Thread-level Parallelism for the Masses

Kunle Olukotun
Computer Systems Lab
Stanford University
Feb 2007
The World has Changed

- Process Technology Stops Improving
  - Moore’s law but …
  - Transistors don’t get faster and they leak more (65nm vs. 45nm)
  - Wires are much worse
- Single Thread Performance Plateau
  - Design and verification complexity is overwhelming
  - Power consumption increasing dramatically
  - Instruction-level parallelism (ILP) has been mined out

**The Right Hand Turn:**
- Move away from frequency as performance
- Multi– everywhere; MT, CMP

From Intel Developer Forum, September 2004
The Era of Single-Chip Multiprocessors

- Single-chip multiprocessors provide a scalable alternative
  - Relies on scalable forms of parallelism
    - Request level parallelism
    - Data level parallelism
  - Modular design with inherent fault-tolerance and match to VLSI technology
- Single-chip multiprocessors systems are here
  - All processor vendors are following this approach
  - In embedded, server, and even desktop systems
- How do we architect CMPs to best exploit thread-level parallelism?
  - Server applications: throughput
  - General purpose and scientific applications: latency
Outline

- Motivation: The era of chip multiprocessors
- Throughput and low power: Sun Niagara
- Latency: Stanford TCC
TLP for the Masses (Google)

Source: Luiz Barroso, ACM Queue, Sept 2005
- TCO dominated by power costs
  - 4 year server life cycle @ $ 0.09 KWh
- We must improve performance/watt

Source: Luiz Barroso, ACM Queue, Sept 2005
## Commercial Server Workloads

<table>
<thead>
<tr>
<th></th>
<th>Web99</th>
<th>JBB</th>
<th>TPC-C</th>
<th>TPC-H</th>
<th>SAP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td>Web server</td>
<td>Java App. server</td>
<td>OLTP</td>
<td>DSS</td>
<td>ERP</td>
</tr>
<tr>
<td><strong>Instruction-level parallelism</strong></td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td><strong>Thread-level Parallelism</strong></td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td><strong>Working set</strong></td>
<td>large</td>
<td>large</td>
<td>large</td>
<td>large</td>
<td>large</td>
</tr>
<tr>
<td><strong>Data sharing</strong></td>
<td>low</td>
<td>med</td>
<td>large</td>
<td>med</td>
<td>large</td>
</tr>
</tbody>
</table>
Server Throughput Computing Design

- Commercial server applications
  - Lots of tasks → multiple threads
  - Low ILP and high memory stall

- Best performance (throughput) achieved with multiple threads
  - Scalable form of parallelism
  - Tradeoff latency of single thread for throughput of multiple threads
  - Forgo single wide OOO CPU for multiple simple CPUs
  - Medium to large caches
  - Lots of memory bandwidth
Maximizing CMP Throughput with Simple Cores

- J. Davis, J. Laudon, K. Olukotun PACT '05 paper
- Examined several UltraSPARC II, III, IV, and Niagara designs, accounting for differing technologies
- Constructed an area model based on this exploration
- Assumed a fixed-area large die (400 mm$^2$), and accounted for pads, pins, and routing overhead
- Looked at performance for a broad swath of scalar and in-order superscalar processor core designs
Simpler Cores Offer Higher Performance

- Optimize for Chip IPC
  - 400 mm² area architectures (4–20 cores)
  - L2 caches (1.5MB – 2.5MB)
  - Multiple pipes and multithreading is important
  - Scalar pipes are 37%–46% better than superscalar pipes (12 vs. 7 cores) [PACT 2005] for details
Processor-Cache Balance is Important

- Performance on TPC-C
- C1: 64KB L1 caches, C2: 32 KB L1 caches, both 2p4t cores
Simple cores improve perf/watt
Same performance at 20% of power
8 simple cores same power as 2 complex cores

Source: Tilak Agerwala, Micro May-June 2005
Niagara 1 Design Principles

- Afara Websystems (2000)
  - Acquired by Sun Microsystems (2002)
- Designed for throughput and low power on commercial server applications
  - Niagara 1 (UltraSPARC T1) 2005
- Many simple cores vs. few complex cores
  - Exploit Thread (request) Level Parallelism vs. ILP
  - Improves power efficiency (MIPS/watt)
  - Lower development cost, schedule risk with simple pipeline
  - Improve yield by selling non-perfect parts
- Designed for good performance with cache misses
  - Lots of memory bandwidth per chip
  - Runs real apps even better than benchmarks
  - Hide cache, branch stalls with threads vs. ILP
Niagara CMP Overview

- SPARC V9 implementation
- 8 cores x 4 threads = 32 threads
- 90GB/sec crossbar switch
- High-bandwidth 4-way shared 3MB Level-2 cache on chip
- 4 DDR2 channels
- ~300M transistors
- 378 sq. mm die
- Single issue pipeline
- 4 threads, switch every cycle
- Per thread registers, instruction buffers and store buffers
  - 20% area overhead
Niagara Memory System

- Instruction cache
  - 16kB, 4-way set associative, 32B line size
- Data cache
  - 8kB, 4-way set associative, 16B line size
  - Write-through cache, write-around on miss
- L2 cache
  - 3 MB, 12-way set associative, 64B line size
  - Write-through
  - 4-way banked by line
- Coherency
  - Data cache lines have 2 state: valid or invalid
  - Data cache is write through  no modified state
  - Caches kept coherent by tracking lines in directories in L2
- Consistency
  - TSO provided by crossbar
  - L2 is consistency point  threads that share L1 cache must wait to see to stores
Memory System Performance

- **Latencies**
  - Load-to-use: 3 cycles
  - Unloaded L2 latency: 22 cycles
  - Unloaded memory latency: ~90 ns

- **Bandwidth**
  - L2 bandwidth: 76.8 GB/s
  - Memory bandwidth: 25.6 GB/s
TPC-C Multithreading Performance

- 3x increase in throughput for 20% area increase
- 33% increase in latency

Core CPI

<table>
<thead>
<tr>
<th>1 Thread Per Core</th>
<th>4 Threads Per Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.39</td>
<td>1.8</td>
</tr>
</tbody>
</table>
Niagara 1 Die Photo

- 1 FPU
- 90 nm
- 279M transistors
- 378 mm$^2$
## 90nm Microprocessor Comparison

<table>
<thead>
<tr>
<th>Processor</th>
<th>Niagara</th>
<th>Opteron</th>
<th>Pentium D</th>
<th>Power 5+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores/chip x threads/core</td>
<td>8 x 4</td>
<td>2 x 1</td>
<td>2 x 2</td>
<td>2 x 2</td>
</tr>
<tr>
<td>Peak IPC/core</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Peak IPC/chip</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>L1 I/D (KB)</td>
<td>16/8</td>
<td>64/64</td>
<td>12/16</td>
<td>64/32</td>
</tr>
<tr>
<td>L2 (MB)</td>
<td>3 shared</td>
<td>1 per core</td>
<td>1 per core</td>
<td>1.9 shared</td>
</tr>
<tr>
<td>Memory BW (GB/s)</td>
<td>25.6</td>
<td>12.8</td>
<td>8.5</td>
<td>12.8</td>
</tr>
<tr>
<td>Frequency (GHz)</td>
<td>1.2</td>
<td>2.4</td>
<td>3.2</td>
<td>2.3</td>
</tr>
<tr>
<td>Power</td>
<td>79</td>
<td>110</td>
<td>130</td>
<td>120</td>
</tr>
</tbody>
</table>
Throughput Performance

- **Throughput Performance Chart**
  - **Axes:**
    - Y-axis: Performance relative to Pentium D
    - X-axis: SPECintRate, SPECfprate, SPECJBB05, SPECWeb05, TPC-C
  - **Legend:**
    - Blue: Power5+
    - Red: Opteron
    - Green: Niagara

- **Data Points:**
  - SPECintRate: Power5+ > Opteron = Niagara
  - SPECfprate: Power5+ < Opteron < Niagara
  - SPECJBB05: Power5+ < Opteron < Niagara
  - SPECWeb05: Niagara > Power5+ > Opteron
  - TPC-C: Niagara > Power5+ = Opteron
Performance/Watt

- **Power5+**
- **Opteron**
- **Niagara**

Performance/Watt relative to Pentium D

- SPECintRate
- SPECFPRate
- SPECJBB05
- SPECWeb05
- TPC-C
Niagara 1 Bottlenecks

- Bottlenecks in pipeline and DRAM interface
- Spare bandwidth in caches
Transforming Niagara 1 into Niagara 2

- Goal: double throughput with same power
  - Double number of cores?
- Double number of threads from 4 to 8
  - Choose 2 threads out of 8 to execute each cycle
- Double execution units from 1 to 2 and add FPU
- Double set associativity of L1 instruction cache to 8-way
- Double size of fully associative DTLB from 64 to 128 entries
- Double L2 banks from 4 to 8
  - 15 percent performance loss with only 4 banks and 64 thread
Niagara 2 at a Glance

- 8 cores x 4 threads
  - 2 pipes/core
  - 2 thread groups
- 1 FPU per core
- Crypto coprocessor per core
  - DES, 3DES, AES, RC4, etc
- 4MB L2, 8-banks, 16-way S.A
- 4 x dual-channel FBDIMM ports (60+ GB/s)
  - > 2x Niagara 1 throughput and throughput/watt
- 1.4 x Niagara 1 int
- > 10x Niagara 1 FP
Niagara 2 Die

- 65 nm
- 342 mm²
The Looming Crisis

• By 2010, software developers will face…

• CPU’s with: (Niagara 2 and follow ons)
  • 20+ cores
  • 100+ hardware threads
  • Heterogeneous cores and application specific accelerators

• GPU’s with general computing capabilities

• Parallel programming gap: Yawning divide between the capabilities of today’s programmers, programming languages, models, and tools and the challenges of future parallel architectures and applications
Challenges in Parallel Programming

- Finding independent tasks
- Mapping tasks to threads
- Defining & implementing synchronization protocol
- Race conditions
- Deadlock avoidance

- Memory model
- Composing tasks
- Scalability
- Parallel performance analysis
- Recovering from errors

→ Transactions address a lot of parallel programming problems
With Christos Kozyrakis

Goal
- Make shared memory parallel programming accessible to the average developer

Shared-memory parallel programming with transactions
- No threads, no locks, just transactions...

Programmer defined transactions are the only abstraction:
- Parallel work
- Communication and synchronization
- Memory coherence and consistency
- Failure atomicity and recovery
- Performance optimization
Transactional memory definition

• Memory transaction: A sequence of memory operations that either execute completely (commit) or have no effect (abort)

• An “all or nothing” sequence of operations
  ◆ On commit, all memory operations appear to take effect as a unit (all at once)
  ◆ On abort, none of the stores appear to take effect

• Transactions run in isolation
  ◆ Effects of stores are not visible until transaction commits
  ◆ No concurrent conflicting accesses by other transactions

• Similar to database ACID properties
The basic **atomic** construct:

\[
\text{lock}(L); \ x++; \ \text{unlock}(L); \quad \Rightarrow \quad \text{atomic} \ \{x++;\}
\]

- **Declarative** – user simply specifies, system implements “under the hood”

**Basic atomic construct universally proposed**

- HPCS languages (Fortress, X10, Chapel) provide atomic in lieu of locks
- Research extensions to languages – Java, C#, Haskell, …

- **Lots of recent research activity**
  - Transactional memory language constructs
  - Compiling & optimizing atomic
  - Hardware & software implementations of transactional memory
The Atomos Transactional Programming Language

- Atomos derived from Java
  - Transactional Memory for concurrency
    - *atomic* blocks define basic nested transactions
    - Removed *synchronized* and *volatile*
  - Transaction based conditional waiting
    - Derivative of Conditional Critical Regions and Harris *retry*
    - Removed *wait, notify, and notifyAll*
    - *Watch sets* for efficient implementation
  - Open nested transactions
    - *open* blocks committing nested child transaction before parent
    - Useful for language implementation but also available for apps
  - Violation handlers
    - Handle expected violations without rolling back in all cases
    - Not part of the language, only used in language implementation
Transactional Memory Benefits

- Easier to achieve scalable results
  - Locks
    - Coarse-grained locks simple to use
      However limits scalability
    - Fine-grained locks can improve scaling
      However hard to get right
  - Transactions
    - As easy to use as coarse-grained locks
    - Scale as well as fine-grained locks
HashMap performance

Transactions scales as well as fine-grained locks
SPECjbb2000 with Nested Transactions

- 5.4x speedup with 8 processor CMP
- Violation time reduced and improved cache behavior
- Details in [ISCA’06]
Transactional coherence/consistency with non-blocking guarantees

See [ISCA’04] for details
Scaling TCC

- Eliminate commit broadcast of original TCC
- Eliminate write through cache design
- Supports parallel commits
- First scalable TM implementation of a directory based distributed shared memory that is live-lock free
Scalable TCC Performance
64 proc DSM

- Write-through commit no longer works
  - 2-phase commit ⇒ parallel commit
  - Commit is not a bottleneck
- Excellent scalability
Conclusions

- Sun Niagara
  - CMP optimized for performance/watt on commercial server applications
  - Design approach: trade latency for throughput → low power
  - Simple pipelines, CMP and multithreading, high bandwidth memory hierarchy
  - 2-4x improvement in performance/watt compared to conventional microprocessors on commercial applications
- Stanford TCC: a shared-memory parallel model with transactions
  - Software-defined transactions as the only abstraction for parallelism, communication, failure atomicity, and optimization
  - Simple path to correct code:
    - Through speculative parallelism and HW-based atomicity/ordering
  - Intuitive, feedback-driven performance tuning
    - Through continuous data tracking and logging
  - Initial results are encouraging
    - 90% of performance at 10% of programming effort