Designing Multicores for Programmability: The Bulk Multicore Architecture

Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

SNU, Sep 2011
The Multicore Era

• Goals: performance, energy-efficiency & programmability

• What is a Programmable Architecture?
  – Attains high efficiency while relieving the programmer from low-level tasks
  – Helps minimize the chance of (parallel) programming errors
General-purpose shared-memory multicore

- Novel scalable cache-coherence (signatures & chunks)
  - Relieves programmer/runtime from managing shared data
- High-performance sequential memory consistency
  - Provides a SW-friendly environment
- HW primitives for low-overhead program development & debugging
  (data-race detection, deterministic replay, address disambiguation)
  - Helps reduce the chance of parallel programming errors
  - Overhead low enough to be “on” during production runs
The Bulk Multicore

• Idea: Eliminate the commit of individual instructions at a time

• Mechanism:
  – Processors continuously commit chunks of instructions at a time (e.g. 5,000 dynamic instr)
  – Chunks execute atomically and in isolation (using buffering and undo)
  – Memory effects of chunks summarized in HW signatures
  – Chunks can be invisible to SW or generated by compiler

• Advantages over current:
  – Higher programmability
  – Higher performance
  – Simpler processor hardware
Rest of the Talk

• The Bulk Multicore
• How it improves performance
• How it improves programmability
Hardware Mechanism: Signatures  [ISCA06]

- Hardware accumulates the addresses read/written in signatures

- Read and Write signatures
- Summarize the footprint of a Chunk of code
Signature Operations In Hardware

Josep Torrellas
The BULK Multicore Architecture

Inexpensive Operations on Groups of Addresses
Executing Chunks Atomically & In Isolation: Simple!

$W_0 = \text{sig}(B, C)$
$R_0 = \text{sig}(X, Y)$

Thread 0

$W_0 = \text{sig}(B, C)$
$R_0 = \text{sig}(X, Y)$

Thread 1

$W_1 = \text{sig}(T)$
$R_1 = \text{sig}(B, C)$

$(W_0 \cap R_1) \lor (W_0 \cap W_1)$
Chunk Operation + Signatures: Bulk

- Execute each chunk **atomically** and in **isolation**
- (Distributed) arbiter ensures a **total order** of chunk commits

• Supports Sequential Consistency [Lamport79]:
  - High performance: Instructions are fully reordered by HW
    Loads and stores make it in any order to the sig
    Fences are NOOPS
  - Low HW complexity: Need not snoop ld buffer for consistency
Summary: Benefits of Bulk Multicore

• Gains in HW simplicity, performance, and programmability

• Hardware simplicity:
  – Memory consistency support moved away from core
  – Toward commodity cores
Rest of the Talk

• The Bulk Multicore
• How it improves performance
• How it improves programmability
High Performance

- HW reorders accesses heavily (intra- and inter-chunk)
- If chunks driven by compiler: Novel compiler optimizations
BulkCompiler: Compiler for Bulk Multicore [MICRO-09]

- Takes code with synchronization operations (locks, barriers..)
- Adds instructions to drive chunking
  - `beginAtomic PC`
    - Starts new chunk
    - Takes as argument the PC of the Safe-Version of code
  - `endAtomic`
    - Finishes the current chunk

 Atomicity allows compiler to:
  ➔ Optimize the code within chunks
  ➔ Ignore memory model restrictions
Example of BulkCompiler Optimization

• sum, x, y are shared variables

```c
for(...) {
    acquire
    sum += x + y;
    release
}
```

```c
beginAtomic;
for(...) {
    acquire
    sum += x + y;
    release
}
endAtomic;
```

```c
beginAtomic;
int temp = x + y;
for(...) {
    sum += temp;
}
endAtomic;
```

• HW guarantees atomic execution (no synchs needed)
• Compiler allowed to perform arbitrary optimizations inside
• If another thread accesses sum, x, y
  - HW detects failed speculation, squashes, and retries chunk

Josep Torrellas
The BULK Multicore Architecture
More Complete Transformation

- **Low-contention** critical sections:
  - Group many of them in same Atomic Region (AR)
  - Remove acquire / release
  - Insert plain spins on lock variables
    - Lock may be owned
    - Owner will squash you on release
  - Optimize and reorder the code

Josep Torrellas
The BULK Multicore Architecture
More Complete Transformation

- **Low-contention** critical sections:
  - Group many of them in same Atomic Region (AR)

```
beginAtomic
    i_1
    acquire M1
    i_2
    release M1
    i_3
    acquire M2
    i_4
    release M2
    i_5
endAtomic
```

- Remove acquire / release
- Insert *plain* spins on lock variables
  - Lock may be owned
  - Owner will squash you on release
- Optimize and reorder the code
More Complete Transformation

- **Low-contention** critical sections:
  - Group many of them in same Atomic Region (AR)

```plaintext
beginAtomic

\begin{align*}
  & i_1 \\
  & \text{acquire M1} \\
  & i_2 \\
  & \text{release M1} \\
  & i_3 \\
  & \text{acquire M2} \\
  & i_4 \\
  & \text{release M2} \\
  & i_5 \\
\end{align*}

\begin{align*}
  & \text{beginAtomic} \\
  & \text{while (M1 == taken) \{ \}} \\
  & \text{while (M2 == taken) \{ \}} \\
  & i_5 \\
  & i_2 \\
  & i_1 \\
  & i_4 \\
  & \text{endAtomic}
\end{align*}
```

- Remove acquire / release
- Insert **plain** spins on lock variables
  - Lock may be owned
  - Owner will squash you on release
- **Optimize** and reorder the code

⇒ No conventional compiler can do this
Rest of the Talk

• The Bulk Multicore
• How it improves performance
• How it improves programmability
High programmability

- Invisible to the programming model/language
- Supports Sequential Consistency (SC)
  * Software correctness tools assume SC
- Enables novel always-on debugging techniques
  * Only keep per-chunk state, not per-load/store state
    * Deterministic replay of parallel programs with no log
    * Data race detection at production-run speed
Concept: Deterministic Replay of MP Execution

- During **Execution**: HW records into a log the order of dependences between threads
- The log has captured the “interleaving” of threads
- During **Replay**: Re-run the program
  - Enforcing the dependence orders in the log
Conventional Schemes

- Potentially large logs
• During **Execution:**
  – Commit the instructions in chunks, not individually

Combined Log of all Procs:

- P1
- P2
- Pi

If we **fix** the chunk commit interleaving:

Combined Log = NIL
Data Race Detection at Production-Run Speed  [ISCA03]

Data race: Two threads access same data without synch

- If we detect communication between…
  - Ordered chunks: not a data race
  - Unordered chunks: data race

Josep Torrellas
The BULK Multicore Architecture
Extension: Signatures Visible to SW through ISA

- Enables pervasive monitoring [ISCA04]
  - Support numerous watchpoints for free

```
Thread

usr_monitor(Addr){
    ...
}
```

Josep Torrellas
The BULK Multicore Architecture
Extension: Signatures Visible to SW through ISA

- Enables pervasive monitoring [ISCA04]
  - Support numerous watchpoints for free
- Enables novel compiler optimizations [ASPLOS08]
  - Function memoization
  - Loop-invariant code motion
- Enables debugging data races & concurrency bugs [MICRO 09]

Many novel programming/compiler/tool opportunities
Summary: The Bulk Multicore

• 128+ cores/chip, coherent shared-memory (perhaps in groups)

• Simple HW with commodity cores
  – Memory consistency checks moved away from the core

• High performance shared-memory programming model
  – Execution in chunks, possibly driven by the compiler
  – Signatures for disambiguation, cache coherence, and compiler opts

• High programmability:
  – Sequential consistency
  – Sophisticated always-on development support
    • Deterministic replay of parallel programs with no log (DeLorean)
    • Data race detection for production runs (ReEnact)
    • Pervasive program monitoring (iWatcher)
    • Using signatures/ hashes to detect races (SigRace, Light64)
Acknowledgments

Key contributors:
- Luis Ceze
- Calin Cascaval
- James Tuck
- Pablo Montesinos
- Wonsun Ahn
- Milos Prvulovic
- Pin Zhou
- YY Zhou
- Jose Martinez
The Bulk Multicore Architecture for Programmability

Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu
Pervasive Monitoring: Attaching a Monitor Function to Address

- Watch memory location
- Trigger monitoring function when it is accessed

```
Watch(addr, usr_monitor){
    ...
}
```

Josep Torrellas
The BULK Multicore Architecture
Chunk Operation + Signatures: Bulk

• Execute each chunk **atomically** and in **isolation**
• (Distributed) arbiter ensures a **total order** of chunk commits

![Diagram of chunk operations and signatures]

• Supports Sequential Consistency [Lamport79]:
  – **Low hardware complexity**: Need not snoop ld buffer for consistency
  – **High performance**: Instructions are fully reordered by HW loads and stores make it in any order to the sig Fences are NOOPS
Rest of the Talk

• The Bulk Multicore
• How it improves performance
• How it improves programmability
• Extension: Signatures visible to SW through ISA
Signatures & Hashes Visible to SW through ISA

- Enables pervasive monitoring [ISCA04]
  - Support numerous watchpoints for free
- Enables novel compiler optimizations [ASPLOS08]
  - Function memoization
  - Loop-invariant code motion
- Enables debugging data races & concurrency bugs [MICRO 09]

Many novel programming/compiler/tool opportunities
Enabling Novel Compiler Optimizations

New instruction: Begin/End collecting addresses into sig

```
... 
ld r0, x
ld r1, y
st r3, z
... 
... 
```

```
bcolltect Sig

... 
```

```
ecollect Sig
```

Josep Torrellas
The BULK Multicore Architecture
Enabling Novel Compiler Optimizations

New instruction: Begin/End collecting addresses into sig

Josep Torrellas
The BULK Multicore Architecture
Instruction: Begin/End Disambiguation Against Sig

```
  bdisamb Sig
  ...
  ...
  st r4, A
  ld r5, B
  st r6, C
  ...
  ...
  edisamb Sig
```
Instruction: Begin/End Disambiguation Against Sig

```
... ... st r4, A ld r5, B st r6, C ...
... ... edisamb Sig
```

Hardware tests for conflicts with signature register

Disambiguation

Conflict?

Sig

X, Y, Z
Optimization: Function Memoization

• Goal: skip the execution of functions

```python
foo(x);
...
= y
z = ...

foo(x);
```
Example Opt: Function Memoization

- Goal: skip the execution of functions whose outputs are known

```c
foo(x);
... = y
z = ...
foo(x);
```
Example Opt: Loop-Invariant Code Motion

```c
while(...) {
    ...
    ... = <expr>
    ...
}
```

```c
reg = <expr>
while(...) {
    ...
    ... = reg
    ...
}
```
Example Opt: Loop-Invariant Code Motion

```c
while(...) {
    ...
    ... = <expr>
    ...
}
```

```
checkpoint()
reg = <expr>
while(...) {
    ...
    ... = reg
    ...
}
```

```c
if (conflict)
rollback()
<original loop>
```
Different Synchronization Ops

Lock L → Unlock L → Lock L → Unlock L

Lock L → Set F → Unlock L

Wait F → Set F

Barrier → Barrier → Barrier