Bulk Disambiguation of Speculative Threads in Multiprocessors

Luis Ceze, University of Washington

Safe MultiProcessing Architectures at the University of Washington

National Institute of Informatics, Tokyo
Who Am I?

- Joined University of Washington in Fall’07

- How I spent the second half of my 20s:
  - at UIUC as a graduate student in Computer Architecture
  - research on multiprocessor architectures and compilers
  - before UIUC: IBM Research: Blue Gene supercomputer

- Born in São Paulo, Brazil, living in the US for ~9 years
My Current Research

• Areas: Computer Architecture/PL/Systems

• Key projects:
  • making multiprocessors deterministic
  • detecting and avoiding concurrency bugs
  • understanding parallel program behavior with graphs
Who Are You? :)

---
Talks Roadmap

- September 10:
  - Intro, Speculative Execution, Bulk Disambiguation

- September 11:
  - Memory Models, Efficient Enforcement of Sequential Consistency

- September 14:
  - Determinism in Shared-Memory Multiprocessors

- September 15:
  - Concurrency Bugs, Atomicity Violations, How to Prevent Them
  - My view of what is hot
How many times have you seen this plot?

Shared Memory Multiprocessors are Ubiquitous...

• All processor manufacturers offer multi-cores
  • lower design complexity
  • better power efficiency

• Sequential programs unlikely to run much faster

→ Need to make parallel programs ubiquitous as well
Who wrote parallel programs before?

• How did you find concurrency?
• How did you express parallelism?
• How did you debug your program?
• Did it scale?

• Machine model for the lectures:
  • shared memory
  • prevalent model, arguably easier to program than message passing
Why are multiprocessors scary?

- It is hard to find concurrency in general-purpose applications.
- Synchronization is error-prone.
- Non-deterministic behavior.
- Hard to debug.
- Memory consistency models typically obscure.

But we need to make multicores work, the HW/SW industries are betting on this!

From Bob Colwell's talk at FCRC
Architecture Support is Key

- Hooks for debugging
- Fewer correctness requirements
  - speculative parallelization with hardware safety net (TLS)
  - synchronization (TM, Colorama)
- Innovations in the architecture won’t be transparent to the software
- Simple machine model
  - shared memory, sequential consistency
  - deterministic behavior
Transactional Memory/TLS

Transaction Memory (TM)

T0

.....

ld X

ld Y

.....

re-execute

Commit

.....

T1

.....

st X

.....

re-execute

Replacement for locks: no need to worry about deadlocks, mapping, ...

Thread-Level Speculation (TLS)

Sequential

A

.....

st X

.....

B

.....

ld X

.....

T0

.....

A

.....

st X

.....

B

.....

ld X

.....

re-execute

T1

.....

No need to prove parallelism. Original sequential order is a safety net.
Compilers cannot parallelize, why not?

TLS: Assume no dependences, hardware verifies

```c
for(i=0; i<n; i++) {
    X[Y[i]] = X[Z[i]]...
}
```

TLS Task A

```c
for(i=0; i<n/2; i++) {
    X[Y[i]] = X[Z[i]]...
}
```

TLS Task B

```c
for(i=n/2; i<n; i++) {
    X[Y[i]] = X[Z[i]]...
}
```
**TLS Example**

- **TLS Hardware:**
  - Tracks data accesses at run-time
  - Detects dependence violations
  - Kills and restarts tasks

---

**Sequential**

- Time
  - A
  - B

**TLS (no dep violations)**

- Time
  - A
  - B

**TLS (dep violation)**

- Time
  - A
  - B
  - X
Requirements for Speculative Execution

✓ Monitor and enforce data dependences across threads
✓ Manage buffering of speculative state

• How typically done?
  • piggyback on coherence protocol, modify L1 cache

➡ Our goal: Simplify concepts and implementation
Current Mechanisms

Address Disambiguation
• enforce data dependences

Discard Incorrect State
• squash
• invalidate

Multiple Speculative Threads per Cache

• Leverage coherence protocol
• Extend primary caches
• Shortcoming:
  • Tight coupling with critical components
Proposal: Bulk Operations

• Encode addresses accessed by thread in signatures
  
  Read [R]
  Write [W]

• Support signature operations in hardware
  • Process sets of addresses at once — in bulk

• Use signature operations as building blocks to:
  • Monitor and enforce data dependences across threads
  • Manage buffering of speculative state
Example - Bulk Address Disambiguation

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

\[ (W_0 \cap R_1) \lor (W_0 \cap W_1) \]

\[ W_1 = \text{sig}(T) \]
\[ R_1 = \text{sig}(B, C) \]
Bulk Operations Pros & Cons

✓ Conceptual and implementation simplicity

✗ Inexact operations (superset)

✓ Correctness is guaranteed

✓ Competitive performance compared to current schemes

➡ Evaluated in the context of TLS & TM
Outline

- Introduction
- Signatures and Signature Operations
- Commit Process using Signature Operations
- Evaluation
- Conclusion
Accumulating Addresses into Signatures
Signature Operations

S₁, S₂
intersection

S₁ ∩ S₂

S₁, S₂
union

S₁∪S₂

S
intersection

S₁ ∩ S₂

S
union

S = ∅?

is empty?

Address
Signature
Encode

S

Logic
Cache
set bitmask

a ∈ S

membership

S = ∅?

is empty?

a

T/F

S = ∅?

is empty?

Address
Signature
Encode

S

Logic
Cache
set bitmask

a ∈ S

membership
• Set operations map directly to signature operations

• Can choose disambiguation granularity
  • depends on the address encoded (line, word or byte)
  • reduce squashes due to false sharing

• Encoding may cause unnecessary squashes
Composed Operation: Signature Expansion

- Select lines in the cache that belong to the signature
  - used in bulk invalidations
Bulk Invalidation

- Used in the receiver cache to:
  - invalidate lines written by the committing thread (using committing thread’s signature $W_C$)
  - if thread squash, discard speculative state (using local write signature $W_R$)
Commit Process

Committing Thread (C)
- Thread C commits
  - Send out WC
  - Set WC = RC = ∅

Receiving Thread (R)
- Thread R receives WC
  - Bulk disambiguation
    - WC ∩ RC ≠ ∅ \lor WC ∩ WR ≠ ∅
  - Squash?
    - N
      - Bulk invalidation of cache using WR
        - Set WR = RR = ∅
      - Bulk invalidation of cache using WC

All Bulk-based operations
Bulk Disambiguation Module

- **# of Versions**
- **W Signature**
- **R Signature**
- **Signature Functional Units**
- **Controller**
- **Standard Interface**
- **Processor**
- **Network**
- **Cache and Cache/Coherence Controller**

Multiple Speculative Threads in Processor
Why Simpler Architecture?

• Compact representation of sets of addresses

• Well-defined operations that map directly into hardware

• No tight coupling with coherence protocol or cache implementation

See paper for more details
Forwarding Speculative Values in TLS
Evaluation

• TM
  • Modified Jikes RVM to insert transaction annotations
  • SPECjbb2000, Multithreaded Java Grande applications

• TLS
  • Binaries generated by POSH TLS compiler [PPoPP’06]
  • SPECint 2000

• Used SESC simulator [sesc.sourceforge.net]
Signature Accuracy

- 2 Kbit signature:
  - moderate compressed size (~375b)
  - few false positives (~5%)
Performance in TLS

- Bulk: Only 5% performance degradation over Eager
- Most performance loss comes from Eager → Lazy
Performance in TM

- Bulk performance degradation over Lazy negligible
You will also find on the paper...

- Transaction nesting using multiple pairs of signatures
- Overflow and context switch discussion
- In-depth characterization (including bandwidth)
- Supporting forwarding of speculative values in TLS
Other Uses for Speculative Execution

• Prefetching
• Fault tolerance
• Simplifying code generation

• …
Conclusion

• Bulk-only design of speculative multithreading
  • for TM and TLS

• Major conceptual and implementation simplification

• Competitive performance (~5% degradation)

• Next Lecture: Enforcing SC efficiently