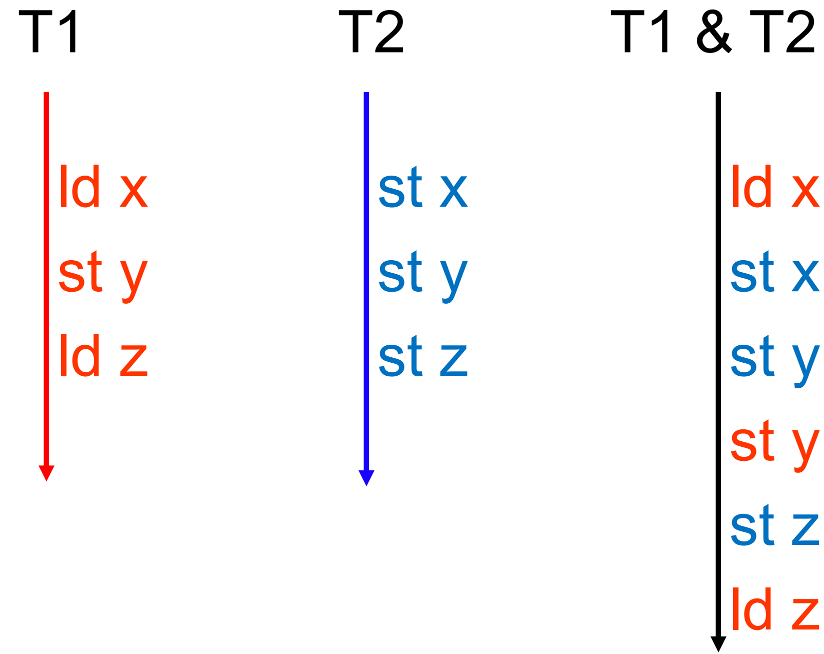
BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support

Wonsun Ahn, Shanxiang Qi, Marios Nicolaides, Josep Torrellas (UIUC)

Jae-Woo Lee, Xing Fang, Samuel Midkiff (Purdue University)

David Wong (Intel Corporation)

Sequential Consistency (SC)



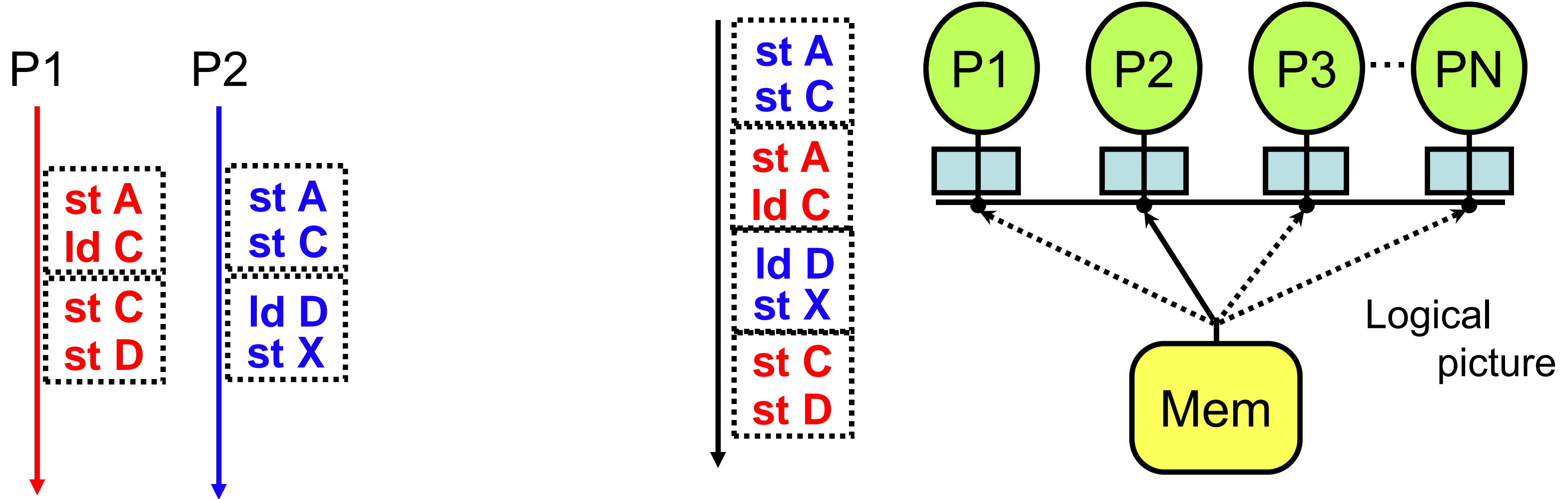
- Accesses must appear to:
  - Interleave in a single total order
  - Follow per-thread program order

Why SC for All Codes (Even Racy Ones)?

- Much easier to debug:
  - Outcomes of loads easier to reason about
  - Debuggers can reproduce buggy interleaving
- Software correctness tools assume SC:
  - More relaxed memory models result in too many interleavings
- Under SC, semantics for data races are clear:
  - Easy specifications for safe languages (such as Java)
- Some system programmers code with data races for performance
  - Less chance of making mistakes due to memory ordering

BulkSC: High-Performance SC HW [ISCA '07]

- Chunk: dynamically formed group of instructions (e.g. 2000 instrs)
- Each chunk executed atomically and in isolation
- (Distributed) arbiter ensures a total order of chunk commits



High performance SC: Within a chunk:

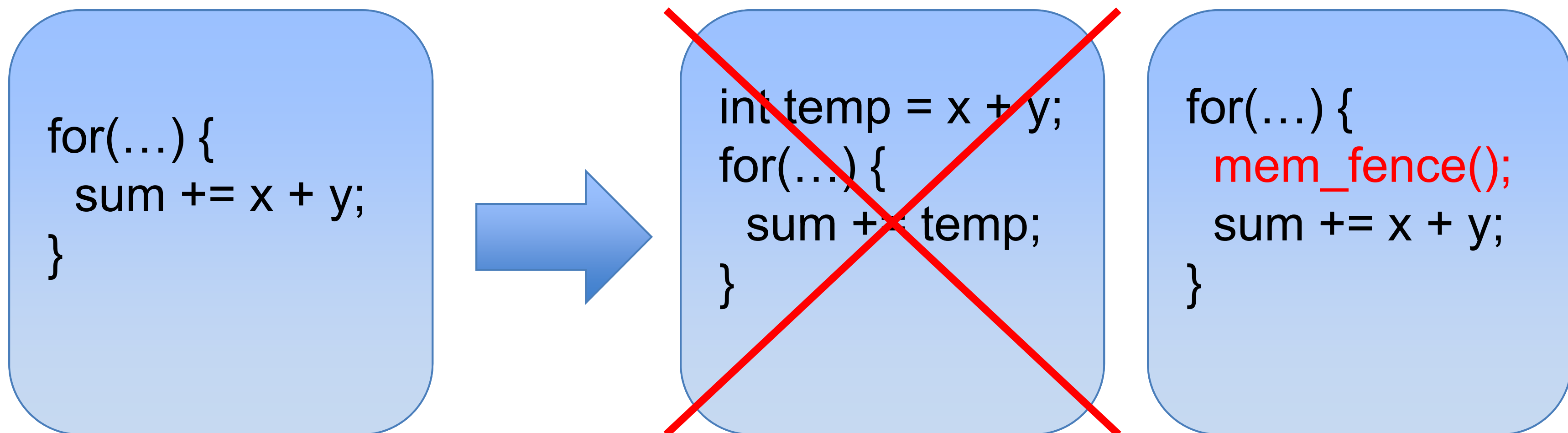
- Instructions are fully reordered by HW
- Fences are no-ops in pipeline

SC Platform: Need SC HW + SC Compiler

- BulkSC → High performance SC HW
- For an SC language memory model: also need SC compiler
- Past work on compilation for SC [PPoPP ’05]:
  - Assumed no special HW support
  - Identified un-reorderable accesses & put fences
  - Always caused slowdowns due to
    - Lost compiler optimization opportunities
    - Hardware overhead of the fences

Example Compilation for SC (Conventional)

- sum, x, y are escaping variables



- Loop invariant code motion introduces illegal ordering under SC
- May have to insert memory fences to prevent HW reordering
- Running on BulkSC:
  - Memory fences are not needed
  - But optimization is still illegal leading to lost opportunity

Contribution of this Work

- Given HW that supports SC through chunk-based execution
- Develop compiler that supports high performance SC by
  - Driving the chunk formation
  - Adapting code transformations to chunks
- Result:
  - Whole system, high-performance SC platform
  - Outperforms the relaxed Java Memory Model by avg 37%
  - Speedups come from code optimization within chunks

Rest of the Talk

- Mechanisms
- Infrastructure
- Results

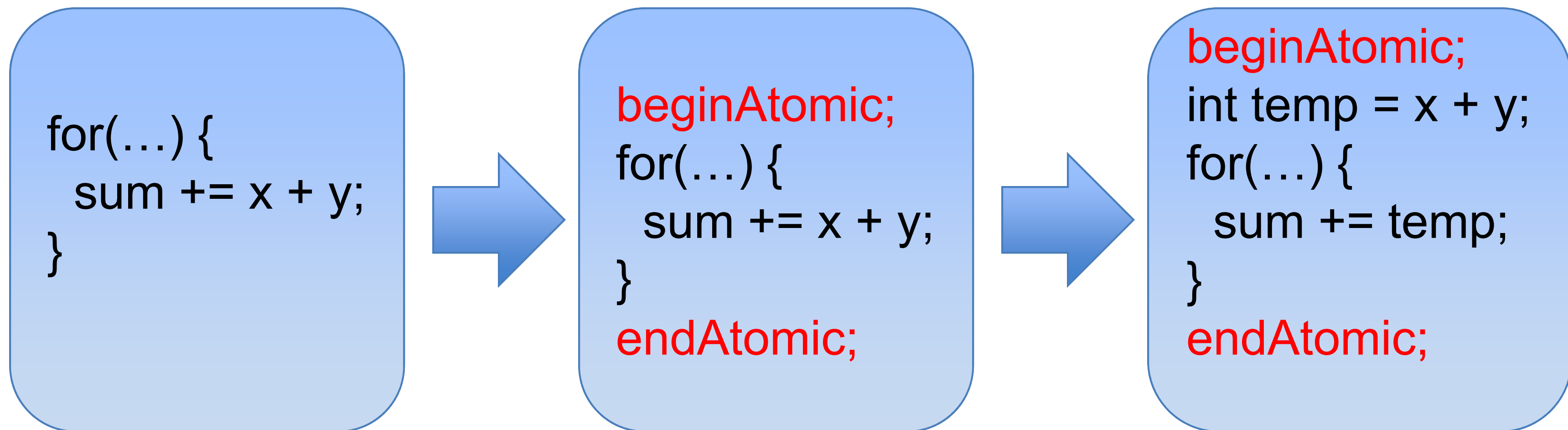
Instructions Added to Drive Chunking

- *beginAtomic PC*
  - Starts new chunk and disables automatic chunking by HW
  - Takes as argument the PC of the Safe-Version of code
- *endAtomic&Cut*
  - Finishes the current chunk
  - Re-enables automatic chunking by HW
- *endAtomic*
  - Re-enables automatic chunking by HW

➔ Atomicity allows compiler to ignore SC restrictions

Example Compilation for SC (BulkCompiler)

- sum, x, y are escaping variables



- HW guarantees atomic execution
- Compiler allowed to perform arbitrary optimizations inside
- If another thread accesses sum, x, y
  - HW detects failed speculation, squashes, and retries chunk

Transformations on Synchronization

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

beginAtomic

i_1

acquire M1

i_2

release M1

i_3

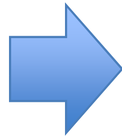
acquire M2

i_4

release M2

i_5

endAtomic



beginAtomic

i_1

while (M1 == taken) { }

i_2

i_3

while (M2 == taken) { }

i_4

i_5

endAtomic

- Remove acquire / release
- Insert plain spins on lock variables
  - Lock may be owned
  - Owner will squash on release
- Optimize and reorder the code
- Critical sections need not be nested inside atomic regions

Transformations on Synchronization

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

beginAtomic

i_1

acquire M1

i_2

release M1

i_3

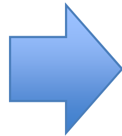
acquire M2

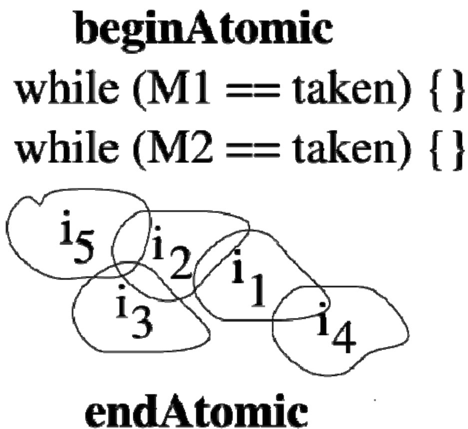
i_4

release M2

i_5

endAtomic



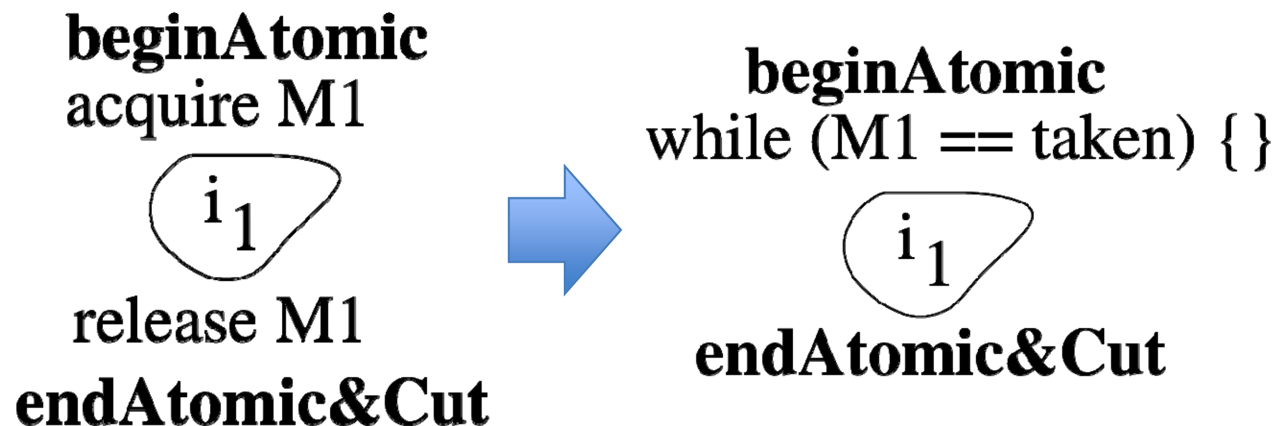


- Remove acquire / release
- Insert plain spins on lock variables
  - Lock may be owned
  - Owner will squash on release
- Optimize and reorder the code
- Critical sections need not be nested inside atomic regions

Speedups relative to relaxed memory models

Transformations on Synchronization (Cont'd)

- High-contention critical sections:
  - Tight-fit an atomic region
  - Reduce chance of squashes

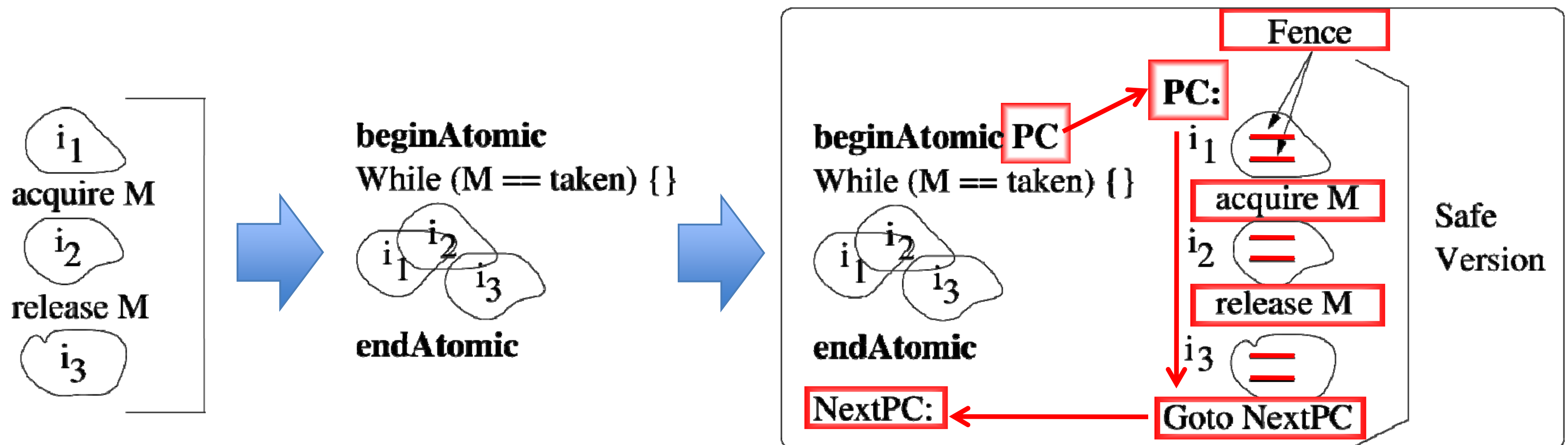


➔ Prevent slowdowns due to chunk squashes

Events Requiring Squashes of Atomic Regions

- One-time events
  - Data collisions with remote chunks
  - Interrupts
  - ➔ Just retry the AR
- Recurring events (even after multiple retries)
  - Pathologically repeating data collisions
  - Cache overflows
  - Exceptions, page faults, system calls
  - ➔ Jump to the Safe Version of the AR

Adding Safe Versions to Atomic Regions



- Create for each AR and pass PC as argument to *beginAtomic*
- Semantically a duplicate of the AR. But..
  - Does not use ARs to ensure SC (instead use fences)
  - Restricts compiler optimizations to SC only

➔ HW guarantees forward progress in Safe Versions

BulkCompiler Algorithm

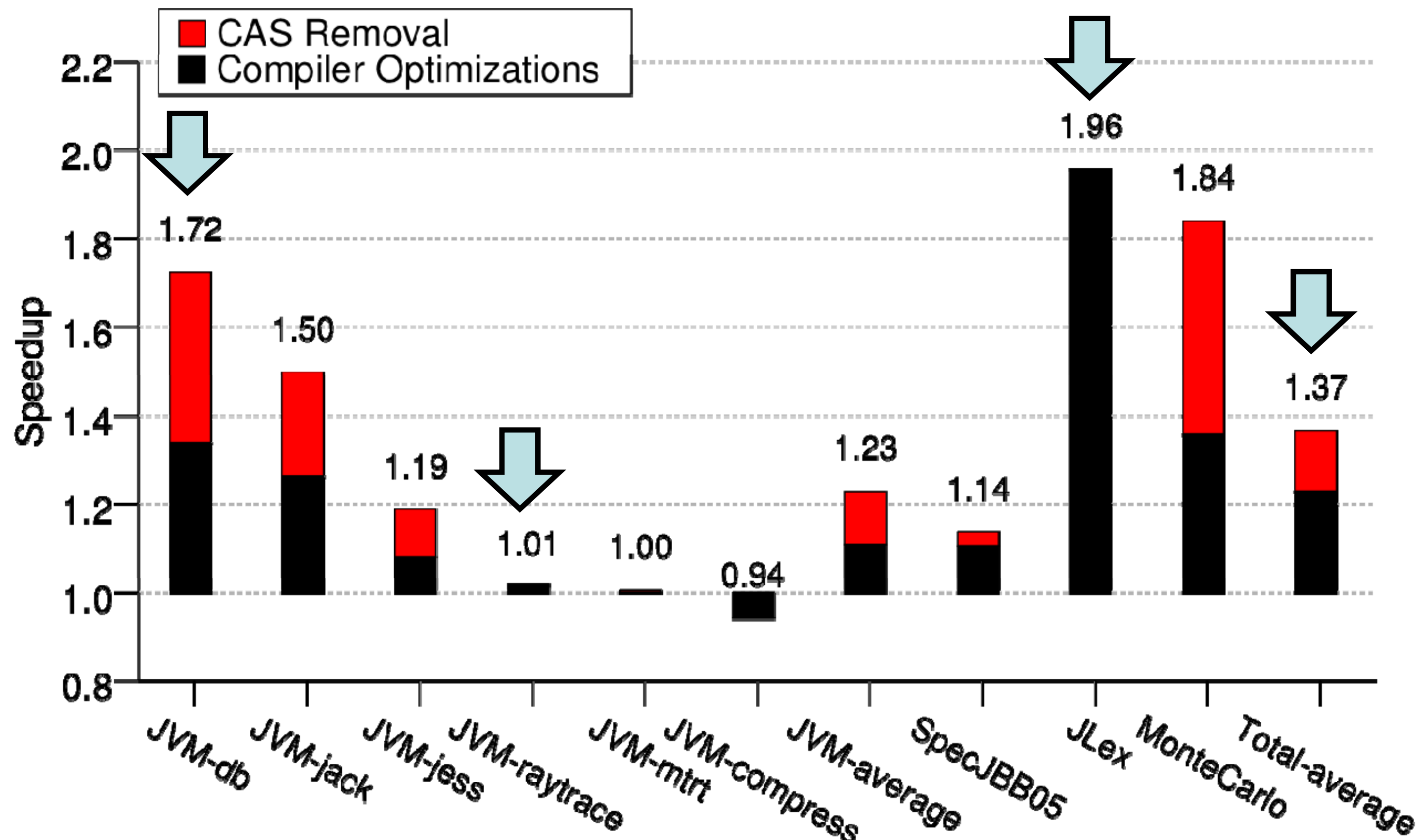
- Used Java Hotspot server compiler [JVM '01]
  - State-of-the-art commercial JVM
- Steps:

1. Perform escape analysis and mark escaping references.
2. Enclose each escaping reference in an atomic region.
3. Expand each atomic region, merging when two meet.
   - Done to maximize compiler optimizations.
4. Generate Safe Versions for all the atomic regions.
5. Replace synch (CAS) operations with plain accesses.

Evaluation

- Compare two environments:
  - Baseline:
    - Unmodified Hotspot JVM
    - Conventional multicore (4 cores @ 4GHz w/ Simics)
    - Provides Java Memory Model to programmer
      - Java Memory Model is very relaxed (similar to RC)
  - BulkCompiler:
    - Hotspot JVM + BulkCompiler
    - BulkSC multicore (4 cores @ 4GHz w/ Simics)
    - Provides SC to programmer
- Applications:
  - SPECjbb 2005, SPECjvm 98, JLex, Montecarlo

Speedup over Baseline



- Average speedup is 37% (23% from compiler opts)
- While still supporting SC

Conclusions

- Using chunk-based SC hardware, the compiler can:
  - Provide high performance SC, improving programmability
  - Achieve avg speedup of 37% over Java Memory Model
  - Speedups come from code optimization within chunks
- Atomic region support in HW enables new compiler optimizations
- Future work:
  - Study novel compiler optimizations
  - Apply to other memory models

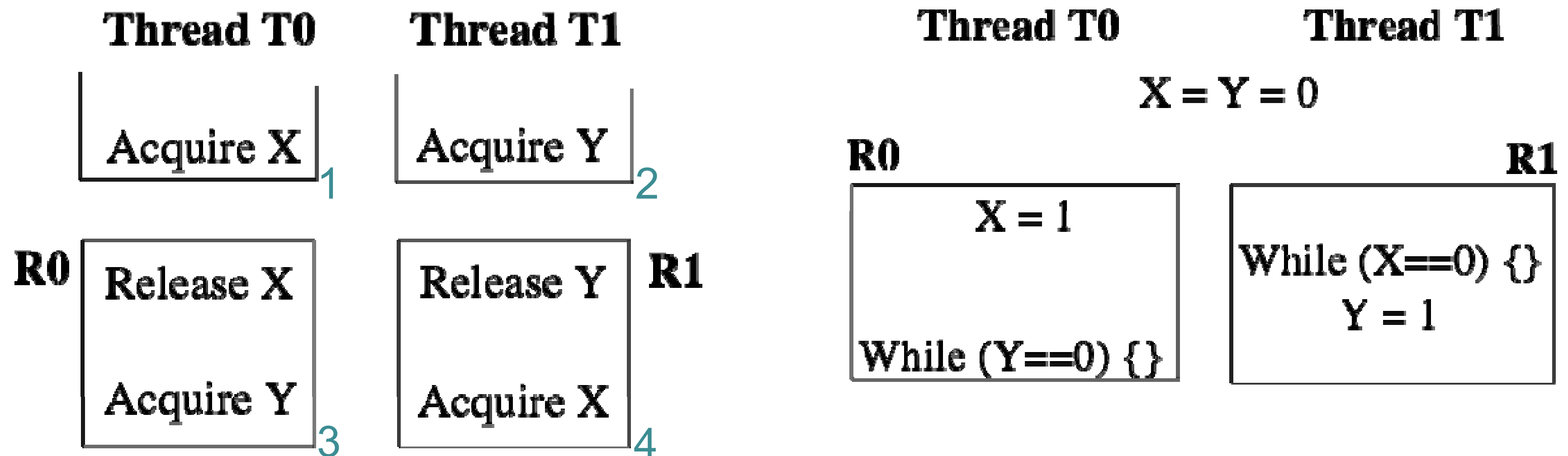
BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support

Wonsun Ahn, Shanxiang Qi, Marios Nicolaides, Josep Torrellas (UIUC)

Jae-Woo Lee, Xing Fang, Samuel Midkiff (Purdue University)

David Wong (Intel Corporation)

Tolerating Deadlocks



- HW detects deadlock using timeout mechanism
  - When instruction count of chunk becomes very large
- Squash offending chunk and jump to Safe Version

Characterization

Application	% of Dyn Instructions in ARs	# of Dynamic ARs	Dyn AR Size	# Sync Blocks per AR	Write Footprint (Lines)	Read Footprint (Lines)	% Instructions in AR Squashed
SPECJBB05	44.5	323086	19117.2	212	489.4	865.6	0.79
JVM-db	75.8	22451	119176.0	2000	84.4	3123.0	0.40
JVM-jack	29.5	2382	30105.2	792	119.7	229.4	1.31
JVM-jess	62.6	33995	43475.6	102	141.1	449.7	0.27
JVM-raytrace	85.8	61419	19771.1	0	51.7	613.9	0.10
JVM-mtrt	77.5	61627	19589.0	0	305.5	1297.0	0.14
JVM-compress	92.7	1632082	5418.6	0	28.1	144.5	0.04
JLex	97.4	45846	131474.0	317	426.9	705.7	0.91
MonteCarlo	99.9	16778	82535.1	2000	11.0	13.0	0.34
Average	74.0	244407	52295.8	602	184.2	826.9	0.48

Execution in Chunks (Atomically & In Isolation)