Scalable Coherence and Many-Core Architectures

CSC572: Seminar on Pervasive Parallelism

Raj Parihar
Department of Electrical and Computer Engineering
CMPs (2008)

- Intel Core-2 Duo
  - 2 cores
  - Shared bus with MESI protocol

- Sun Niagara II
  - 8 cores
  - Crossbar with directory protocol

- IBM Cell
  - 9 cores
  - Ring network/ no coherence
Current CMPs

• IBM Power7
  • 4, 6, 8 cores; Each core 4-way SMT
  • Broadcast based coherence on rings

• Intel Itanium and Nehelam-EX
  • 4 and 8 cores; 2-way hyper threading
  • QuickPath based interconnect

• AMD Istanbul
  • 6 cores
  • HyperTransport based interconnect
Future CMPs

- 32+ cores on single chip
- Scalable interconnect (mesh, packet based)
- Example
  - Intel’s Polaris prototype
  - 80 cores
- Cache coherence?
Motivation

- Future processors would be many-core (32+ cores)
- Shared memory programming is popular
- Applications assume cache coherent memory system

- Hardware based cache coherence solutions
  - Snooping on shared bus/ broadcast medium
  - “Centralized” directory on shared cache
  - Distributed directory on point-to-point interconnect

- How do these options adapt to future CMP?
Hardware Cache Coherence

- Snooping on a shared bus (including ring)
  - Easy to implement
  - Shared bus limits the scalability (BW saturates ~ 8 cores)
- “Centralized” directory on shared caches
  - Xbar used to connect L1 and lower level caches
  - Xbar limits scalability (Area)
    - High area overhead due to all-to-all connections
- Distributed directory
  - Scales to 100+ nodes
  - Complex and difficult to verify
    - Multiple transition states; Forwarding of messages
    - Transaction may open for long time i.e. write invalidations
Software Based Coherence

- Software DSM systems
  - Cache are private and software maintains the coherence

- Advantages
  - Can maintain the coherence at page level, as oppose to block level, to avoid some of the tracking overhead
  - Flexible and can be modified (bug fixing) even after the product is shipped

- Major drawbacks
  - Require moving, comparing ("diff"), and copying data in physical memory pages to enforce coherence
Proposed Solution

• Avoids incoherence as oppose to maintain coherence

• Explores the mid section of cache coherence spectrum
  • One end: hardware-based cache coherence
  • Other end: software-based cache coherence

DSM
An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs

C. Fensch and M. Cintra
University of Edinburgh

(HPCA - 2008)
Baseline Tiled Architecture

- Point-to-point interconnect
  - Similar to MIT RAW, Tilera TILE64 and Intel Polaris
- Shown to be good for ILP- and DLP- based applications
- No support for shared-memory parallel applications
OS-Based Alternative: Key Ideas

- There is only one copy of each data item in all L1 caches
  - Thus no coherence problem
- Data is allocated to L1 caches on page level
  - New hardware table (similar to TLB) to lookup the location of data; called MAP
  - Page table extended to include mapping of V-page to tile
  - OS controls placement
    - Defaults mapping policy: First –touch
    - Other possible policies: application/ compiler mapping hints
- Hardware supports remote cache accesses

Should work well if most data is placed locally and the distances/latency on chip are not too high
Data Placement Choices

- Proposed Scheme: Place virtual pages to architectural tiles
- Two ends of spectrum: Completely static and Completely dynamic
- Why not simply divide the physical address space across tiles and use virtual-to-physical mapping to place pages?
  - More restrictive; Fragmentation of main memory
  - More difficult to change mapping
    - Pages (mapped to the tiles) have to be moved back to memory for new mapping
Remote Cache Access

![Diagram of Remote Cache Access](image-url)
Remote Cache Access – Local Load

1. Use v-addr to index cache, TLB, MAP.
   Prepare net request.
2. Local/Remote Request?
3. Cancel net request.
4. Tag comparison.
Remote Cache Access - Remote Load

1. Same as before
2. Local/Remote Request?
3. Cancel cache/TLB. Create net request.
4. Send msg and receive it.
5. Index remote cache.
6. Tag comparison.
7. Send answer.
Improvements over Baseline System

• Page Migration (Periodically done at the barriers)
  • Pages can be re-assigned to a different tile at barriers
  • Requires flushes of caches at barriers, and system call to reset to OS MAP table

• Sharing of Read-only pages
  • Read-only pages can be mapped to all tiles locally
  • The first tile which tries to write, traps to the OS and becomes the owner of that page
  • Once ownership is established, it is fixed and cannot be transferred
  • Requires invalidation of MAP tables on Lock Acquire to prevent access to stale local data
Allowing Migration of Data

• Must be done at the quiescent state: e.g., barriers
• To migrate pages
  • Invalidate both local and OS MAP entries
  • Write-back dirty cache lines
  • Invalidate cache contents
  • Following access would generate a (possibly) new mapping

• Instructions for cache write-back and invalidate already exist in most ISA
• Need an instruction to invalidate local MAP table
Example: Migration of Data

Intuitively at the barriers, access pattern changes;
Good time for new mapping

1. P0: ld 0xA000 (p→P0)
2. P1: ld 0xA010 (remote)
3. P0: st 0xA020
4. Barrier (write-back + inv. L1, inv. MAP)
5. P1: ld 0xA010 (p→P1)
6. P0: ld 0xA000 (remote)

• Intuitively at the barriers, access pattern changes;
• Good time for new mapping

Can be done anytime – OS needs to issue a system call to reset the MAP tables
Allowing Replication of Read-Only Data

- Multiple readers but single, \textit{fixed}, write
- \textbf{Caveat}: assumes release consistency (RC) applications
- \textbf{Operation}
  - Processors reading data get a local mapping in MAP table
  - First write traps to OS and tile becomes the owner of the page
  - Subsequent reads continue with old local mapping
  - Subsequent writes, and reads without mapping, are directed to owner tile
  - At lock acquire MAP entries have to be invalidated
  - At barriers same procedure as for migration

- \textbf{Key features}
  - OS does not have to keep track of sharers
  - Ownership of modified data is not transferred
Example: Replication of Read-Only Data

1. P0: Id 0xA000 (p→P0)
2. P1: Id 0xA010 (p→P1)
3. P0: st 0xA020 (owner)
4. Barrier (write-back + inv. L1, inv. MAP)
5. P0: Id 0xA020 (p→P0)
6. P1: Id 0xA010 (p→P1)
7. P1: st 0xA000 (owner)
8. P0: Acq Lock (inv. local MAP)
9. P0: Id 0xA000 (p→P1)
Secondary Issues Addressed

- Synchronization primitives to implement locks and barriers
  - *Compare & swap (C&S)*: Easy to implement because there is no replication of lock variable in multiple caches, easier to enforce atomicity
  - *Load link (LL) and store conditional (SC)*: Harder to implement because RESERVE registers can not be kept it local L1 due to incoherence

- Multi-level on-chip cache hierarchies
  - The proposed design assumes single level of caches which is L1 per time
  - Whole idea can be extended to multi-level cache hierarchies i.e. L2 and L3

- Cost comparison with Directory coherence
  - RAC+MAP are simpler than directory controller due to less # of states
  - Storage: For 32-tile system – 352 bytes (MAP) vs 4K bytes (DIR)
Synchronization

- Two options to implement synchronization
  - Compare & Swap; LL and SS
- **Compare & Swap** is easier to implement
  - No replication; Easier to enforce atomicity
  - Requires compare logic to cache controllers
- **Load Link (LL) and Store Conditional (SC)**
  - Harder than a system with cache coherence
  - Typically done by keeping a reserve register in local L1 and rely on cache coherence to detect the conflicting stores
  - In this case, keep the reserve register in home L1
  - Livelock is possible when processors try to attempt to lock **different lock variables** on same page
    - Solution: Once written can not be overwritten by other LL

(a) Lock acquire in a system with cache coherence.

(b) Livelock.
Multiple processors can obtain same lock simultaneously
- Can happen when LL and SC from three processors are interleaved in such a way that 2nd SC succeeds because it matches with 3rd LL
- Solution: Extend reserve register with the ID of tile that successfully sets it and consider SC successful that matches the value from same tile

One more (final) issue
- Processor holding the reserve register fails to issue matching SC, either accidently or maliciously
- Solution: Timeout mechanism to clear the reserve register
Evaluation Environment

- **Simulator**
  - Pipelined, single-issue
  - Network model: Liberty
  - Caches: SimpleScalar

- **Compiler**
  - Gcc 3.4.4 for PowerPC

- **Benchmarks**
  - Splash-2 (32 tiles)
  - ALPBench (16 tiles)

- **System simulated/ compared**
  - Proposed scheme with migration+sharing
  - Idealized distribute directory coherence
  - Software DSM

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
<th>Instr.</th>
<th>Lock</th>
<th>Barr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholesky</td>
<td>tk29.O</td>
<td>1,234M</td>
<td>72,075</td>
<td>3</td>
</tr>
<tr>
<td>FFT</td>
<td>65,536 points</td>
<td>58M</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>LU</td>
<td>512×512 matrix</td>
<td>389M</td>
<td>32</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>16x16 block</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>radix</td>
<td>262,144 keys</td>
<td>54M</td>
<td>406</td>
<td>12</td>
</tr>
<tr>
<td>barnes</td>
<td>16,384 particles</td>
<td>4,361M</td>
<td>69,360</td>
<td>18</td>
</tr>
<tr>
<td>fmm</td>
<td>16,384 particles</td>
<td>2,903M</td>
<td>47,074</td>
<td>34</td>
</tr>
<tr>
<td>ocean</td>
<td>258x258 grid</td>
<td>412M</td>
<td>6,656</td>
<td>900</td>
</tr>
<tr>
<td>radiosity</td>
<td>demo</td>
<td>646M</td>
<td>281,217</td>
<td>19</td>
</tr>
<tr>
<td>raytrace</td>
<td>car</td>
<td>2,006M</td>
<td>95,528</td>
<td>2</td>
</tr>
<tr>
<td>volrend</td>
<td>head</td>
<td>1,344M</td>
<td>38,604</td>
<td>20</td>
</tr>
<tr>
<td>water-nsq</td>
<td>512 molecules</td>
<td>652M</td>
<td>35,360</td>
<td>19</td>
</tr>
<tr>
<td>water-spa</td>
<td>512 molecules</td>
<td>664M</td>
<td>609</td>
<td>19</td>
</tr>
<tr>
<td>facerec</td>
<td>ALP Training</td>
<td>2,826M</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>mpegdec</td>
<td>525_tens_040.m2v</td>
<td>1,049M</td>
<td>29</td>
<td>41</td>
</tr>
<tr>
<td>mpegdec</td>
<td>Output of mpegdec</td>
<td>9,477M</td>
<td>29</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>32K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit latency</td>
<td>3 cycl.</td>
</tr>
<tr>
<td>Miss latency</td>
<td>200+16 cycl.</td>
</tr>
<tr>
<td>Line size</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way</td>
</tr>
<tr>
<td>Writeback buf.</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entries</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page size</td>
<td>4K</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way</td>
</tr>
<tr>
<td>Hit latency</td>
<td>1 cycl.</td>
</tr>
<tr>
<td>Miss latency</td>
<td>200 cycl.</td>
</tr>
<tr>
<td>RAC input queue</td>
<td>32 entry</td>
</tr>
</tbody>
</table>
Speedup over Directory Coherence

- Average gap: 16%, Maximum gap: 32.4%
Speedup over Software DSM

- Speedup for 32 (splash-2) and 16 (ALP Bench) tiles compare to the execution time of single tile
  - SW DSM performs very well on applications that use barriers for synchronization; Possibly due to simplifications
Memory Access

- Remote accesses are very rare for most benchmarks
Load Latency

Although remote access latency is quite high compared to local, remote accesses are rare.
Conclusion

- Alternative to full hardware distributed directory cache coherence that divides work between OS and hardware

- Mostly acceptable performance degradation
  - Average: 16%; Maximum: 32% (radiosity)

- Likely to be less complex than distributed directory coherence
  - Only request-response transactions
  - No forwarding or multiple invalidations necessary
  - Less storage required compared to directory state
Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator


University of Illinois at Urbana-Champaign
(ISCA 2009)
Many Core Accelerator Architectures

• Programmable many core accelerator architectures
  • Hardware entity designed to provide advantages for a class of applications compare to general-purpose CPU
  • Higher performance and high compute density (GFLOPS/mm^2)
  • Lower power and good energy efficiency (pJ/ operation)

• Challenges
  • Inflexible programming models (much of the focus is on this)
  • Lack of conventional memory model (i.e. strict memory models)
  • Hard to scale for irregular parallel applications

• Effect on development
  • Longer time to develop solutions, buggy software
Elements of Many Core System

**Work Distribution**
Scheduling of work onto available chip resources in order to maximize throughput.  
*(Task based distribution, Fixed or parallel loop iterations)*

**Locality Management**
Co-location of tasks onto shared processing resources to reduce latency and frequency of communications  
*(Shared caches among clusters)*

**Synchronization**
Primitives to support fine-grained tasks and synchronize at the end of parallel section  
*(Barriers, fetch-and-fai)*

**Memory Model**
Involves choice of memory hierarchy, inter-processor communication and allowable memory orderings  
*(SW managed memory or multiple address space, explicit vs implicit inter-processor communication)*

**Execution Model**
Mapping of the tasks (application binary) to the processor functional units  
*(ISA, specialized instructions, VLIW vs out-of-order, SIMD vs MIMD, MT)*
Many Core Computing: Today

- Two dominant multicore computing paradigm

  **GP GPUs**
  - 1000s of cores
  - Achieves scalability
  - Inflexible programming

  **GP CMPs**
  - 8 - 32 cores
  - Achieves scalability
  - Flexible programming
  - Best of both world
  - Poor scalability
  - Easy to program
Rigel: Cluster View

- Cluster: Basic building block
  - 64 kB shared WB data cache (Cluster $)

- Eight 32-bit RISC cores/cluster
  - Dual-issue in-order core
  - Per-core fully pipelined SP FPUs
  - 32- GPR regs; Independent fetch unit

- Rigel Architecture (1024 cores)
  - Chip area: 320 mm² @ 45nm
  - Peak throughput: 2.4 TFLOPS @ 1.2 GHz
Rigel: Full Chip View

- Cluster caches not hardware coherent (8 MB total)
- Global-cache memory controllers are the point where coherence is maintained (4 MB total)
- Uniform cache access to global caches

8 Tiles Per Chip
16 Clusters per Tile

Global Interconnect

Global Cache Banks

8 Global Double Data Rate (GDDR) Channels
Rigel Programming Model

• Rigel task model API operations
  • Managing the resources of queues located in the memory
  • Inserting and removing units of work at those queues
  • Atomic primitives to enforce coherence in SW

• Queue management API calls
  • TQ_Create, TQ.EnqueueGroup, TQ_Dequeue, TQ.Enqueue

• Scheduling and locality for similar tasks
  • Shared cluster cache enables low-cost fine-grain communications among the tasks within task group

• Atomic primitives
  • Atomic fetch, increment and addition for global cache
  • Load-link and store-conditional for cluster caches
Rigel Programming Model (Cont…)

- Rigel Task Model uses SPMD execution paradigm where all cores share a single address space
- Parallel work units, referred as tasks, are inserted and removed from the queues between barriers
  - Period between two barriers is *interval* of computation
  - The barriers provide *partial* ordering of tasks
  - Barriers are used to synchronize the execution of all cores using a queue, which is managed in memory and can be cached in global cache – L3
- Task model presents programmer a monolithic global queue, but is implemented using hierarchical task queues
Benchmark Overview

- Conjugate Gradient Linear Solver (cg)
  - Performed on sparse matrices from the Harwell-Boeing collection of dimension 4884 (147,631 non-zero elements)

- K-Means Clustering (kmeans)
  - A computer vision kernel, performed on 16k element 18-dimensional data sets

- Dense-Matrix Multiply (dmm)
  - 1024x1024 blocked dense matrix multiply

- GJK Collision Detection (gjk)
  - For a scene consisting of 512 randomly-generated convex polyhedra of varying size and complexity

- Heat Transfer simulation (heat)
  - An iterative 2D stencil computation that performs a heat transfer simulation on a 4096x512 element grid

- Medical Image Reconstruction Kernel (mri)
Evaluation and Comparison

- Baseline has global cache and memory BW equal to that of a full tile of 128 cores
- Rigel Cluster of 8-cores
  - 16 GFLOPS(peak) @ 1.2 GHz in 2 mm² at 45nm
- Compare a contemporary Intel i7 quad core
  - 25 GFLOPS(peak); 4-wide SIMD, 10x area, 3x Freq

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Power ($\frac{W}{mm^2}$)</th>
<th>Perf. ($\frac{GOPS}{mm^2}$)</th>
<th>Machine Balance ($\frac{GBPS}{GOPS}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CellBE</td>
<td>.3</td>
<td>1.8</td>
<td>.13</td>
</tr>
<tr>
<td>Intel Quad-core</td>
<td>.5</td>
<td>.4</td>
<td>.25</td>
</tr>
<tr>
<td>NVIDIA GTX280</td>
<td>.3–.4</td>
<td>3.3</td>
<td>.14</td>
</tr>
<tr>
<td>ATI R700</td>
<td>.55–.9</td>
<td>6.4</td>
<td>.1</td>
</tr>
<tr>
<td>Rigel</td>
<td>.3</td>
<td>8</td>
<td>.05</td>
</tr>
</tbody>
</table>

Table 1: Power, area, and performance comparison of Rigel to accelerators normalized to 45nm.
Rigel: Task Model Evaluation

• The cost of dequeue and enqueue operations limits the minimum exploitable task length – which is usually 10s to 1000s of cycles

• The overhead of barriers, enqueue, and dequeue as a % of task length is not more than a few % for all runs other than cg
  • Cg’s nature limits the enqueuing cores’ ability to distribute the work fast
Rigel: Summary

- Rigel architecture, based on SPMD model, is presented
  - A case for software maintained coherence; achieves scalability

- Task management requires minimal additional hardware

- 1000-core accelerator achieves
  - Area/performance: 8 GFLOPS/mm² @ ~100 mW at 45 nm
  - Programmability: Task based API + MIMD execution

- Crucial to support fast task enqueue, dequeue and barriers
Cohesion: A Hybrid Memory Model for Accelerators

John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, Sanjay J. Patel

University of Illinois at Urbana-Champaign
(ISCA 2010)
Many Core Computing: Today

- Two dominant multicore computing paradigm

GP GPUs
1000s of cores

Software-managed Coherence Protocol

Achieves scalability
- Inflexible programming

Achieves scalability
- Flexible programming
- Best of both world

GP CMPs
8 - 32 cores

Hardware-managed Coherence Protocol

Cohesion
SW-to-HW Transitions

- Poor scalability
- Easy to program
CMP Memory Model Choices

**Conventional Multicore CPU**
- Ex: Intel i7, Sun Niagara
- Optimized for:
  - Minimal latency
  - Tightly coupled sharing
  - Fine-grained synchronization
  - Minimal programmer effort
- Provides:
  - Single address space
  - Hardware caching
  - Strong ordering
  - HW-managed coherence

**Contemporary Accelerator**
- Ex: NVIDIA GPU, IBM Cell
- Optimized for:
  - Maximum throughput
  - Loosely coupled sharing
  - Coarse-grained synchronization
  - Short silicon design cycle
- Provides:
  - Multiple address spaces
  - Scratchpad memories
  - Relaxed ordering
  - SW-managed coherence

Source: J. Kelm’s ISCA-2010 presentation
COHESION: Overview

- A system with a single address space that supports a variety of coherence domains: SWcc and HWcc
- The ability to perform temporal reallocation between coherence domains without copies
- Fine-granularity cache coherence management (cache line level)
SWcc vs HWcc

- **SWcc**: Embedded in compiler, runtime, programming model (*Push*)
  - No memory is required for directories
  - No design effort is needed to verify the coherence protocol
  - Less network traffic and relaxed design constraints compare to HWcc
  - Reduced instruction stream efficiency due to additional explicit cache flush and invalidate instructions
  - Flush instructions whose targets are evicted are wasted

- **HWcc**: Implemented and tracked in hardware (*Pull* mechanism)
  - Can enforce strict memory models; Provides good programmability
  - Tracking the state of lines in few bits is more efficient than SWcc
  - Enables easier speculative prefetching and data migration than SWcc
  - Invalidation of shared data requires many messages; Also lengthens the critical path
Network Traffic Reduction

- HWcc: source of additional messages
  - Write misses and read release/invalidation operation
- SWcc has the ability to mass invalidate shared read data, signaling many invalidations with only a few messages
  - Suffers from additional instructions which are for cache flush and invalidation
  - Some of those instructions are superfluous (L2 line for which the target is evicted)
Baseline Architecture

- Baseline: Variant of Rigel architecture
  - 1024-core CMP, HW caches, single address space, MIMD
  - Default: Keep all of memory coherence in HWcc domain

Each core has private L1 data (1kB) and instruction (2kB) cache
## Baseline: Architectural Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>1024</td>
<td>–</td>
</tr>
<tr>
<td>Memory BW</td>
<td>192</td>
<td>GB/s</td>
</tr>
<tr>
<td>DRAM Channels</td>
<td>8</td>
<td>–</td>
</tr>
<tr>
<td>L1I Size</td>
<td>2</td>
<td>KB</td>
</tr>
<tr>
<td>L1D Size</td>
<td>1</td>
<td>KB</td>
</tr>
<tr>
<td>L2 Size</td>
<td>64</td>
<td>KB</td>
</tr>
<tr>
<td>L2 Size (Total)</td>
<td>8</td>
<td>MB</td>
</tr>
<tr>
<td>L3 Size</td>
<td>4</td>
<td>MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Size</td>
<td>32</td>
<td>bytes</td>
</tr>
<tr>
<td>Core Freq.</td>
<td>1.5</td>
<td>GHz</td>
</tr>
<tr>
<td>DRAM Type</td>
<td>GDDR5</td>
<td>–</td>
</tr>
<tr>
<td>L1I Assoc.</td>
<td>2</td>
<td>way</td>
</tr>
<tr>
<td>L1D Assoc.</td>
<td>2</td>
<td>way</td>
</tr>
<tr>
<td>L2 Assoc.</td>
<td>16</td>
<td>way</td>
</tr>
<tr>
<td>L2 Latency</td>
<td>4</td>
<td>clks</td>
</tr>
<tr>
<td>L3 Assoc.</td>
<td>8</td>
<td>way</td>
</tr>
</tbody>
</table>

### Directory Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory Size (realistic)</td>
<td>16K</td>
<td>entries</td>
</tr>
<tr>
<td>Directory Assoc. (realistic)</td>
<td>128</td>
<td>ways</td>
</tr>
<tr>
<td>Directory Size (optimistic)</td>
<td>∞</td>
<td>entries</td>
</tr>
<tr>
<td>Directory Assoc. (optimistic)</td>
<td>Full</td>
<td>–</td>
</tr>
<tr>
<td>L2 Ports</td>
<td>2</td>
<td>R/W</td>
</tr>
<tr>
<td>L3 Ports</td>
<td>1</td>
<td>R/W</td>
</tr>
<tr>
<td>L3 Latency</td>
<td>16+</td>
<td>clks</td>
</tr>
<tr>
<td>L3 Banks</td>
<td>32</td>
<td>–</td>
</tr>
</tbody>
</table>
COHESION: Architecture

- HWcc protocol: simple MSI protocol (Sparse directory) -- Default
- SWcc protocol: Task centric memory model (CG Region Table)
- Transition protocol: Message based transition (FG Region Table)
  - Transition doesn’t require copy operation rather relies upon bunch of messages, sent to home, which perform the transition between two domain
**COHESION: Operations**

- **C1**: Dir Hit, L3 Hit; HWcc
- **C2**: Dir Miss, CGRT Hit; SWcc
- **C3**: SWcc lines are treated as incoherent by HW

**Sparse Directory** (One per L3$ bank)
- set$_0$
- set$_1$
- ... set$_n$, set$_n-1$

- **sharers**
- **tag**
- **I/M/S**

**Coarse-grain Region Table** (Global Table)
- code segment
- stack segment
- global data

**Fine-grain Region Table** (Strided across L3 banks)
- base_addr
- start_addr
- size
- valid

- Coherence bit vectors (1 bit/line in memory)
- 16 MB table
- 4 GB memory

---

- Minimum 1 cycle penalty for fine grain table lookup
- If bit is SET: treated similar to SWcc
- If bit is CLEAR: treated similar to HWcc
- Tables are setup by the runtime at initialization
COHESION: Domain Transitions

- Initiated by uncached read-modify-write performed by runtime on FG-RT
- Issuing core blocks until the transition completes
  - Multiple line state transitions requests are serialize by directory line-by-line
- Directory controllers is responsible for HWcc ↔ SWcc

- SW to HW transition: Cohesion controller probes L2’s to reconstruct states
- HW to SW transition
  - If line is dirty, directory sends a WB request to all writers
  - HW stops tracking these lines which are tracked by SW from now on

---

(a) $HW_{cc} \Rightarrow SW_{cc}$ conversion

(b) $SW_{cc} \Rightarrow HW_{cc}$ conversion
COHESION: Programmer-visible API

- Two kind of heaps in the system
  - A conventional C-style heap (coherent in HW)
  - Incoherent heap (not kept HWcc)
- Incoherent heap is used for data that may transition domains during execution

<table>
<thead>
<tr>
<th>API Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>void * malloc(size_t sz)</td>
</tr>
<tr>
<td>void free(void * hwccptr)</td>
</tr>
<tr>
<td>void * coh_malloc(size_t sz)</td>
</tr>
<tr>
<td>void coh_free(void * swccptr)</td>
</tr>
<tr>
<td>void coh_SWcc_region(void * ptr, size_t sz)</td>
</tr>
<tr>
<td>void coh_HWcc_region(void * ptr, size_t sz)</td>
</tr>
</tbody>
</table>
Runtime: COHESION, SWcc, HWcc

- Compared to realistic hardware assumptions, Cohesion delivers many times better performance due to
  - Reduced message traffic and
  - Reduction in the number of flush operations issued by the SWcc configurations, many of which may be unnecessary (L2 line is evicted)
Directory Size Sensitivity

- Reduces performance cliffs in sparse directory designs
- **Benefits**: Smaller on-die coherence structures
  - Area saving, Power saving, better scalability
COHESION: Summary

- COHESION: a hybrid memory model
  - Allows applications to combine the benefits of software and hardware coherence schemes into one unified model

- Significant reduction in traffic and area achieved in a hybrid model compared to hardware-only coherence
  - 2x reduction in message traffic
  - 2.1x reduction in directory utilization

- Aptly suited to heterogeneous systems
  - Potential to integrate host processors and accelerators under one memory model
Comparison: COHESION vs Rigel

• Similarities
  • Both target 1000+ core system and scale well
  • Both implement single address space; Flexible programming

• Differences (Cache Coherence and Memory Model)
  • Rigel: Primarily SW managed cache coherence (with minimal additional hardware)
  • Cohesion: Hybrid cache coherence which allows the transition between HW and SW managed coherent domain in runtime depending upon the application phases and nature
Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators

Yunsup Lee*, Rimas Avizienis*, Alex Bishara*, Richard Xia*, Derek Lockhart†, Christopher Batten†, and Krste Asanovi´c*

*UC Berkeley, †Cornell University
Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators

• Basic idea and intuitions
  • Paper presents set of 5 architectural design for DLP workloads
  • Compares their expected programmability* and efficiency*
  • Proposes a new vector-thread architecture, called Maven, which has greater efficiency and easier programmability

• General concepts
  • Data parallel applications: graphics rendering, computer vision, audio processing, physical simulation, and machine learning
  • Efficiency: energy and area efficiency
  • Programmability: How easy is it to write software for the accelerator?

A maven (also mavin) is a trusted expert in a particular field, who seeks to pass knowledge on to others. The word maven comes from Hebrew, via Yiddish, and means one who understands, based on an accumulation of knowledge; Source: Wikipedia
More General Concepts

• DLP systems
  • General purpose host processor
    • Executes system code, non-DLP application codes
    • Distributes DLP kernels/computations to the accelerators
  • Programmability
    • How easy is it to write programs/software for the accelerators?
    • Mostly measured by man-hours
  • Efficiency
    • Energy/task or tasks/second/area

• 5 architectures studied in the work
  • MIMD, vector-SIMD, subword-SIMD, SIMT, VT
Types of Data-Level Parallelism

(a) Regular DA, Regular CF

\begin{verbatim}
for ( i = 0; i < n; i++ )
  C[i] = x * A[i] + B[2*i];
\end{verbatim}

(b) Irregular DA, Regular CF

\begin{verbatim}
for ( i = 0; i < n; i++ )
  E[C[i]] = D[A[i]] + B[i];
\end{verbatim}

(c) Regular DA, Irregular CF

\begin{verbatim}
for ( i = 0; i < n; i++ )
  x = ( A[i] > 0 ) ? y : z;
  C[i] = x * A[i] + B[i];
\end{verbatim}

(d) Irregular DA, Irregular CF

\begin{verbatim}
for ( i = 0; i < n; i++ )
  if ( A[i] > 0 )
    C[i] = x * A[i] + B[i];
\end{verbatim}

(e) Irregular DA, Irregular CF

\begin{verbatim}
for ( i = 0; i < n; i++ )
  C[i] = false; j = 0;
  while ( !C[i] & (j < m) )
      C[i] = true;
\end{verbatim}

Extremely irregular DLP:

Task-level parallelism; TSP
Little difficulty in mapping regular and irregular DLP to uTs

Does not exploit DLP to improve area and energy efficiency

Example: 1000-core Rigel accelerator

(a) MIMD

```
for (i = 0; i < n; i++)
    if (A[i] > 0)
        C[i] = x * A[i] + B[i];
```

```
1 div  m, n, nthr
2 mul  t, m, tidx
3 add  a_ptr, t
4 add  b_ptr, t
5 add  c_ptr, t
6 sub  t, nthr, 1
7 br.neq t, tidx, ex
8 rem  m, n, nthr
9 ex:
10 load x, x_ptr
11 loop:
12 load a, a_ptr
13 br.eq a, 0, done
14 load b, b_ptr
15 mul t, x, a
16 add c, t, b
17 store c, c_ptr
18 done:
19 add a_ptr, 1
20 add b_ptr, 1
21 add c_ptr, 1
22 sub m, 1
23 br.neq m, 0, loop
```
Some instructions are executed only once by CT (inst. 1, 10-14)

For uTs memory accesses (4-5, 9), VMU can move data in blocks

Irregular DLP requires the use of vector flags to implement data dependent conditional control flow (6)

Complicated irregular DLP may require many flags and complex flag arithmetic

Example: T0 vector microprocessor
Subword-SIMD

Uses wide scalar registers and datapaths (often overlaid on a double-precision FP unit) to provide a “vector-like” unit

Example: IBM Cell Processor; Supports scalar and 16x8, 8x16, 4x32, 2x64-bit vector operations

Subword-SIMD have fixed-length vectors, memory alignment constraints, and limited support for irregular DLP

Vector SIMD is better suited for DLP so subword SIMD is not explored in this work
**SIMT**

SIMT is hybrid – combining MIMD logical view with vector SIMD microarchitecture

No CTs; HT has to manage uTs (often a block of uTs)

Example: NVIDIA Fermi graphics processor;
32 multithreaded SIMT cores each with 16 lanes

```plaintext
for (i = 0; i < n; i++)
    if (A[i] > 0)
        C[i] = x * A[i] + B[i];
```

SIMT provides simple way to map complex data-dependent CF with uT scalar branches (4)

uTs must redundantly execute some instructions (inst 1-2, 5-7)

Regular data accesses must be encoded to multiple scalar accesses (inst. 3, 8, 11)
Vector Thread

Hybrid design; HT manages CTs which in turn manage array of uTs

Like vector SIMD, CT can amortize control overheads and executes efficient vector memory instructions

Example: Maven and Scale processor

```
for ( i = 0; i < n; i++ )
    if ( A[i] > 0 )
        C[i] = x * A[i] + B[i];
```

Explicit start and stop instructions for uTs execution (inst. 8 and 22)
Programmability Efficiency Trade off

- MIMD
- Vector-SIMD
- subword SIMD
- SIMT
- Vector Thread (VT)

Efficiency (Area and Energy)

Expected Programmability

- Augmented minimal hardware mechanism for improved programmability
- Uses same RISC ISA for vector and scalar threads
  - Eliminates the need of an efficient vector compiler

Maven: VT + Extra minimal HW to improve programmability
Focus of the paper is to compare the various architectural design patterns with respect to a single data-parallel tile.
Vector-Based Core Microarchitecture

Vector Unit = Vector lanes + Vector Memory Unit + Vector Issue Unit

Vector lane = Unified 6r3w-port regfile + 5 Vector Functional Units (2 VAU + VLU + VSU + VGU)

VMU coordinates data movement between memory system and vector regfile

CP splits each vector memory instruction and sends it to VMU and VIU which issues it to VLU or VSU

Banked regfile saves area
# Evaluations Results: Cycle Time

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Num Cores</th>
<th>Num Regs</th>
<th>Num μTs</th>
<th>Per Core</th>
<th>Peak Throughput</th>
<th>Power</th>
<th>Total Area (mm²)</th>
<th>Cycle Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1md-c4r32§</td>
<td>4</td>
<td>32</td>
<td>4</td>
<td></td>
<td>4</td>
<td>4</td>
<td>149</td>
<td>3.7</td>
</tr>
<tr>
<td>m1md-c4r64§</td>
<td>4</td>
<td>64</td>
<td>8</td>
<td></td>
<td>4</td>
<td>4</td>
<td>216</td>
<td>4.0</td>
</tr>
<tr>
<td>m1md-c4r128§</td>
<td>4</td>
<td>128</td>
<td>16</td>
<td></td>
<td>4</td>
<td>4</td>
<td>242</td>
<td>4.2</td>
</tr>
<tr>
<td>m1md-c4r256§</td>
<td>4</td>
<td>256</td>
<td>32</td>
<td></td>
<td>4</td>
<td>4</td>
<td>299</td>
<td>4.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Num Cores</th>
<th>Num Lanes</th>
<th>Max vlen</th>
<th>Range</th>
<th>Per Core</th>
<th>Peak Throughput</th>
<th>Power</th>
<th>Total Area (mm²)</th>
<th>Cycle Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsimd-c4v1r256+bi§</td>
<td>4</td>
<td>1</td>
<td>8 – 32</td>
<td></td>
<td>256</td>
<td>4c + 16v</td>
<td>4l + 4s</td>
<td>396</td>
<td>5.6</td>
</tr>
<tr>
<td>vsimd-c1v4r256+bi§</td>
<td>1</td>
<td>4</td>
<td>32 – 128</td>
<td></td>
<td>256</td>
<td>1c + 16v</td>
<td>4l + 4s</td>
<td>224</td>
<td>3.9</td>
</tr>
<tr>
<td>vt-c4v1r256</td>
<td>4</td>
<td>1</td>
<td>8 – 32</td>
<td></td>
<td>256</td>
<td>4c + 16v</td>
<td>4l + 4s</td>
<td>428</td>
<td>6.3</td>
</tr>
<tr>
<td>vt-c4v1r256+b</td>
<td>4</td>
<td>1</td>
<td>8 – 32</td>
<td></td>
<td>256</td>
<td>4c + 16v</td>
<td>4l + 4s</td>
<td>404</td>
<td>5.6</td>
</tr>
<tr>
<td>vt-c4v1r256+bi</td>
<td>4</td>
<td>1</td>
<td>8 – 32</td>
<td></td>
<td>256</td>
<td>4c + 16v</td>
<td>4l + 4s</td>
<td>445</td>
<td>5.9</td>
</tr>
<tr>
<td>vt-c4v1r256+bi+2s</td>
<td>4</td>
<td>1</td>
<td>8 – 32</td>
<td></td>
<td>256</td>
<td>4c + 16v</td>
<td>4l + 4s</td>
<td>409</td>
<td>5.9</td>
</tr>
<tr>
<td>vt-c4v1r256+bi+2s+d§</td>
<td>4</td>
<td>1</td>
<td>8 – 32</td>
<td></td>
<td>256</td>
<td>4c + 16v</td>
<td>4l + 4s</td>
<td>410</td>
<td>5.9</td>
</tr>
<tr>
<td>vt-c1v4r256+bi+2s§</td>
<td>1</td>
<td>4</td>
<td>32</td>
<td></td>
<td>256</td>
<td>1c + 16v</td>
<td>4l + 4s</td>
<td>205</td>
<td>3.9</td>
</tr>
<tr>
<td>vt-c1v4r256+bi+2s+mc</td>
<td>1</td>
<td>4</td>
<td>32</td>
<td></td>
<td>256</td>
<td>1c + 16v</td>
<td>4l + 4s</td>
<td>223</td>
<td>4.0</td>
</tr>
</tbody>
</table>

**Table 1: Subset of Evaluated Tile Configurations** – Multi-core and multi-lane tiles for MIMD, vector-SIMD, and VT patterns. Configurations with § are used in Section 5.3. statistical power column is from post-PAR; simulated power column shows min/max across all gate-level simulations; configuration column: b = banked, bi = banked+int, 2s = 2-stack, d = density-time, mc = memory coalescing; num μTs column is the number of μTs supported with the default of 32 registers/μT; arith column: xc + yv = x CP ops and y vector unit ops per cycle; mem column: x1 + ys = x load elements and y store elements per cycle.
Area Comparison

Multi-core VT to a multi-core vector-SIMD tile => area overhead of 6% for extra VT mechanisms to improve programmability

![Area Comparison Diagram](image)

(a) Area Breakdown for Evaluated Tile Configurations

(b) ASIC Layout for vt-c4v1r256+bi+2s+d

Figure 7: Area and VLSI Layout for Tile Configurations – (a) area breakdown for each of the 22 tile configurations normalized to the mind-c4r32 tile, (b) ASIC layout for vt-c4v1r256+bi+2s+d with individual cores and memory crossbar highlighted.
Microarchitectural Tradeoffs

Adding per-bank integer ALU partially offsets the performance loss; Regfile improves energy and performance both.

Increasing uTs reduces performance due to overhead of spawn and join for more uTs.

Improves performance but at increased energy; increase is due to larger regfile.

Larger regfile access outweighs increased vector length for energy and performance.

In VT, adding more vector registers elements improves temporal amortization of the CP.

**Figure 8: Impact of Additional Physical Registers, Intra-Lane Regfile Banking, and Additional Per-Bank Integer ALUs – Results for multi-core MIMD and VT files running the bsearch-cmv microbenchmark.**

Banking a regfile reduces regfile access energy but decreases perf due to bank conflicts.

Larger regfile access outweighs increased vector length for energy and performance.

In VT, adding more vector registers elements improves temporal amortization of the CP.
Efficiency vs Performance and Area

Left to right is increase in irregularity in DLP

Adding more uTs in MIMD is not area effective; Load balancing becomes more challenging

Vector based machines are faster and energy-efficient than MIMD

VT is more efficient than vector-SIMD for first four kernels

MAVEN:
Vt-c1v4r256+bi+2s

Figure 11: Implementation Efficiency and Performance for MIMD, vector-SIMD, and VT Patterns Running Application Kernels – Each column is for different kernel. Legend at top. mimd-c4r256 is significantly worse and lies outside the axes for some graphs. There are no vector-SIMD implementations for strsearch and physics due to difficulty of implementing complex irregular DLP in hand-coded assembly. mcore = multi-core vector-SIMD/VT tiles, mlane = multi-lane vector-SIMD/VT tiles, r32 = MIMD tile with 32 registers (i.e., one µT).
Vector vs uT Memory Access

- uT memory access is 5x worse energy and 7x worse performance compare to vector memory accesses
- Memory coalescing recoups some of the lost performance; still not enough

Figure 10: Impact of Memory Coalescing – Results for multi-lane VT tile running *vvadd*. 
Summary

• Effective data-parallel accelerators
  • Must handle regular and irregular DLP efficiently
  • Also must retain good programmability

• Detailed VLSI implementations and evaluations confirm
  • Vector-based microarchitectures are more area and energy efficient even for fairly irregular DLP compare to scalar arch.

• Maven is a simpler vector-SIMD based system which provides greater efficiency and easier programmability
  • Includes efficient dynamic convergence for microthreads
  • ALUs distributed close to the banks within a banked vector register file
Cache Coherence: Evolution

- **Sparse Dir, 1990**
- **Directory, 1988**
- **Treadmarks, 1996**
- **Cashmere, 1997**
- **Snoopy, 1983**
- **GP GPU, 2005**
- **Rigel, 2009**
- **Cohesion, 2010**
- **OS-coh, 2008**
Acknowledgement

- Some of the figures and slides are adapted from C. Fensch’s HPCA 2008 presentation

- Some of figures and diagrams are adapted from John H. Kelm’s ISCA 2009 and ISCA 2010 presentations