back
- Prefetch buffer: extended fill buffer to increase data fetch bandwidth
- Cache lines either modified or invalid
Resource Usage

<table>
<thead>
<tr>
<th>Resource</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Kbit Block RAMs</td>
<td>144 (24%)</td>
</tr>
<tr>
<td>Logic Registers</td>
<td>16K (15%)</td>
</tr>
<tr>
<td>LUTs</td>
<td>20K</td>
</tr>
</tbody>
</table>

- Cache module is heavily parameterized
  - Numbers reflect 4KB, 2-way set associative cache
- And our FPGA is a Stratix II...
Outline

- Motivation
- The Stanford FARM
- Using FARM
Communication Mechanisms

- CPU $\rightarrow$ FPGA
  - Write to Memory Mapped Register (MMR)

<table>
<thead>
<tr>
<th>Number of Register Reads</th>
<th>Registers on FARM FPGA</th>
<th>Registers on a PCIe Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>672 ns</td>
<td>1240 ns</td>
</tr>
<tr>
<td>2</td>
<td>780 ns</td>
<td>2417 ns</td>
</tr>
<tr>
<td>4</td>
<td>1443 ns</td>
<td>4710 ns</td>
</tr>
</tbody>
</table>
Communication Mechanisms

- CPU ➔ FPGA
  - Write to Memory Mapped Register (MMR)
  - Asynchronous write to FPGA (streaming interface)
    - FPGA owns special address ranges which causes non-temporal store.
    - Page table attribute: Write-Combining.
      (Weaker consistency than non-cacheable)
  - Write to cacheable address; FPGA reads it out later (coherent polling)
Communication Mechanisms

- **FPGA ➔ CPU**
  - CPU read from MMR (non-coherent polling)
  - FPGA writes to cacheable address; CPU reads it out later (coherent polling)

Non-coherent polling

Coherent polling
Communication Mechanisms

- FPGA → CPU
  - CPU read from MMR (non-coherent polling)
  - FPGA writes to cacheable address; CPU reads it out later (coherent polling)
  - FPGA throws interrupt
Proof of Concept: Transactional Memory

- Prototype hardware acceleration for TM
- Transactional Memory
  - Optimistic concurrency control (programming model)
  - Promise: simplifying parallel programming
  - Problem: Implementation overhead
    - Hardware TM: expensive, risky
    - Software TM: too slow
    - Hybrid TM: FPGAs are ideal for prototyping...
Briefly...

- Hardware performs conflict detection and notification

Messages
- Address transmission (CPU→FPGA)
  - At every shared read
  - Fine-grained & asynchronous
  - Stream interface
- Ask for Commit (CPU→FPGA→CPU)
  - Once at the end of a transaction.
  - Synchronous; full round-trip latency
  - Non-coherent polling
- Violation notification (FPGA→CPU)
  - Asynchronous
  - Coherent polling
Performance Results

![Graph showing performance results]

- Software TM
- FPGA Scheme 1
- FPGA Scheme 2
- Upper Bound

The graph compares the speedup achieved by different schemes under two vacation conditions: Vacation-Low and Vacation-High. The y-axis represents speedup, while the x-axis shows the number of processors (1, 2, 4, 8). The graph illustrates the performance enhancement as the number of processors increases.
Thank You!

Questions?
Backup Slides
Summary: TMACC

- A hybrid TM scheme
  - Offloads conflict detection to external HW
  - Saves instructions and meta-data
  - Requires no core modification

- Prototyped on FARM
  - First actual implementation of Hybrid TM
  - Prototyping gave far more insight than simulation.

- Very effective for medium-to-large sized transactions
  - Small transaction performance gets better with ASIC or on-chip implementation.
  - Possible future combination with best-effort HTM
What can I prototype with FARM?

- **Question**
  - What units/nodes can I put together?
  - What functions can I put on FPGA units?

- Heterogeneous systems
- Co-processor or off-chip accelerator
- Intelligent memory system
- Intelligent I/O device
- Emulation of future large scale CMP system
Verification Environment

- Bus Functional Model
  - cHT Simulator from AMD
  - Cycle-based
  - HDL co-simulation via PLI interface

- FARM SimLib
  - A glue library that connects high-level test-benches to cycle-based BFM

- High-level test-bench
  - Simple Read/Write + Imperative description + Complex functionality ...
  - Concept similar to Synopsis VERA or Cadence Specman

```
... v1 = Read (Addr1);
v2 = Read (Addr2);
v3 = foo (v1, v2);
Delay (N);
Write(Addr3, v3);
```
### Implementation Result

- We prototyped TMACC on FARM
- HW Resource Usage

<table>
<thead>
<tr>
<th></th>
<th>Comm. IP</th>
<th>TMACC-GE</th>
<th>TMACC-LE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4Kb BRAM</td>
<td>144 (24%)</td>
<td>256 (42%)</td>
<td>296 (49%)</td>
</tr>
<tr>
<td>Logic Register</td>
<td>16K (15%)</td>
<td>24K (22%)</td>
<td>24K (22%)</td>
</tr>
<tr>
<td>LUT</td>
<td>20K</td>
<td>30K</td>
<td>35K</td>
</tr>
<tr>
<td>FPGA Type</td>
<td></td>
<td>Altera Stratix II EPS130 (-3)</td>
<td></td>
</tr>
<tr>
<td>Max Freq</td>
<td></td>
<td>100 MHz</td>
<td></td>
</tr>
</tbody>
</table>
### Tables

<table>
<thead>
<tr>
<th>Name</th>
<th>Input parameters</th>
<th>RD/tx</th>
<th>WR/tx</th>
<th>CPU cycles/tx</th>
<th>Memory usage (MB)</th>
<th>Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>vacation-low</td>
<td>n2 q90 u98 r1048576 t4194304</td>
<td>220.9</td>
<td>5.5</td>
<td>37740</td>
<td>573</td>
<td>very low</td>
</tr>
<tr>
<td>vacation-high</td>
<td>n4 q60 u90 r1048576 t4194304</td>
<td>302.14</td>
<td>8.5</td>
<td>37642</td>
<td>573</td>
<td>low</td>
</tr>
<tr>
<td>genome</td>
<td>g16384 s64 n16777216</td>
<td>55.8</td>
<td>1.9</td>
<td>48836</td>
<td>1932</td>
<td>low</td>
</tr>
<tr>
<td>labyrinth</td>
<td>x512-y512-z7-n512.txt</td>
<td>180</td>
<td>177</td>
<td>$6.1 \times 10^9$</td>
<td>32</td>
<td>high</td>
</tr>
<tr>
<td>scca2</td>
<td>s20 i1.0 u1.0 13 p3</td>
<td>1</td>
<td>2</td>
<td>2360</td>
<td>1320</td>
<td>very low</td>
</tr>
<tr>
<td>kmeans-low</td>
<td>m40 n40 65536-d32-c16.txt</td>
<td>25</td>
<td>25</td>
<td>680</td>
<td>16</td>
<td>low</td>
</tr>
<tr>
<td>kmeans-high</td>
<td>m256 n256 65536-d32-c16.txt</td>
<td>25</td>
<td>25</td>
<td>690</td>
<td>16</td>
<td>high</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parameter set label</th>
<th>A1*sizeof(int)</th>
<th>A2</th>
<th>R</th>
<th>W</th>
<th>C</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) working-set size</td>
<td>0.5 ~ 64 (MB)</td>
<td>-</td>
<td>80</td>
<td>4</td>
<td>false</td>
<td>8</td>
</tr>
<tr>
<td>(b) transaction size</td>
<td>64 (MB)</td>
<td>-</td>
<td>10 ~ 400</td>
<td>max(1, R * 0.05)</td>
<td>false</td>
<td>8</td>
</tr>
<tr>
<td>(c) true conflicts</td>
<td>64 (MB)</td>
<td>256 ~ 16,384</td>
<td>40</td>
<td>2</td>
<td>true</td>
<td>8</td>
</tr>
<tr>
<td>(d) write-set sizes</td>
<td>64 (MB)</td>
<td>-</td>
<td>80</td>
<td>1 ~ 128</td>
<td>false</td>
<td>8</td>
</tr>
<tr>
<td>(e) # of threads (med-sized TX)</td>
<td>64 (MB)</td>
<td>-</td>
<td>80</td>
<td>4</td>
<td>false</td>
<td>1 ~ 8</td>
</tr>
<tr>
<td>(f) # of threads (small-sized TX)</td>
<td>64 (MB)</td>
<td>-</td>
<td>4</td>
<td>1</td>
<td>false</td>
<td>1 ~ 8</td>
</tr>
</tbody>
</table>
Graphs

(a) impact of working-set size

(b) impact of transaction size

(c) impact of true conflicts

(d) impact of write-set size

(e) impact of number of threads
Graphs (projection)

(d) impact of write-set size

(e) impact of number of threads

(f) performance comparison of TMACC-GE and TL2 for short sized transactions

<table>
<thead>
<tr>
<th>WR</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>RD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Legend:
1. TL2 performs better by more than 3%
2. Two schemes show similar performance
3. TMACC-GE performs better by more than 3%
Hardware Acceleration

- FARM is ideal vehicle for evaluating accelerators
  - FPGA closely coupled with CPUs
- High level analytical model for accelerator speedup:

\[
\text{Speedup} = \frac{G(T_{on} + T_{off})}{G(T_{on} + \alpha T_{off}) + t_{ovhd} - t_{ovlp}}
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{off})</td>
<td>Time to execute the offloaded work on the processor</td>
</tr>
<tr>
<td>(\alpha)</td>
<td>Acceleration factor for the offloaded work (doubled rate would have (\alpha = 0.5))</td>
</tr>
<tr>
<td>(T_{on})</td>
<td>Time to execute remaining work (i.e. unaccelerated work) on the processor</td>
</tr>
<tr>
<td>(G)</td>
<td>Percentage of offloaded work done between each communication with the accelerator</td>
</tr>
<tr>
<td>(t_{ovlp})</td>
<td>Time the processor is doing work in parallel with communication and/or work done on the accelerator</td>
</tr>
<tr>
<td>(t_{ovhd})</td>
<td>Communication overhead</td>
</tr>
</tbody>
</table>

(A) Fully Synchronous  
(B) Merged Return (Half Synchronous)  
(C) Asynchronous
Analytical Model

- **b**: breakeven point for half-synch model
- **a**: breakeven point for full synch model
Initial Application: Transactional Memory

- Accelerate STM without changing the processor
  - Use FPGA in FARM to detect conflicts between transactions
  - Significantly improve expensive read barriers in STM systems
  - Can use FPGA to atomically perform transaction commit
    - Provides strong isolation from non-transactional access
    - Not used in current rendition of FARM
What’s inside TMACC HW?

- A set of generic BloomFilters + control logic
  - (BloomFilter: a condense way to store ‘set’ information)
  - Read-set: Addresses that a thread has read
  - Write-set: Addresses that other threads have written

- Conflict detection
  - Compare read-address against write-set
  - Compare write-address against read-set
Problem of Being Off-Core

- Asynchronous communications
- Variable latency to reach the HW
  - Network latency
  - Amount of time spent in the store buffer
- How can we determine correct ordering?
Global and Local Epochs

- **Global Epochs**
  - Each command embeds *epoch number (a global variable)*.
  - Finer grain but requires global state
  - Know A < B, C but nothing about B and C

- **Local Epochs**
  - Each threads declare start of new epoch
  - Cheaper, but coarser grain (non-overlapping epochs)
  - Know C < B, but nothing about A and B or A and C
Two TMACC Schemes

- We proposed two TM schemes.
  - One using global epoch (TMACC-GE); the other using local epoch (TMACC-LE)

- Trade-Offs
  - TMACC-GE is more accurate in conflict detection. (i.e. less false positives)
  - TMACC-GE has more SW overhead. (i.e. global epoch management)
  - TMACC-LE uses even less meta-data.
    - It allows, but detects, reading partial-committed data.
  - TMACC-LE is more expensive in HW resource.
    - Due to BloomFilter copy operation

- Misc. optimizations
  - Global epoch merging, private global epoch, local epoch splitting ...
Performance Analysis: micro-benchmark

- Why micro-benchmark?
  - Simple and easy to understand
  - Free from pathologies and 2nd-order effects ➔ Focus on overhead
  - Decouple effects of parameters

- Parameters
  - Size of Working Set (A1)
  - Size of Transaction; Number of Read/Writes (R,W)
  - Degree of Conflicts (C, A2)

- Implementation
  - Random array accesses
  - Array1[A1]: partitioned (non-conflicting)
  - Array2[A2]: fully-shared (possible conflicts)

---

Parameters: A1, A2, R, W, C

```
TM_BEGIN
for I = 1 to (R + W) {
  p = (R / R + W)

  /* Non-conflicting Access */
  a1 = rand(0, A1 / N) + tid * A1/N;
  if (rand_f(0,1) < p))
    TM_READ(Array1[a1])
  else
    TM_WRITE(Array1[a1])

  /* Conflicting Access */
  if (C) {
    a2 = rand(0, A2);
    if (rand_f(0,1) < p))
      TM_READ(Array2[a2])
    else
      TM_WRITE(Array2[a2])
  }
}
TM_END
```
**Micro-benchmark Results**

(a) Working set size (A1)
- The knee is size of cache.
- Constant spread-out of speed-ups

(b) Transaction size (R; W = R *.05)
- All violations are false positive.
- Plateau in the middle; drop for small-sized TXs.

- TL2: baseline STM
- Unprotected: upper-bound of performance
- Y-axis
  - Speed up with 8 cores.
  - % of violation
FPGA Breakdown

- Cache
- RSM
- Committer

HT Interface

HT Core

HT
CPU → FPGA Communication

- **Driver**
  - Modify system registers to create DRAM address space mapped to FPGA
    - “Unlimited” size (40 bit addresses)
  - User application maps addresses to virtual space using mmap
  - No kernel changes necessary
CPU → FPGA Commands

- Uncached stores
  - Half-synchronous communication
  - Writes strictly ordered

- Write combining buffers
  - Asynchronous until buffer overflow
  - Command offset: configure addresses to maximize merging

- DMA
  - Fully asynchronous
  - Write to cached memory and pull from FPGA
FPGA → CPU Communication

- FPGA writes to coherent memory
  - Need a static physical address (e.g. pinned page cache) or coherent TLB on FPGA
  - Asynchronous but expensive, usually involves stealing a cache line from CPUs...

- CPU reads memory mapped registers on the FPGA
  - Synchronous, but efficient
Communication in TM

- **CPU → FPGA**
  - Use write-combining buffer
  - DMA not needed, yet.

- **FPGA → CPU**
  - Violation notification uses coherent writes
    - Free incremental validation
  - Final validation uses MMR
Tolerating FPGA-CPU Latency

- Decouple timeline of CPU command firing from FPGA reception
  - Embed a global time stamp in commands to FPGA
  - Software or hardware increments time stamp when necessary
    - Divides time into “epochs”
    - Currently using atomic increment – looking into Lamport clocks
  - FPGA uses time stamp to reason about ordering
Example: Use in TM

- **Read Barrier**
  - Send command with global timestamp and read reference to FPGA
  - FPGA maintains per-txn bloom filter

- **Commit**
  - Send commands with global timestamp and each written reference to FPGA
  - FPGA notifies of already known violations
  - Maintains a bloom filter for this epoch
    - Violates new reads with same epoch
Time Stamp illustration

CPU 0
Read x

CPU 1
Start Commit
Lock x
Violate x

FPGA
Synchronization “Fence”

- Occasionally you need to synchronize
  - E.g. TM validation before commit
  - Decoupling FPGA/CPU makes this expensive – should be rare
- Send fence command to FPGA
- FPGA notifies CPU when done
  - Initially used coherent write – too expensive
  - Improved: CPU reads MMR
Results

Single thread execution breakdown for STAMP apps

![Bar chart showing single thread execution breakdown for STAMP apps.](Image)
Results

Speedup over sequential execution for STAMP apps

- **Vacation (Low)**
- **Yada**
- **Genome**
- **Labyrinth**
- **SSCA2**
- **Intruder**
Classic Lessons

- Bandwidth

- CPU vs Simulator
  - In-order single-cycle CPUs do not look like modern processors (Opteron)

- Off chip is hard
  - CPUs optimized for caches not off-chip communication
Proof of Concept: Transactional Memory

- Prototype hardware acceleration for TM

Transactional Memory
- Optimistic concurrency control (programming model)
- Promise: simplifying parallel programming
- Problem: Implementation overhead
  - Hardware TM (HTM) – expensive
  - Software TM (STM) – slow
  - Hybrid TM

Idea
- Accelerate STM with *out-of-core* hardware (e.g. an off-chip accelerator)
- No core modification, but still good performance
Possible Directions

- Possibility of building a much bigger system (~28 cores)
- Security
  - Memory watchdog, encryption, etc.
- Traditional hardware accelerators
  - Scheduling, cryptography, video encoding, etc.
- Communication Accelerator
  - Partially-coherent cluster with FPGA connecting coherence domains
Let us accelerate you…

- How could your domain/app use an FPGA co-processor?