Lessons Learned in Designing Speculative Multithreading Hardware

Josep Torrellas

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

The Challenge for Architects

- Design parallel architectures that make it easy to write parallel code
- Compilers: no parallelization unless 100% safe:
  - Hard-to-analyze access patterns
    - Subscripted array subscripts
    - Pointer accesses

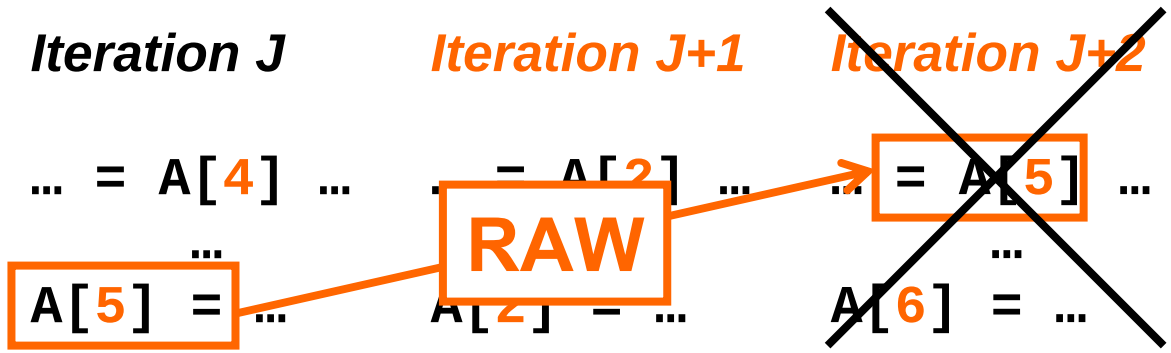
```
for(i=0;i<n;i++) {
  … = A[B[i]] …
        …
  A[C[i]] = …
}
```

Iteration J	Iteration J+1	Iteration J+2
… = A[4] …	… = A[2] …	… = A[5] …
…	…	…
A[5] = …	A[2] = …	A[6] = …

Speculative Multithreading (SM) or Thread-Level Speculation (TLS)

- Execute potentially-dependent tasks in parallel
  - Assume no dependence across tasks will be violated
  - HW tracks memory accesses; buffers unsafe state
  - Detect any violation
  - Squash offending tasks, repair polluted state, restart tasks

```
for(i=0;i<n;i++){
  … = A[B[i]] …
      …
  A[C[i]] = …
}
```



Hardware Provides Support to …

- **Checkpoint register state** at the beginning of task
- **Buffer** state being generated:
  - Speculate on code long enough that state ove cache hierarchy (or buffers)
- **Monitor** communication across tasks to enforce ordering (cache coherence protocol)
- If dependence violation: fast **undo** sid speculative task (invalidate cache, restore regs)
- If no dependence violation: task **commit**

State Buffering And Undo

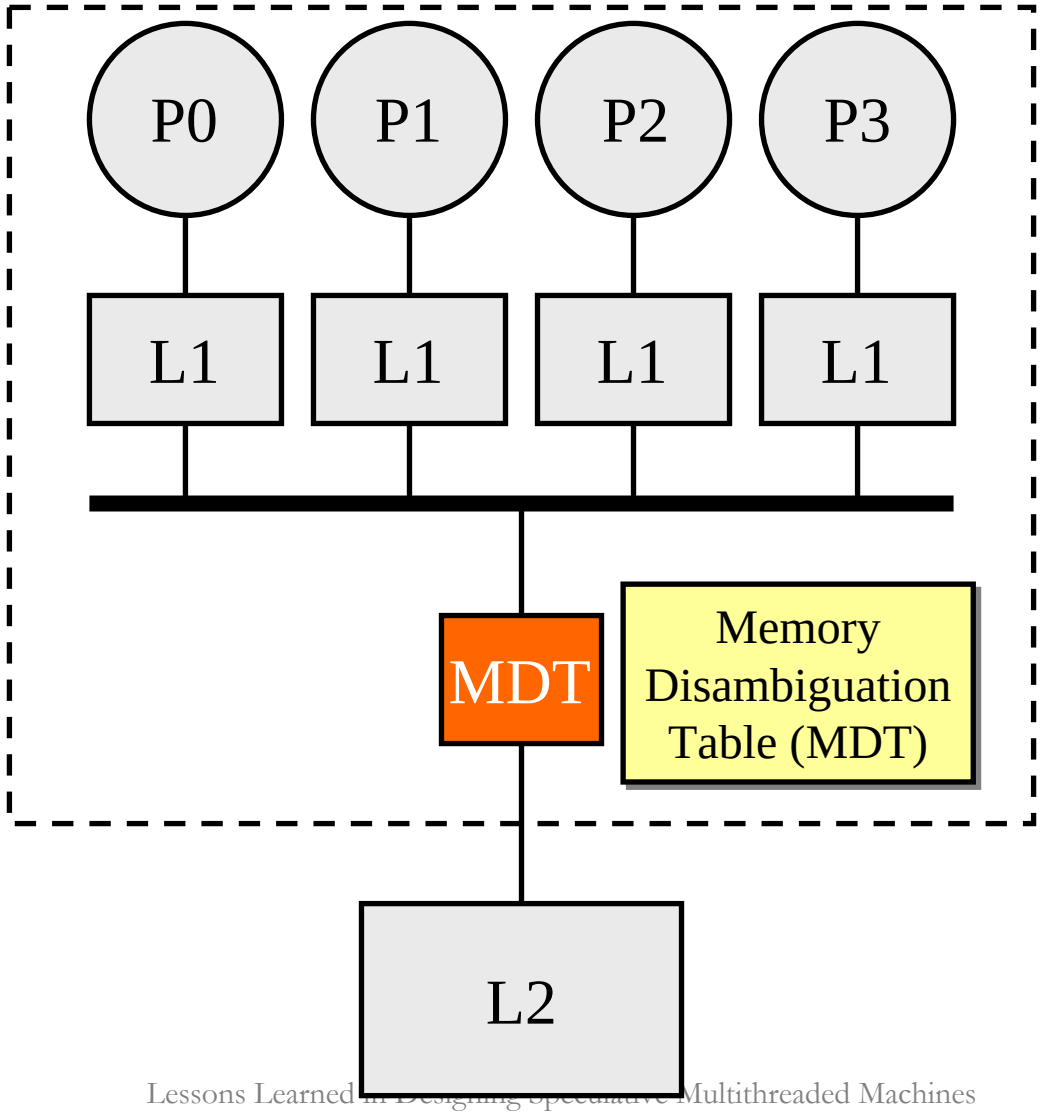
Dependence Violation Detection

Rest of the Talk: How we used TLS for 10 years for…

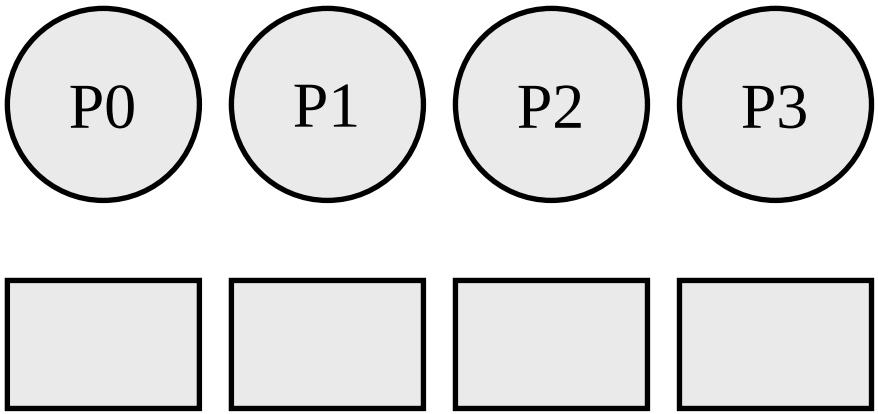
- Performance
- Software dependability
- Hardware reliability
- Performance revisited
- … and what we learned

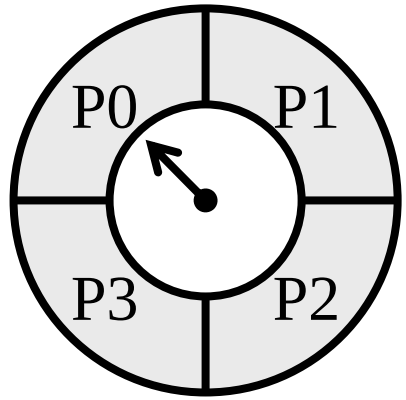
Goal 1: Performance

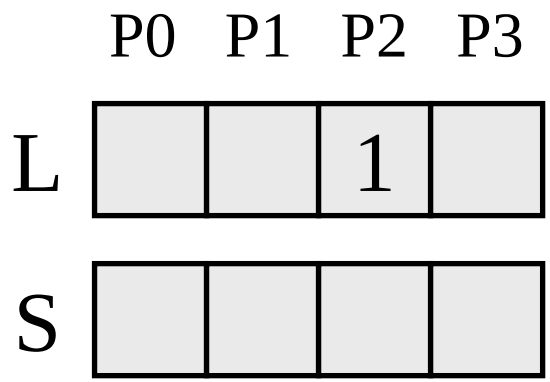
Speculative Chip-Multiprocessor (CMP) [Krishnan ICS96]



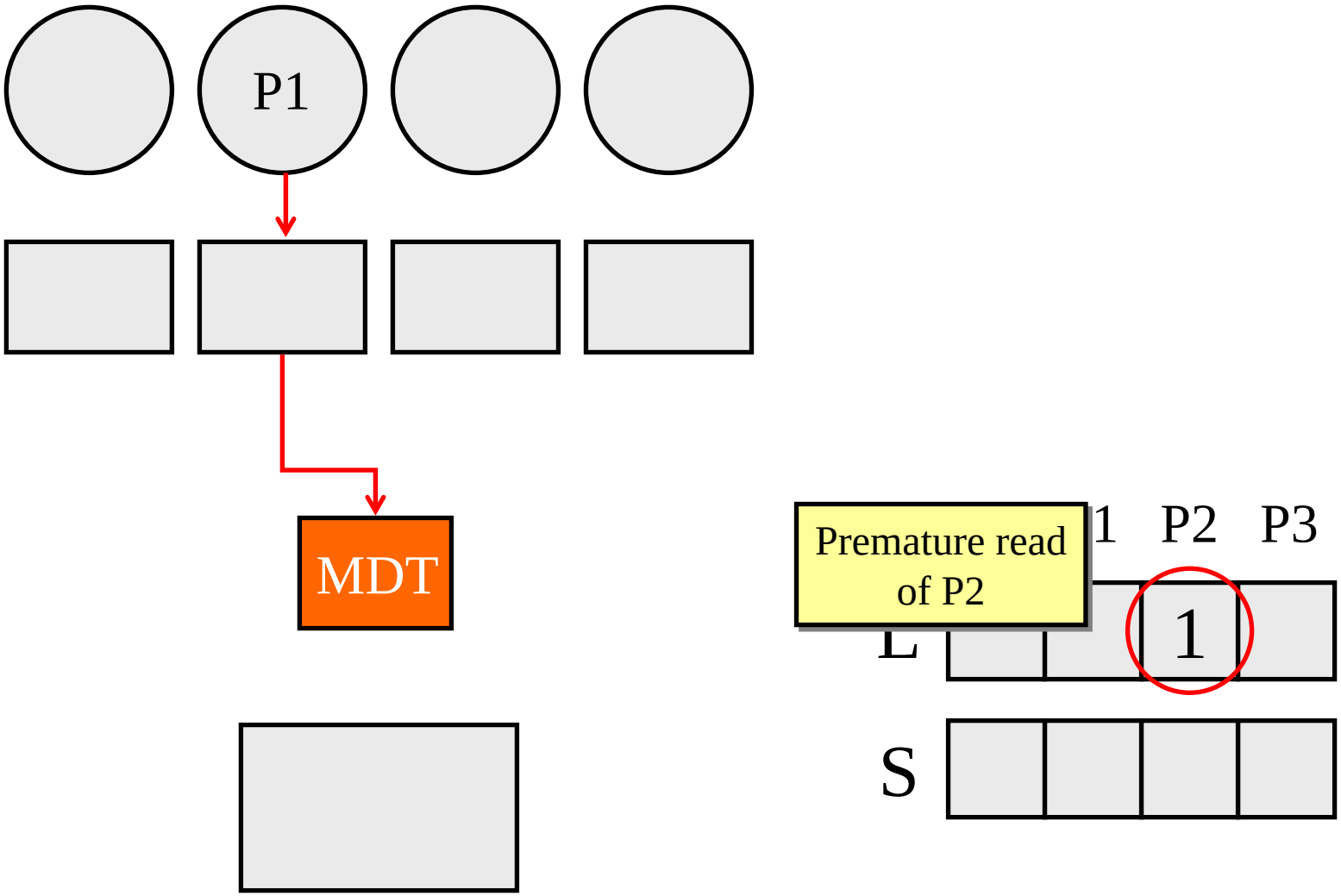
Speculative Memory Accesses



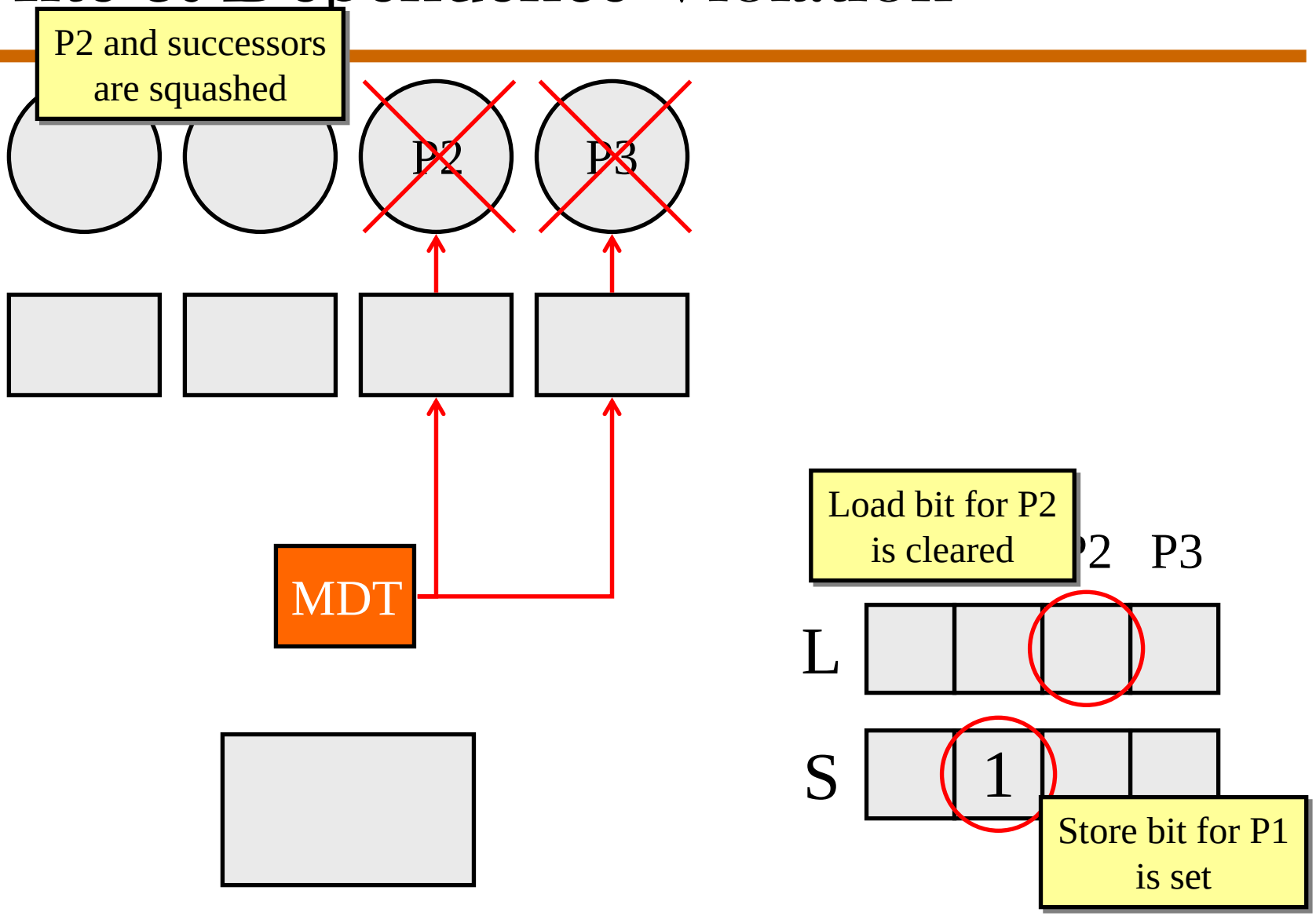




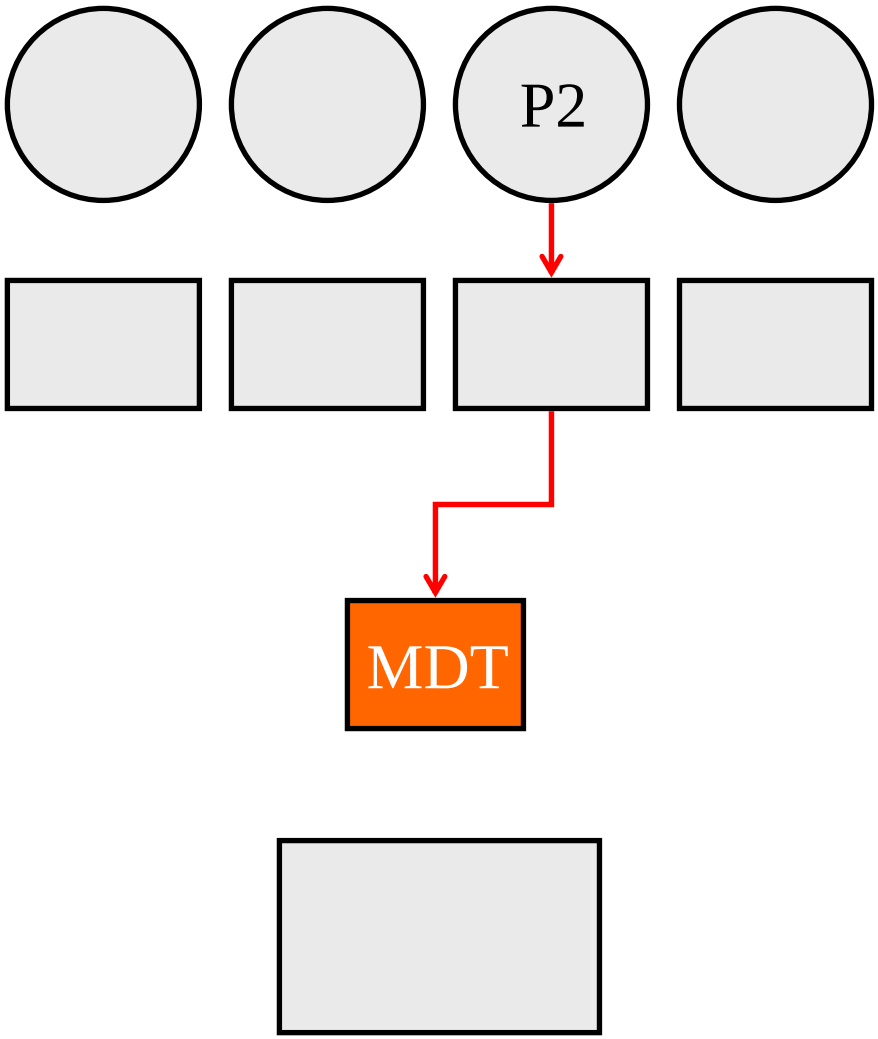
Write & Dependence Violation

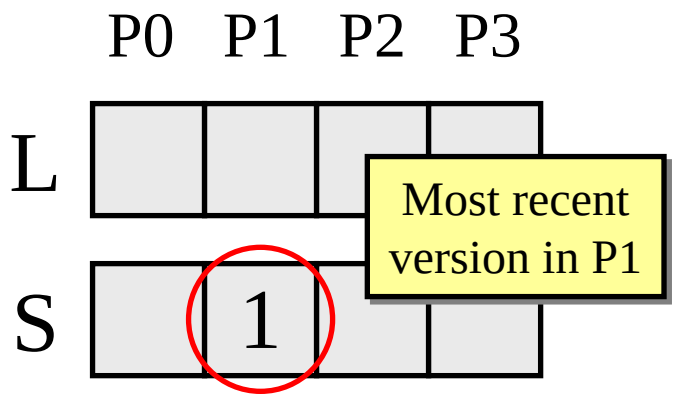


Write & Dependence Violation

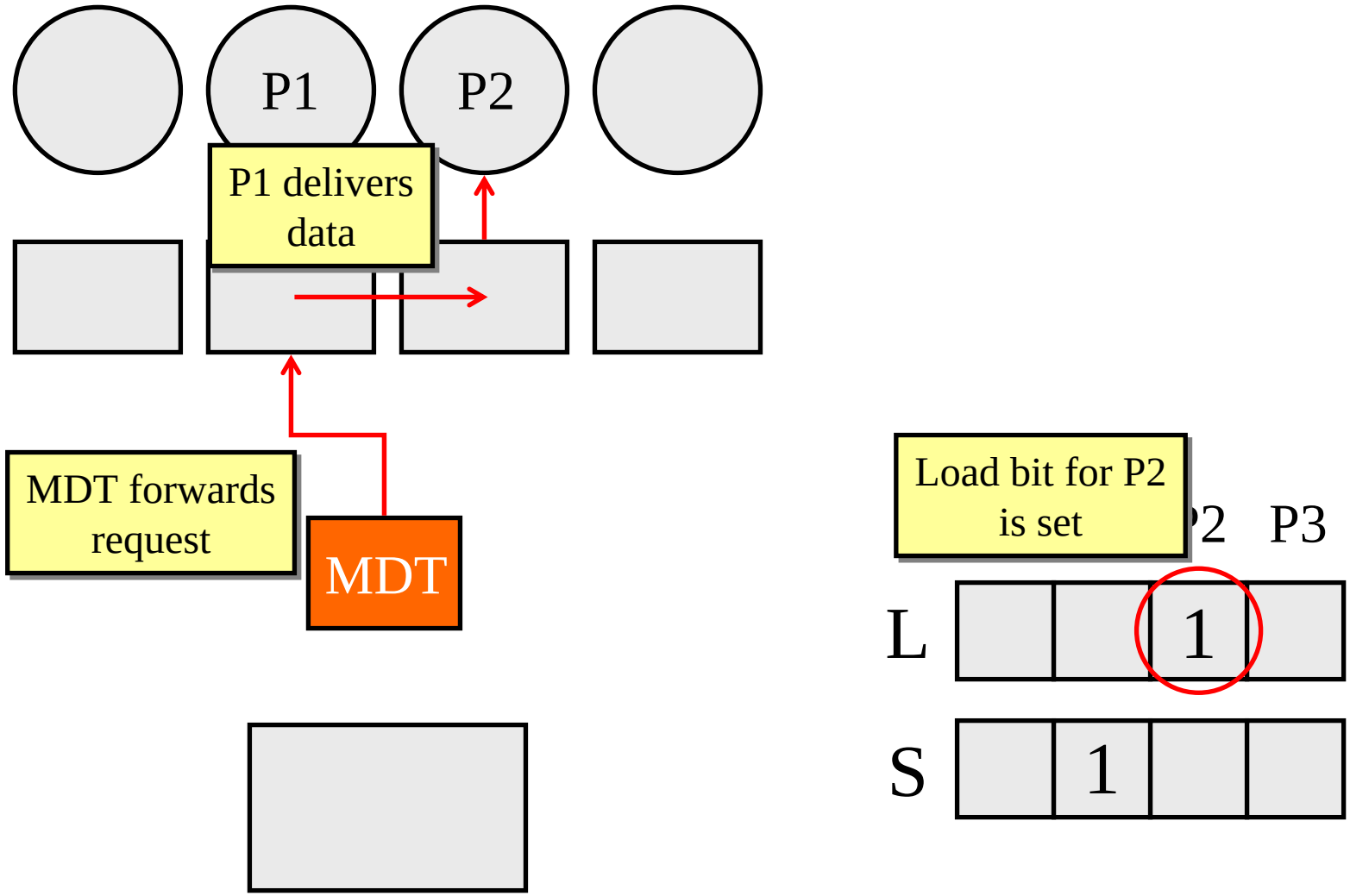


Read & Value Forwarding

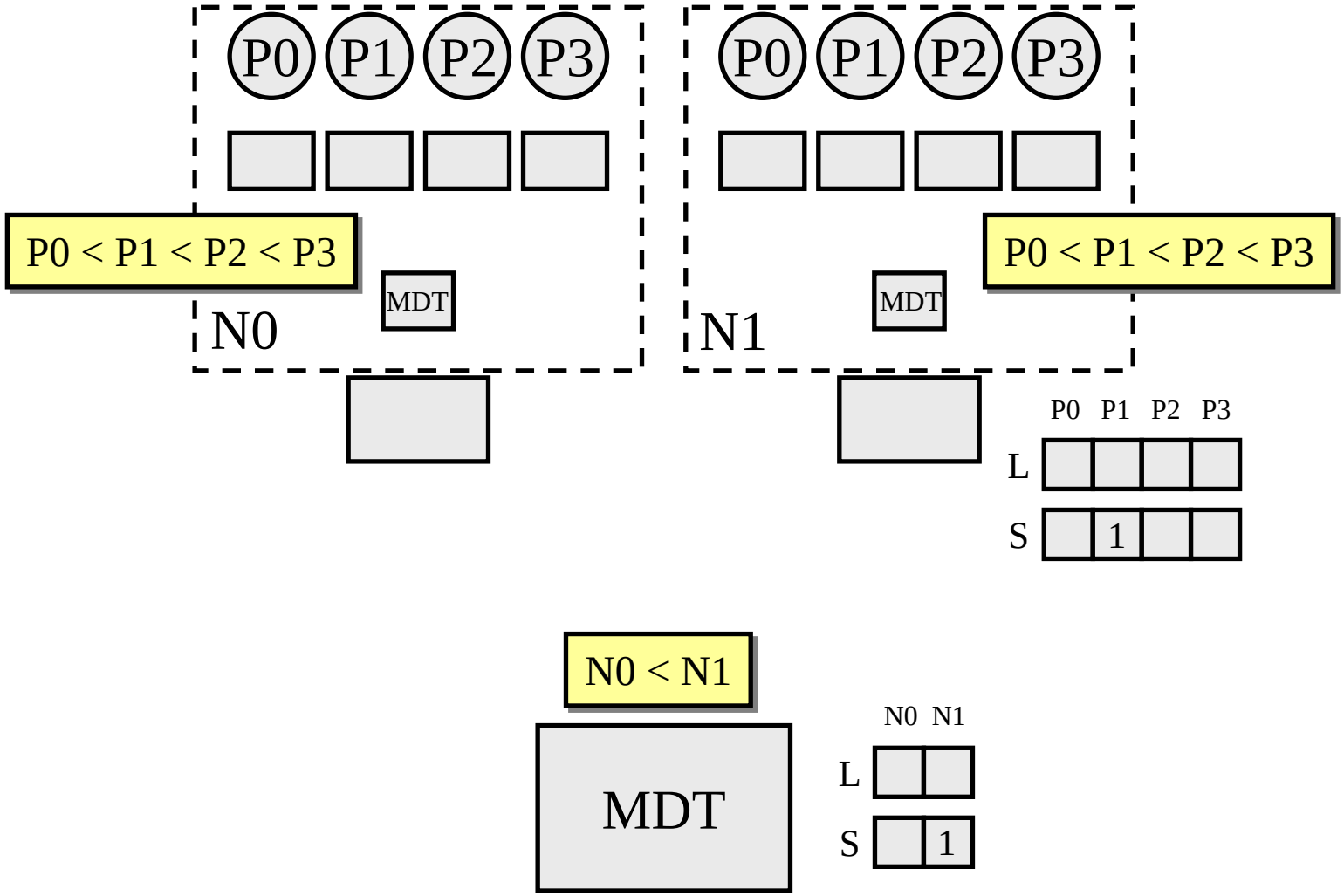




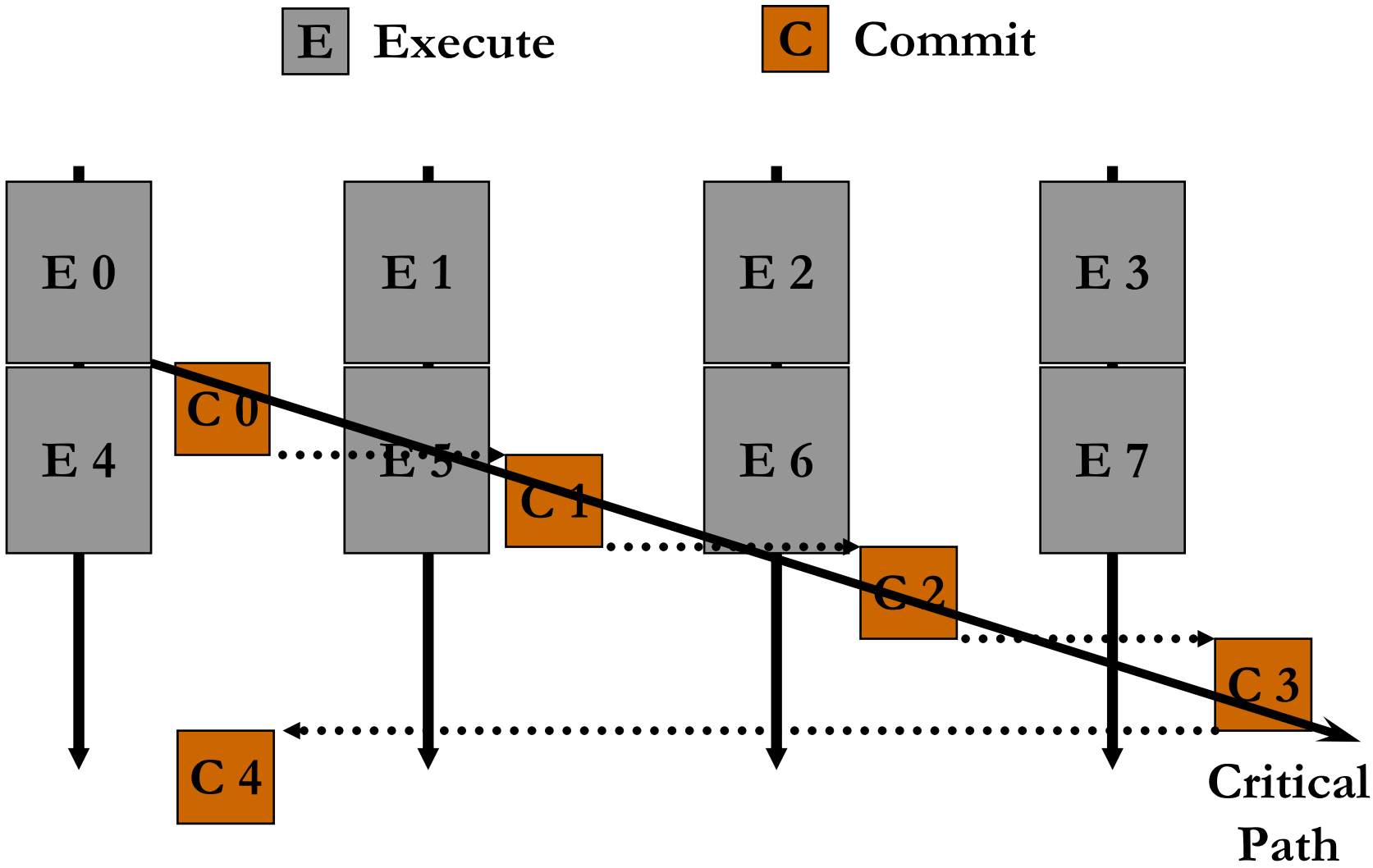
Read & Value Forwarding



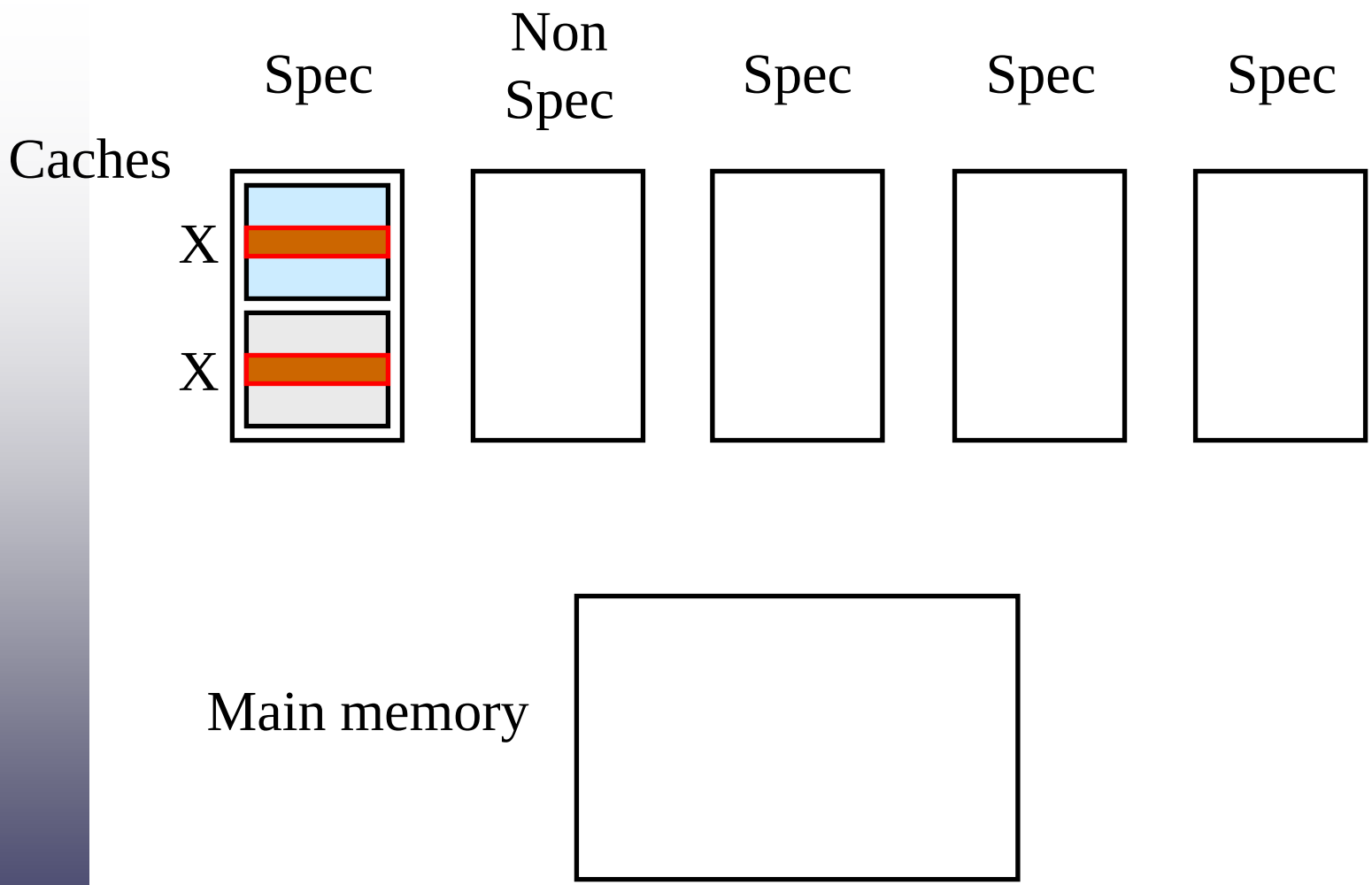
Scalable Multiprocessor [Cintra ISCA00]



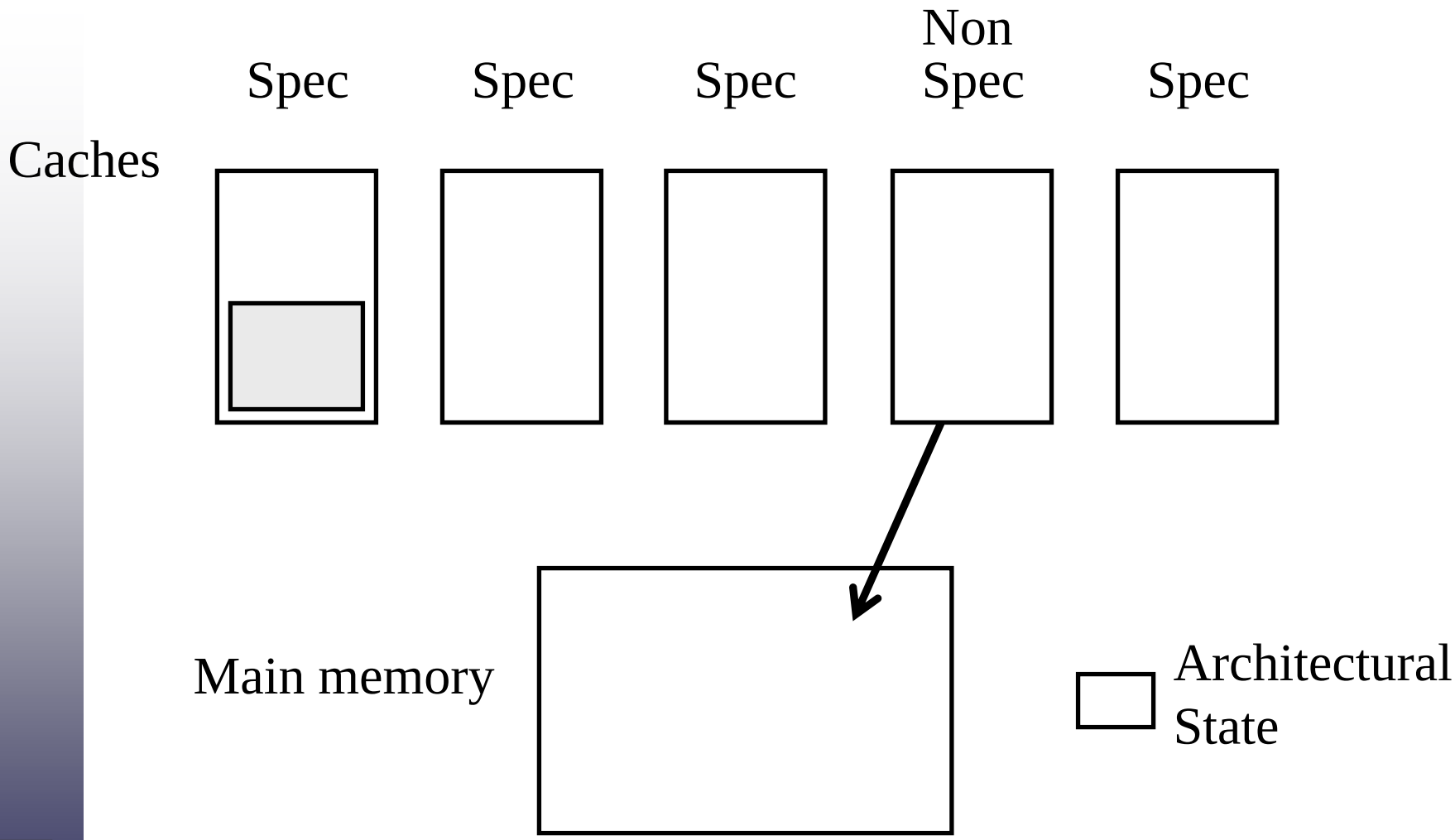
Removing Bottlenecks: Task Commit Serialization [Prvulovic ISCA01]



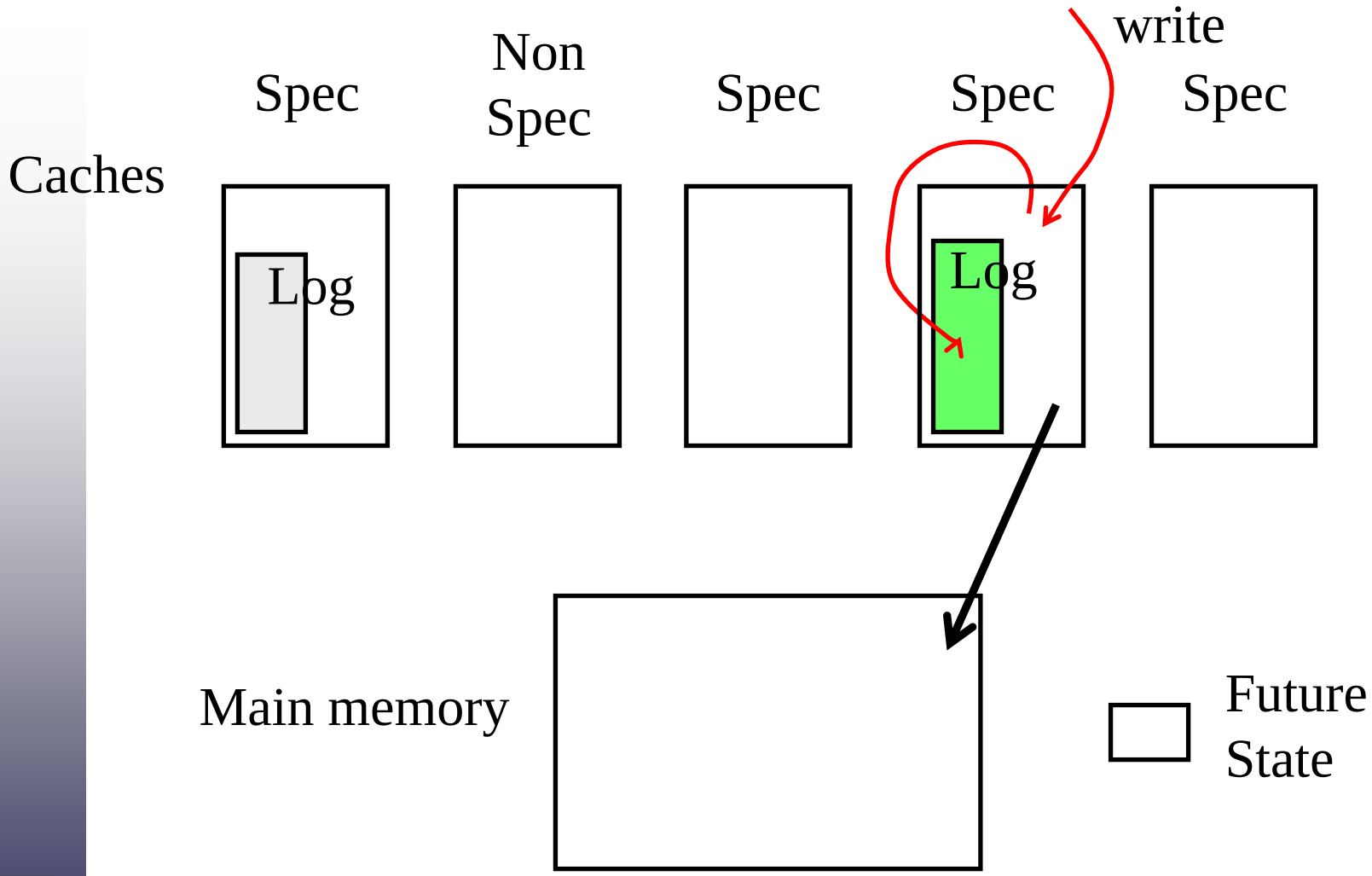
Buffering Speculative State



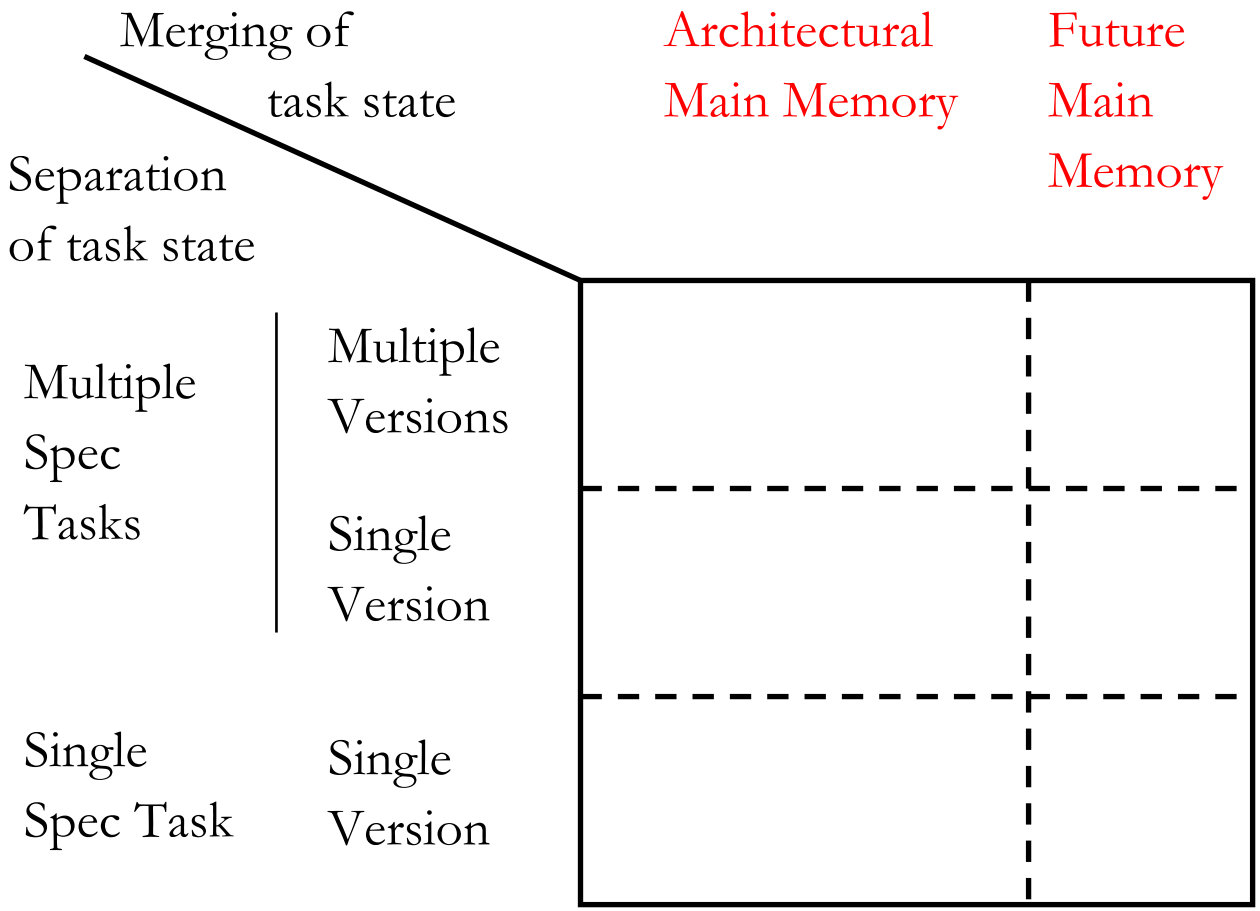
Merging of Task State: Architectural MM



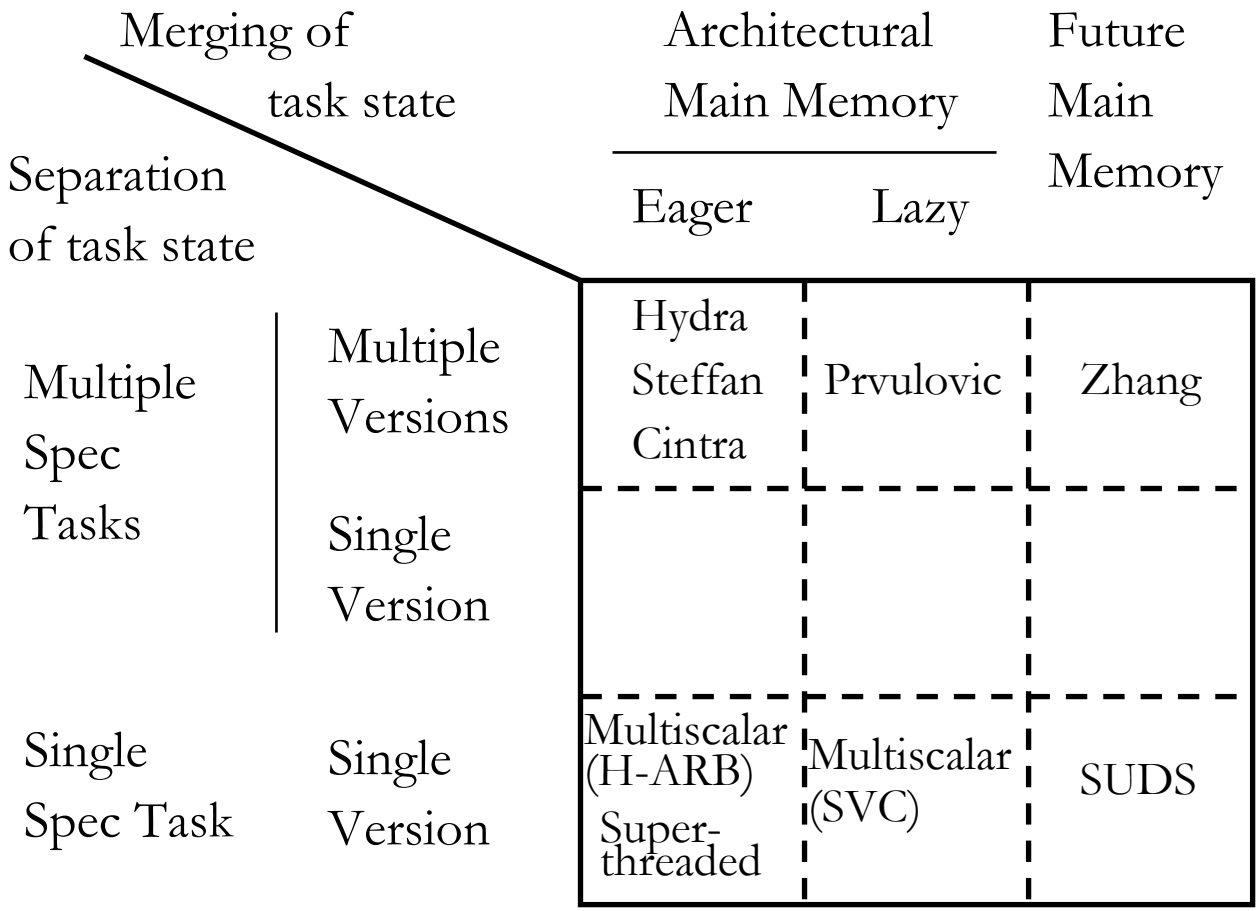
Merging of Task State: Future MM



Taxonomy of Buffering Approaches [Garzaran HPCA03]

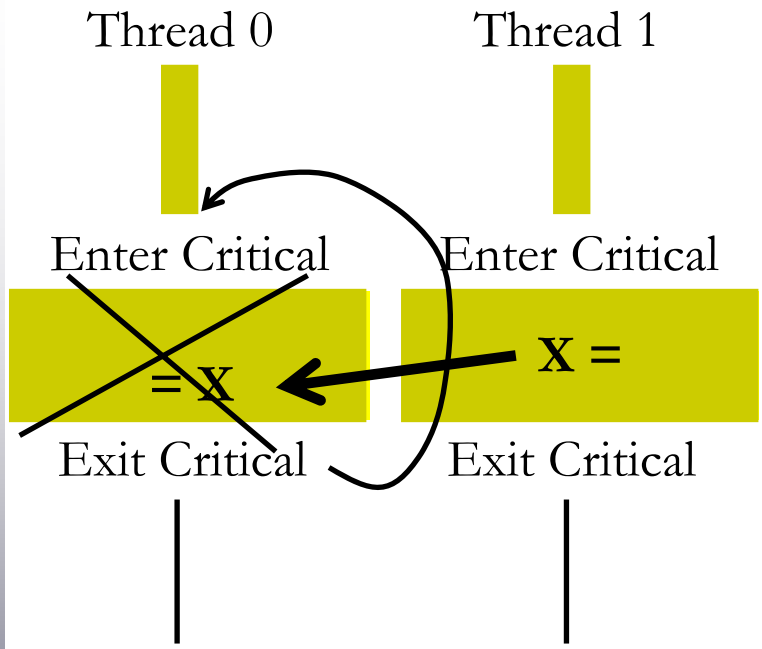


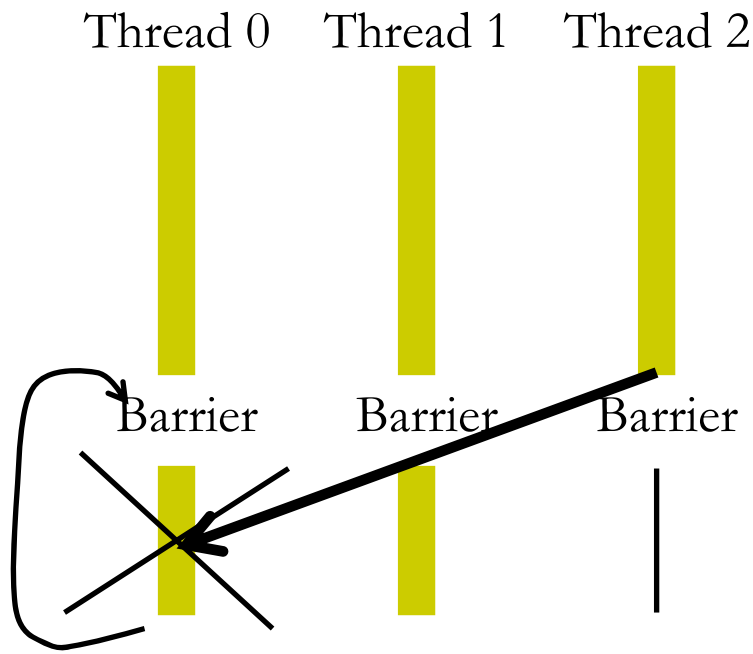
Taxonomy of Buffering Approaches [Garzaran HPCA03]



Goal 2: Performance in Other Environments

Using TLS on Explicitly Parallel Codes





- Advantages
  - Faster parallel execution
  - More Programmable: OK to write coarse critical sections or put additional barriers

Speculative Synchronization [Martinez ASPLOS02]

- Write code without fine-tuning synchronization….
  - Coarse critical sections
  - More barriers than potentially necessary

  … And still attain high performance

- Threads do not stall on Taken locks or Raised barriers
- They simply continue, executing speculative tasks
- Maintain 1 or more *safe threads* → forward progress
  - Lock: owner
  - Flag: producer
  - Barrier: lagging threads

Checkpointed Early Resource Recycling (Cherry) [Martinez MICRO02]

- Problem: Limited processor resources (registers, LD/ST queue entries..)
- Opportunity: Resources reserved until instruc retirement
  - registers
  - LD/ST queue entries
- Solution: recycle before retirement

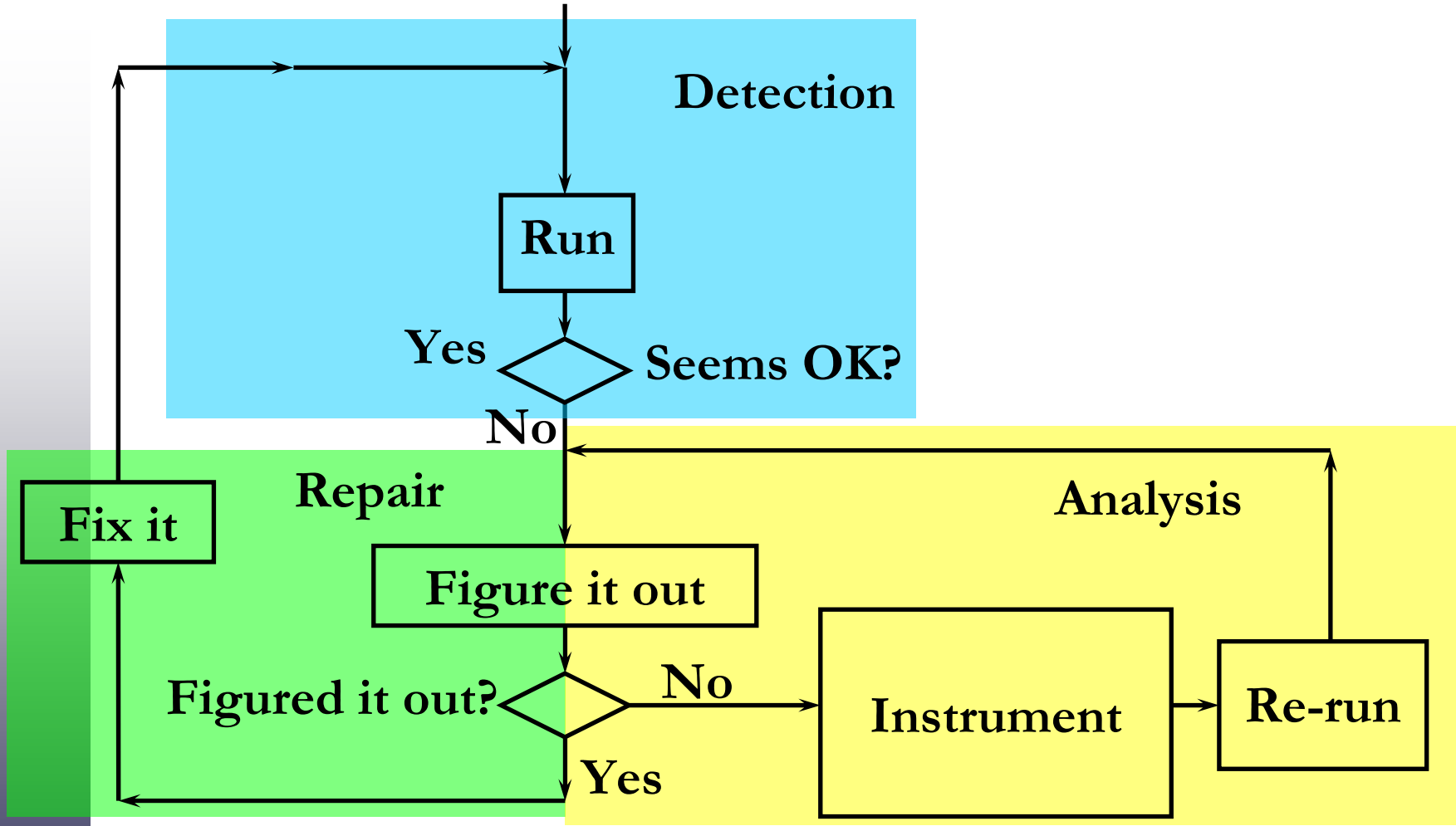
Result: higher ILP with same resources

Reuse Undo Support from TLS

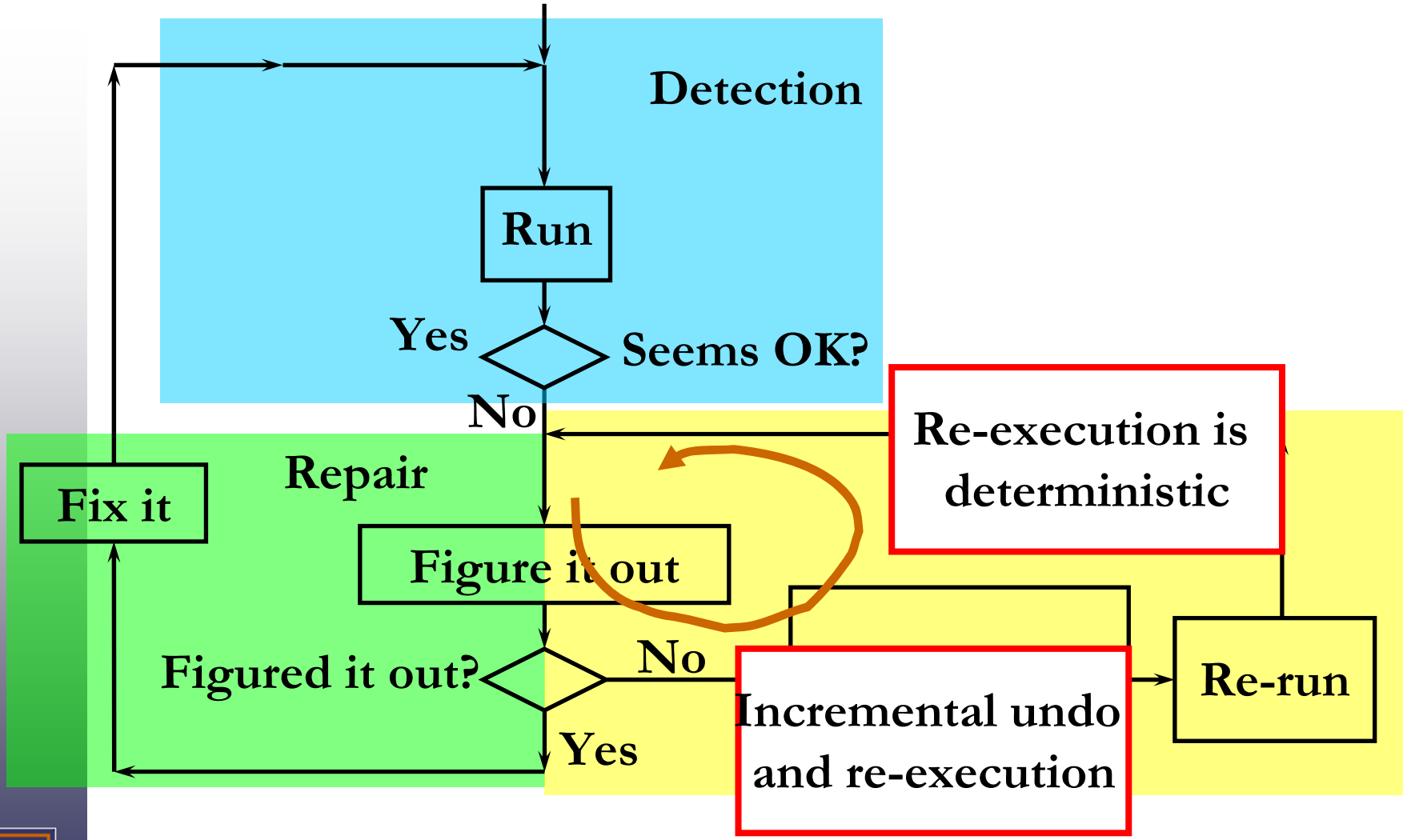
- Take register checkpoint before starting to recycle early
- Recycle a store queue entry:
  - Update is sent to the cache speculatively
- If exception on instruction with recycled resources:
  - Rollback state in registers and cache

Goal 3: Software Dependability

Conventional Debugging



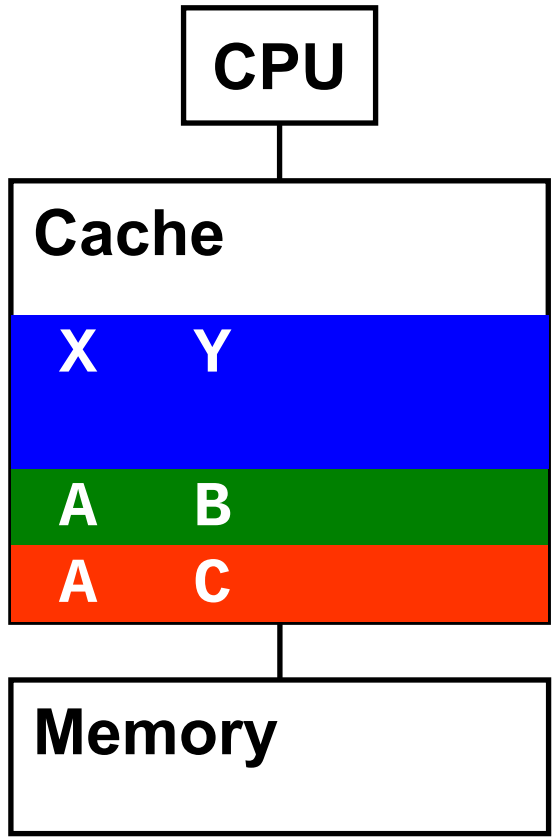
Enhancing Debugging with Speculation



ReEnact: Using TLS to Debug Data Races [Prvulovic ISCA03]

Break dynamic instructions into chunks

```
ST  X
ST  Y
ADD
 ...
ST  A
ST  B
ST  A
 ...
ST  A
ST  C
```

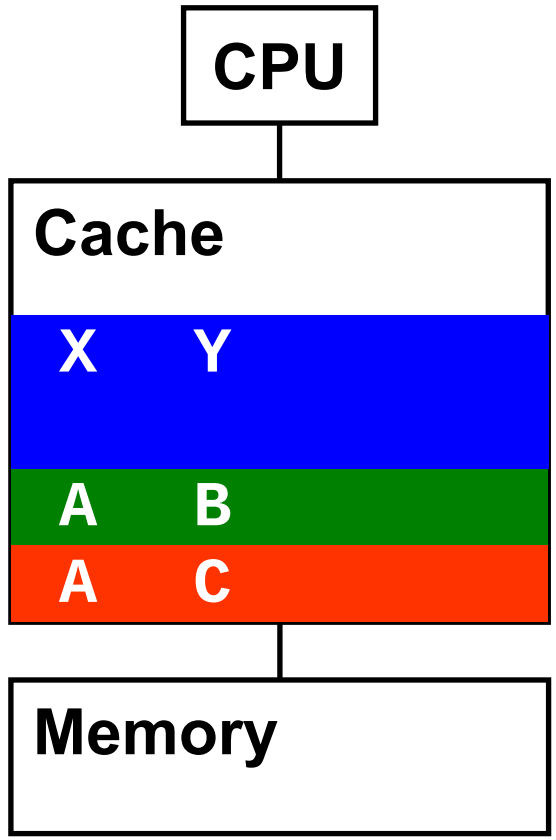


Undo Chunks

Dynamic Instructions

```
ST  X
ST  Y
ADD
 ...
ST  A
ST  B
ST  A
 ...
ST  A
ST  C
```

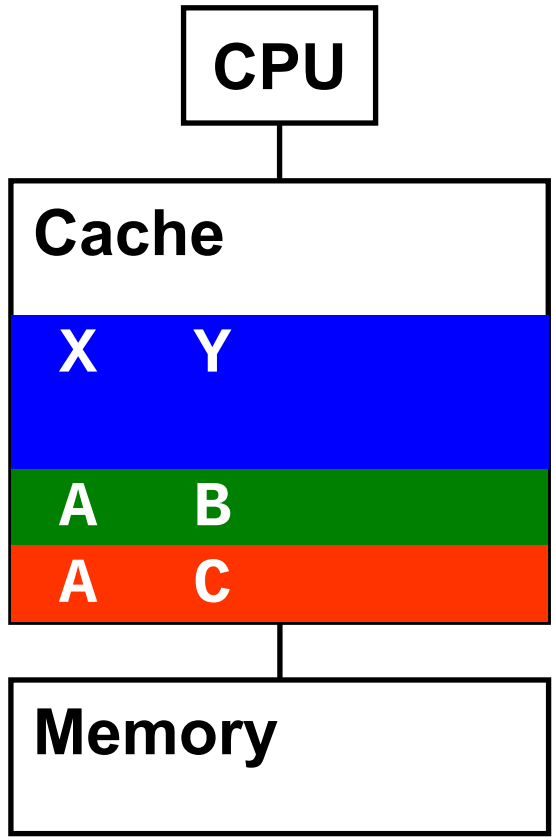
Analysis requires a rerun of these chunks



Undo Chunks

Dynamic Instructions

```
ST  X
ST  Y
ADD
 ...
ST  A
ST  B
ST  A
 ...
ST  A
ST  C
```

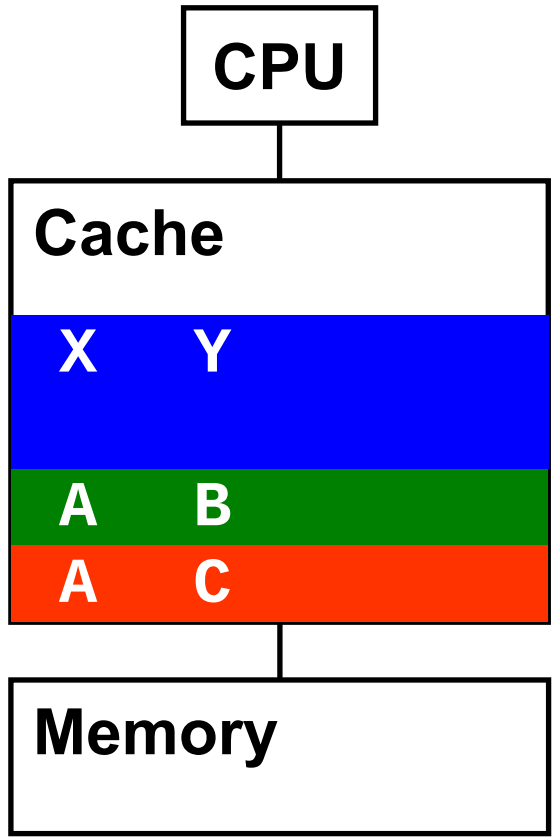


Chunk Commit

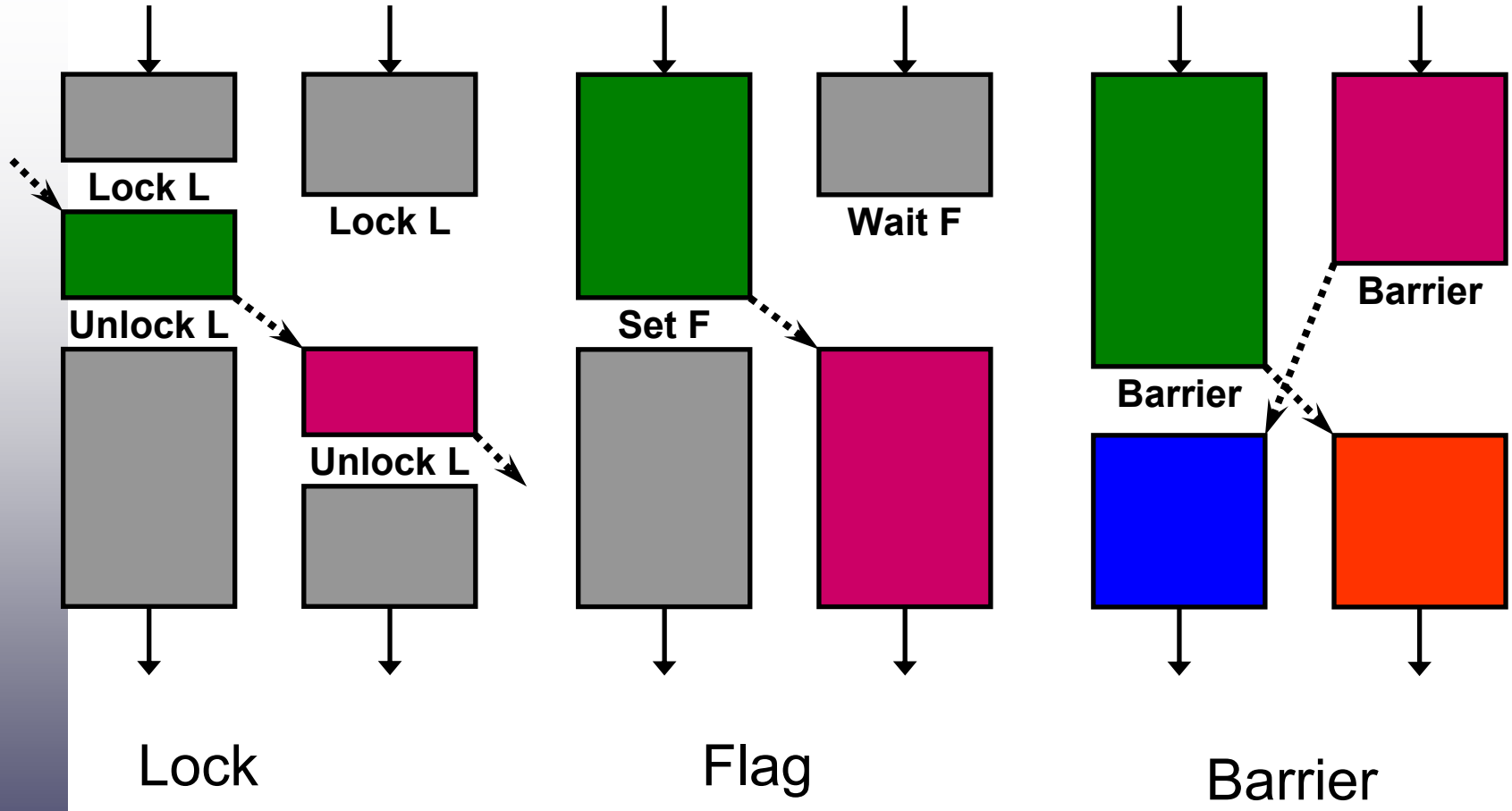
Dynamic Instructions

```
ST  X
ST  Y
ADD
 ...
ST  A
ST  B
ST  A
 ...
ST  A
ST  C
```

Need to displace X from this chunk



Chunk Ordering by Synchronization



Data Race Detection

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

Data Race Detection Example: Missing Lock

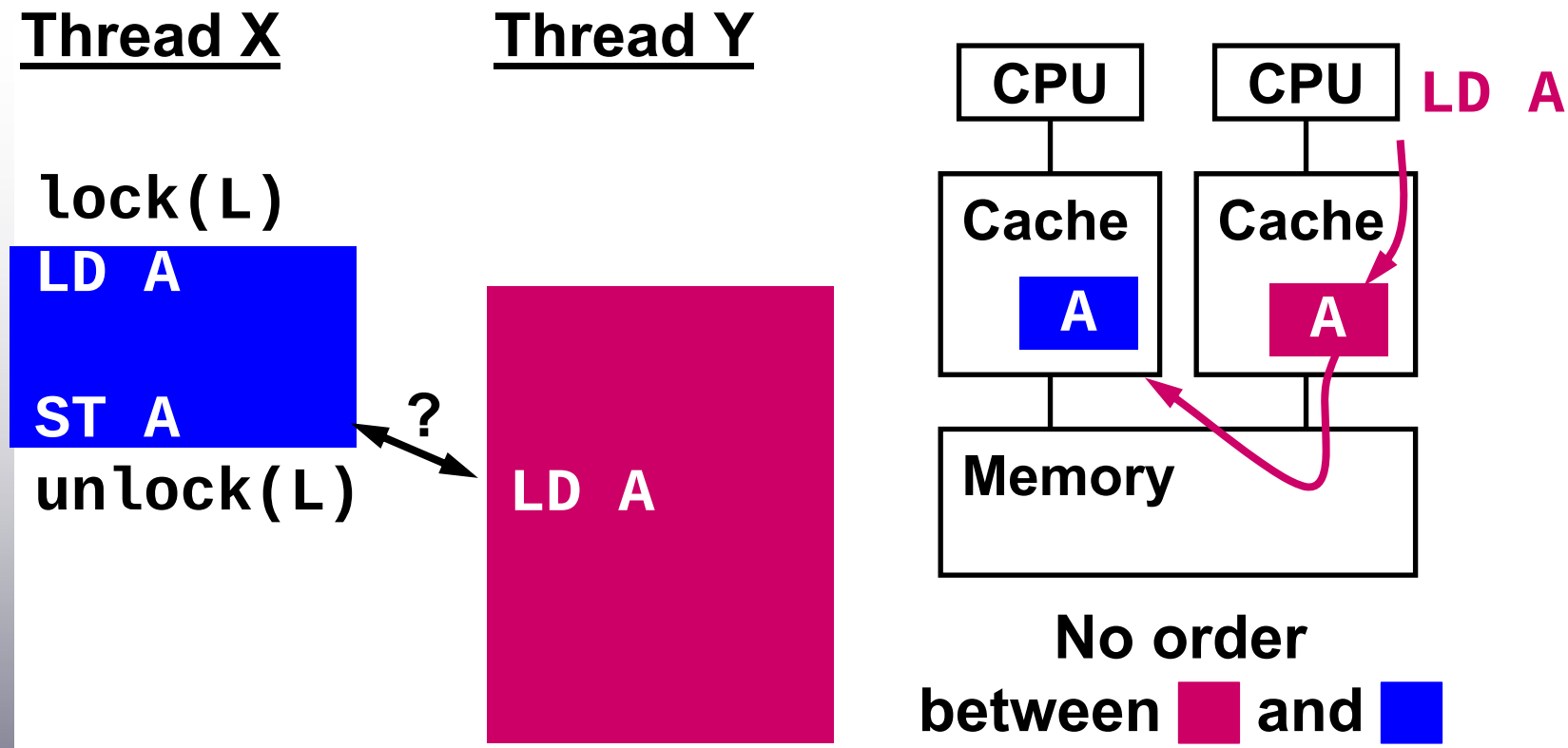
Thread X

```
lock(L)
LD A
ST A
unlock(L)
```

Thread Y

```
LD A
ST A
```

Detection: Data Race



Found Race Signature

Thread X	Thread Y
LD A	
ST A	
	LD A
	ST A

Match against a library of races

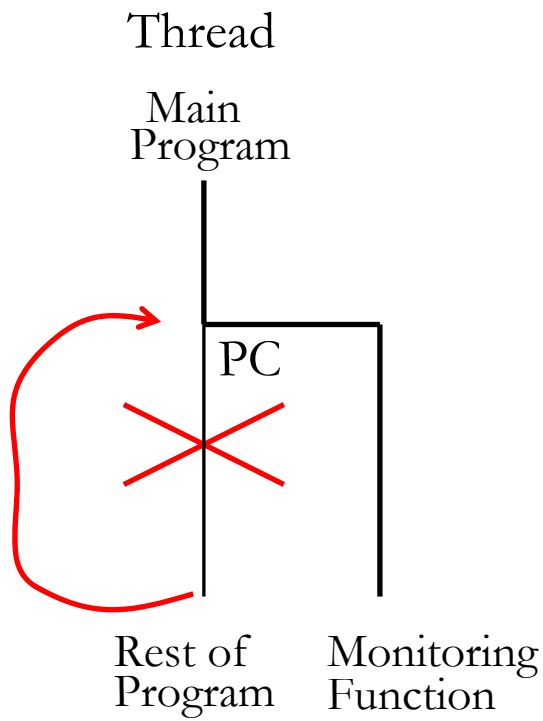
IWatcher: Attaching a Monitor Function to an Address [Zhou ISCA04]

Watch memory location and trigger monitoring function when it is accessed

```
    instr
Watch(addr, monitor_fn1)
    instr
    instr
    instr
*p = ...
    instr
    instr
    instr
```

Cache

Watched?		
1	addr	data




```
Monitor_fn1 (Addr){
  return(addr != 0)
}
```

Significance: Where is the Bug?

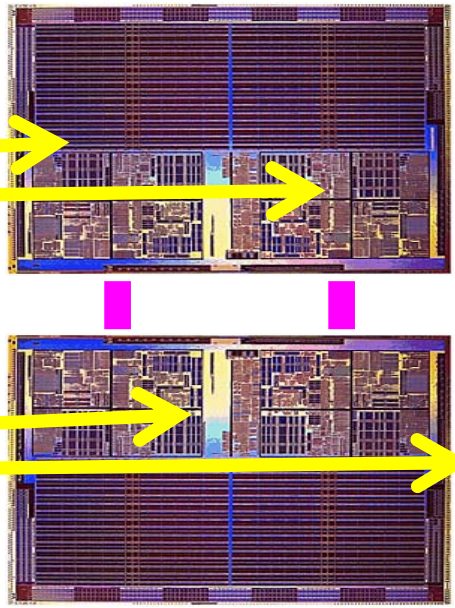
- No need to insert invariant checks
- Find it immediately

```
Watch(&x, &Monitor);
…
p = ...;   /* a bug: p points to x incorrectly */
*p = 5;   /* line A: a triggering access, bug detected with IWatcher*/
…
Assert (x==1 || x==0);    /*line B: bug detected without IWatcher */
```

Goal 3: Hardware Reliability

Paceline: Single-Thread Reliability and Speed [Greskamp PACT07]

Pair each core:

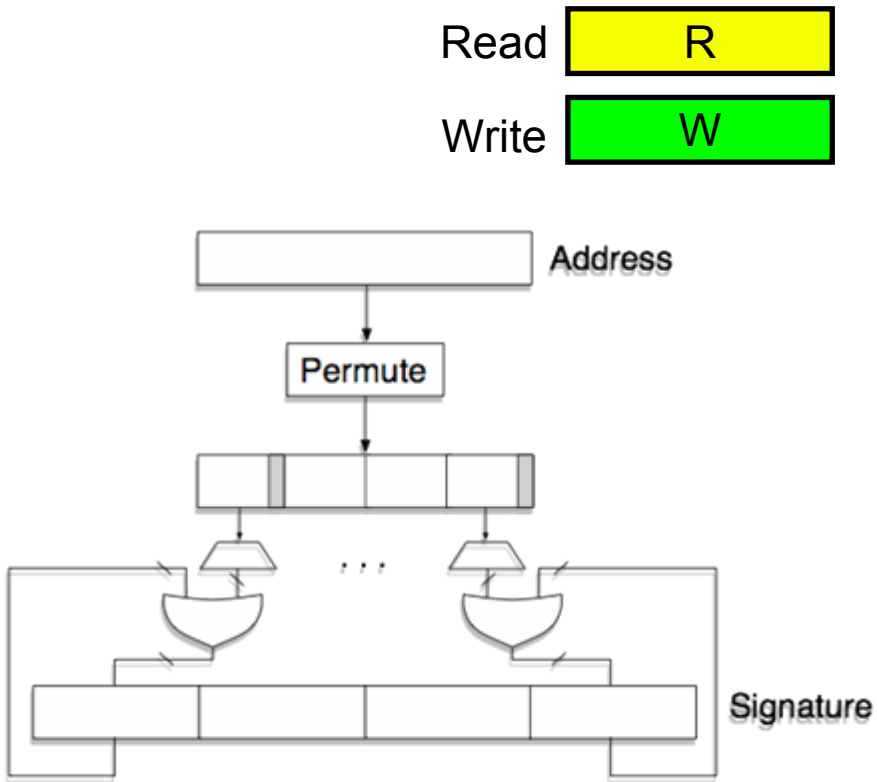


- Leader:
  - Overclocked, delivers high-performance
  - May suffer errors
- Checker:
  - Gets data prefetches and branch hints from Leader
  - Checks that there are no errors
  - Executes fast thanks to Leader
- Use TLS-technology to buffer cache state in the Leader until it can be compared to Checker

Goal 5: Performance Revisited (Simplify Hardware)

Proposal: Bulk Operations

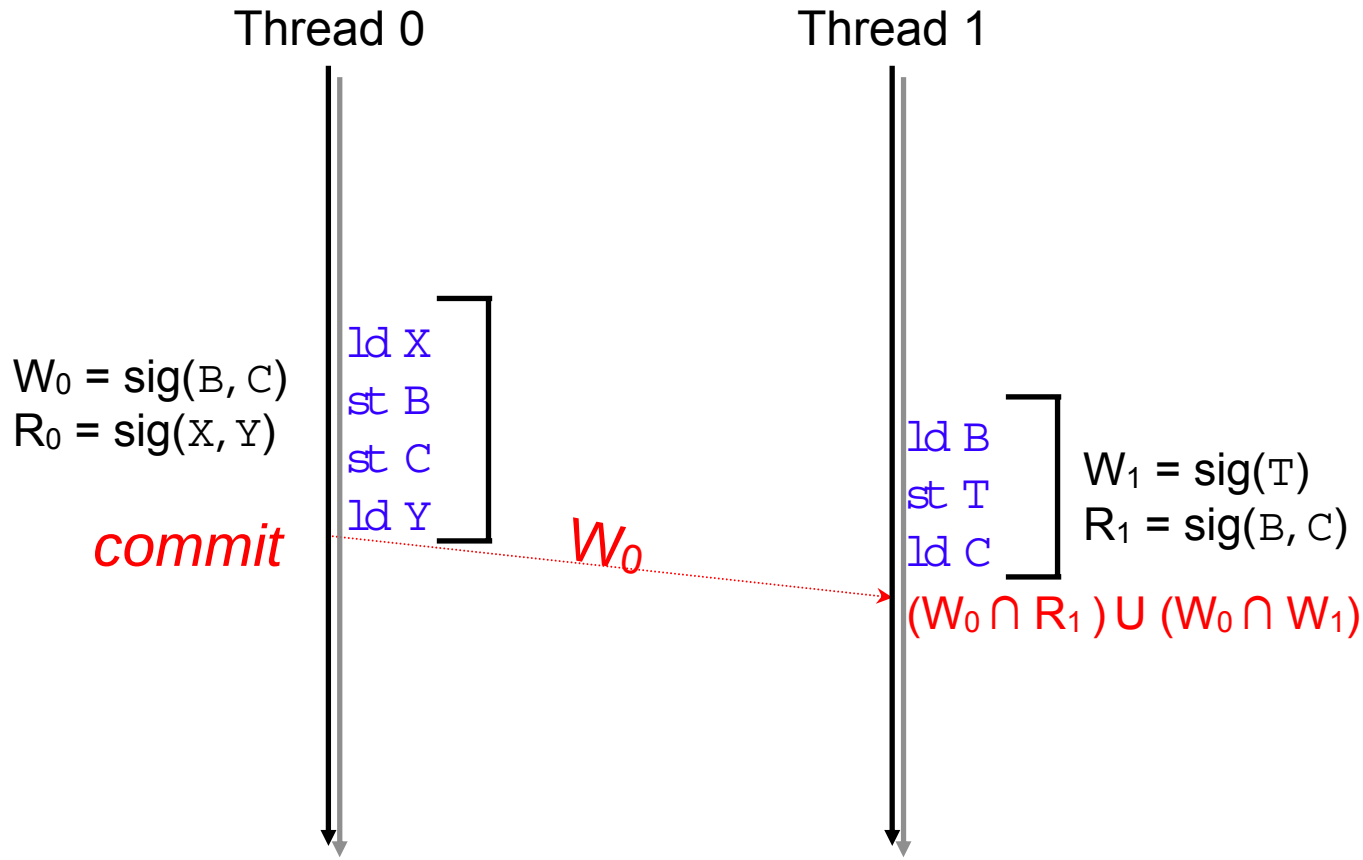
- Encode in HW the addresses accessed by thread in signatures



Proposal: Bulk Operations (II)

- Support signature operations in FU 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
- Works for TLS and for Transactional Memory

Bulk Address Disambiguation [Ceze ISCA06, ISCA07]

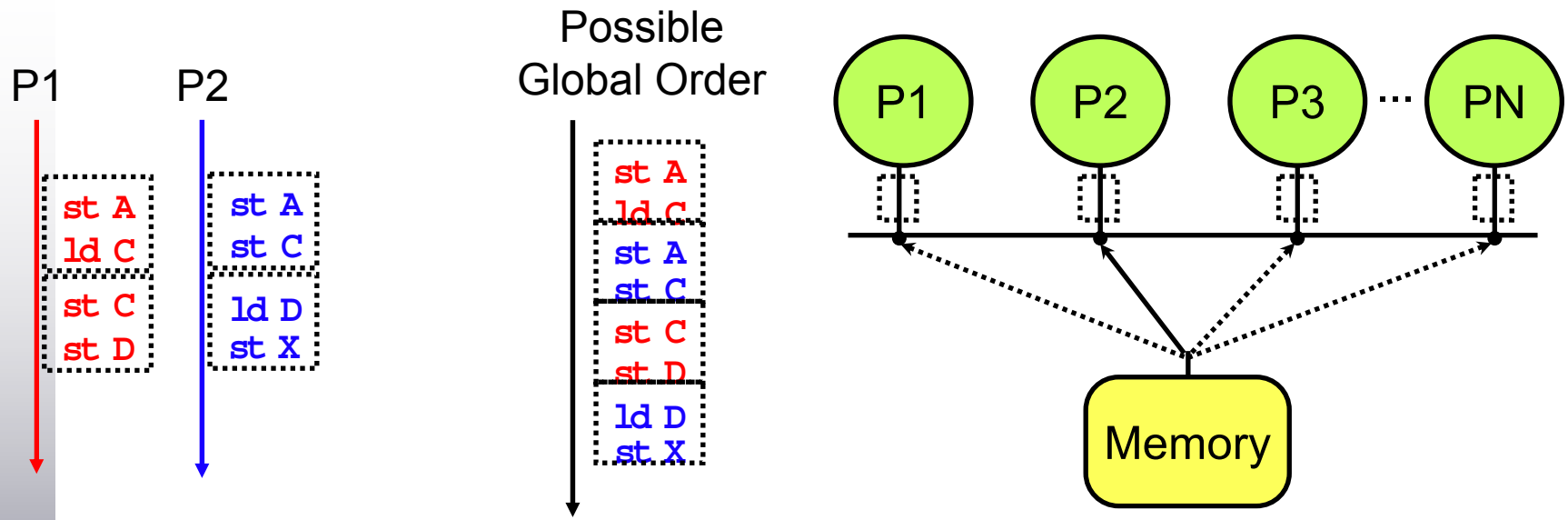


- Signature operations directly supported in HW

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

BulkSC: Bulk Enforcement of Sequential Consistency (SC) [Ceze ISCA07]



➡Group instructions into Chunks, enforce SC only at Chunk granularity

Execute a chunk atomically and in isolation, like a **single instruction**

Support SC:

- At substantially low hardware complexity
- Keeping high performance
- Retaining programmability

Summary: What We Learned

- Tremendous versatility of the speculative multithreading concepts
  - Performance (implicit and explicit parallelism)
  - Programmability, debuggability
  - Hardware reliability
- Speculation does not need to be power inefficient
- Since programmability is crucial, we may soon see variations of this technology in commercial hardware
- Substantial ideas to mine in the multicore era

Lessons Learned in Designing Speculative Multithreading Hardware

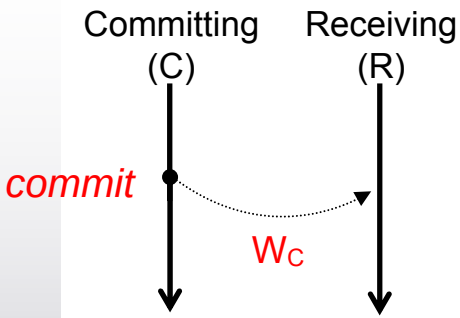
Josep Torrellas

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

Conclusions

- Mechanisms of speculative multithreading
  - Improve performance through parallelization of sequential codes
  - Help performance in explicitly parallel codes & single threads
  - Enhance software dependability
  - Support hardware reliability
  - Its hardware is amenable to simplification
- We may soon see variations of it in commercial hardware
- Substantial ideas to mine in the multicore era

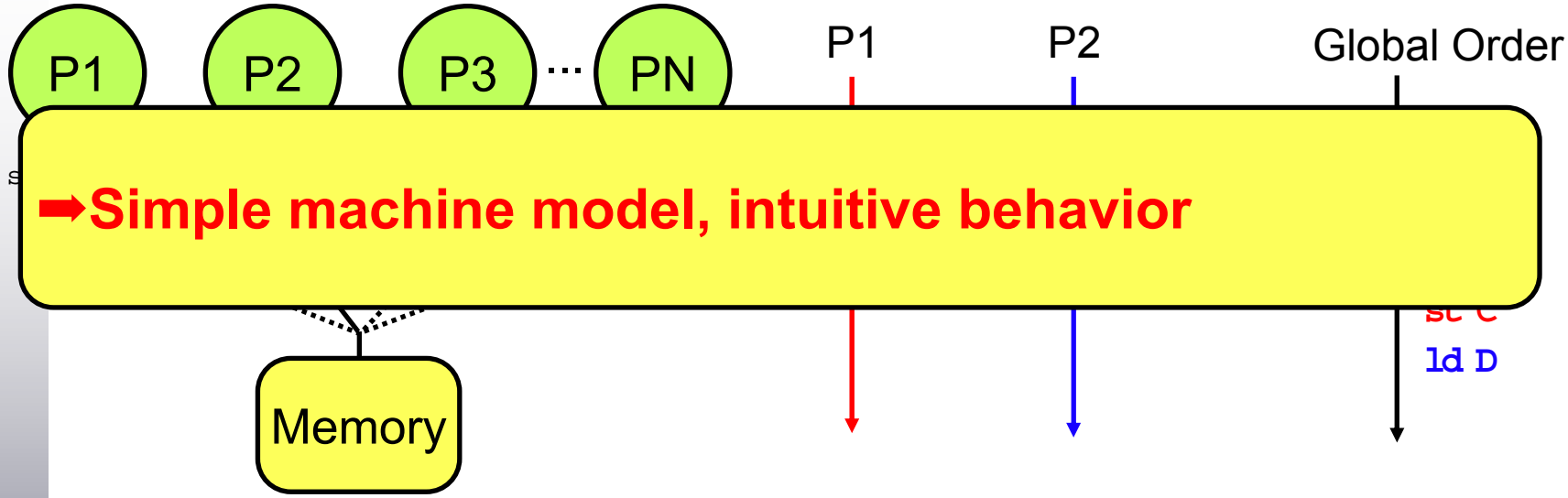
Bulk Disambiguation [Ceze ISCA06, ISCA07]



$$(W_C \cap R_R \neq \varnothing) \cup (W_C \cap W_R \neq \varnothing)$$

- Set operations map directly to signature operations
- Encoding may cause unnecessary squashes

Sequential Consistency (SC)



- **Per-processor program order**: memory operations from individual processors maintain program order
- **Single sequential order**: the memory operations from all processors maintain a single sequential order

Problems with SC Enforcement

- Low Performance
  - restrictions on performance-enhancing reordering of memory operations
  - displacements
  - coupled with key structures (LSQ, ROB, reg file, $)
  - typically fine-grain (instruction-level) undo
- Most current systems do not support SC

➡We would like to change that!

➡Support SC with simple hardware and high performance

Transactional Memory

- See the previous talk

Bulk Operations Pros & Cons

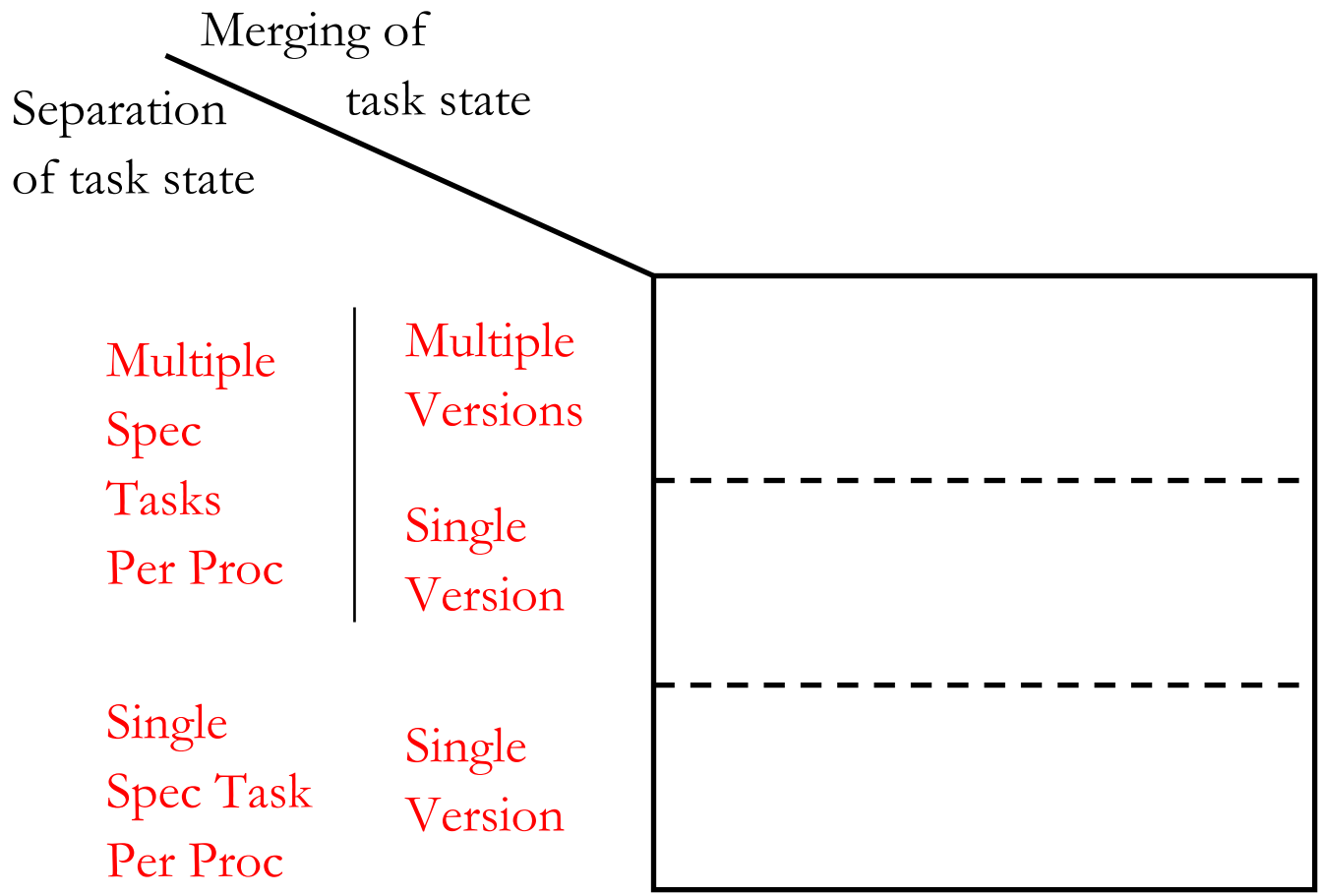
✓ Major conceptual and implementation simplicity

✗ Inexact operations (superset)

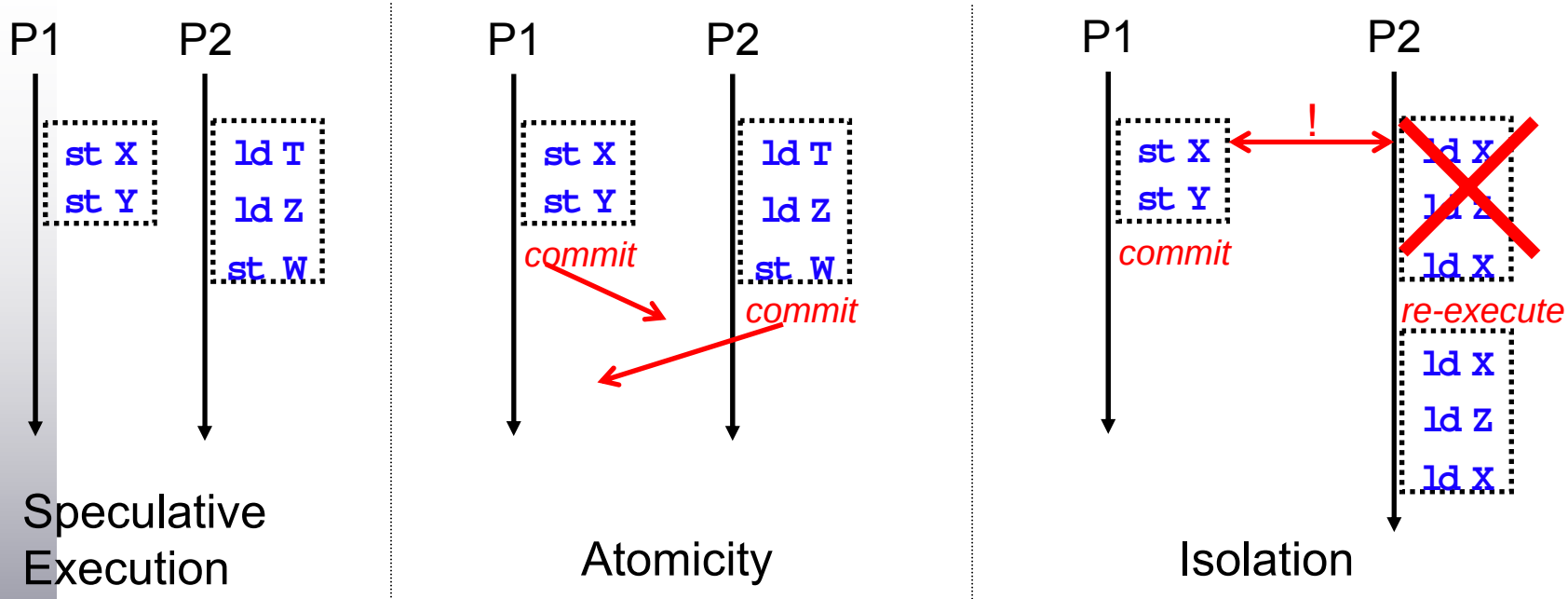
✓ Correctness is guaranteed

✓ Competitive performance compared to current schemes

Taxonomy of Buffering Approaches [Garzaran HPCA03]



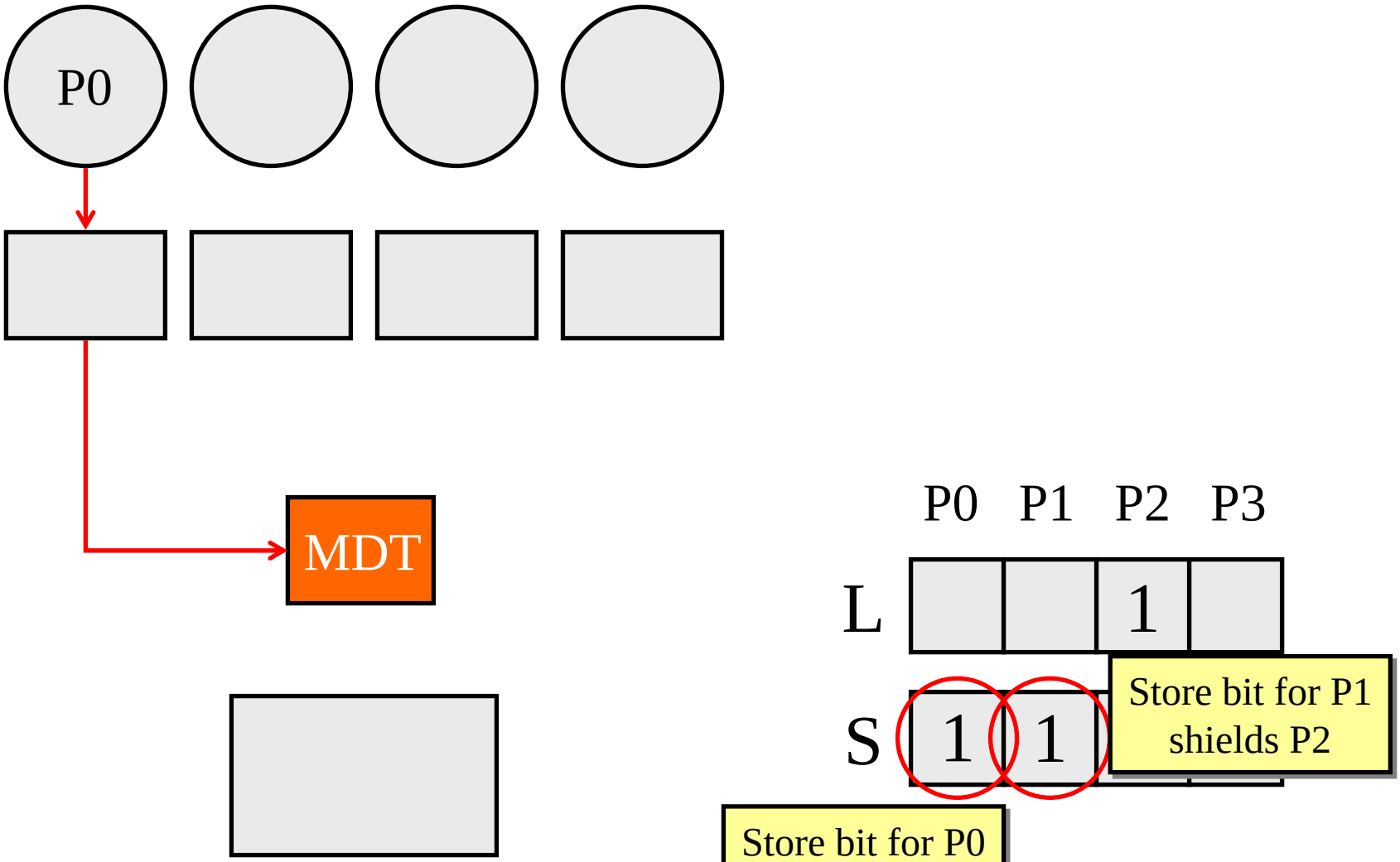
Chunk Execution: Atomicity and Isolation



Atomicity: all updates in the chunk are made visible to other processors at once (all or nothing)

Isolation: a chunk should not see "outside" state changing during its execution

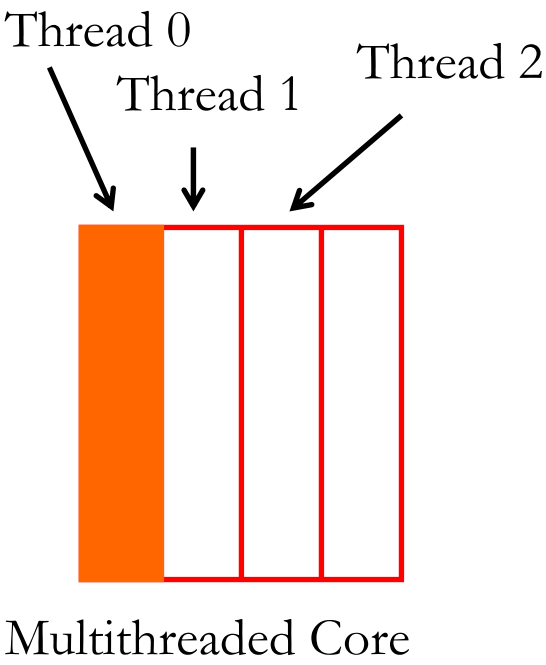
Write & Shielding

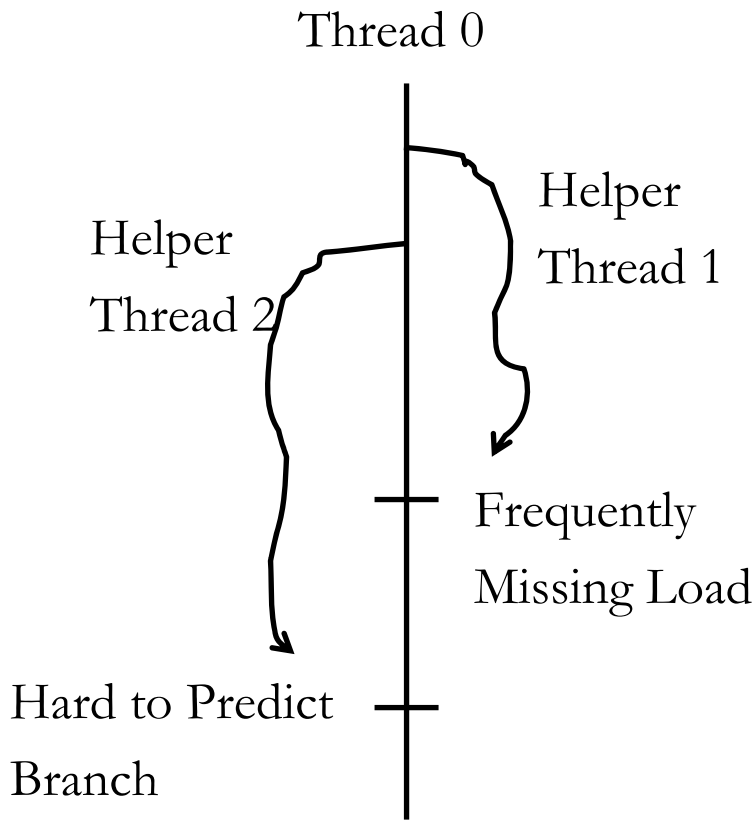


The Challenge

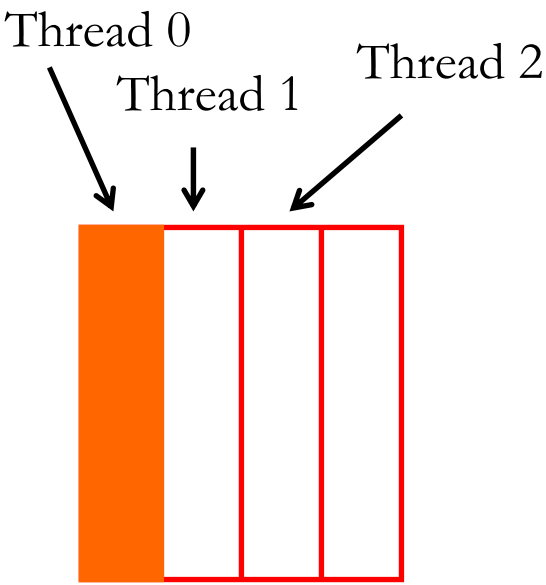
- Design parallel architectures that make it easy to write parallel code
- **Application to explicitly parallel codes:**
  - **Speculative Synchronization**
- Interesting support: ability to UNDO a speculative task
  - Application to enhance ILP
  - Application to debugging

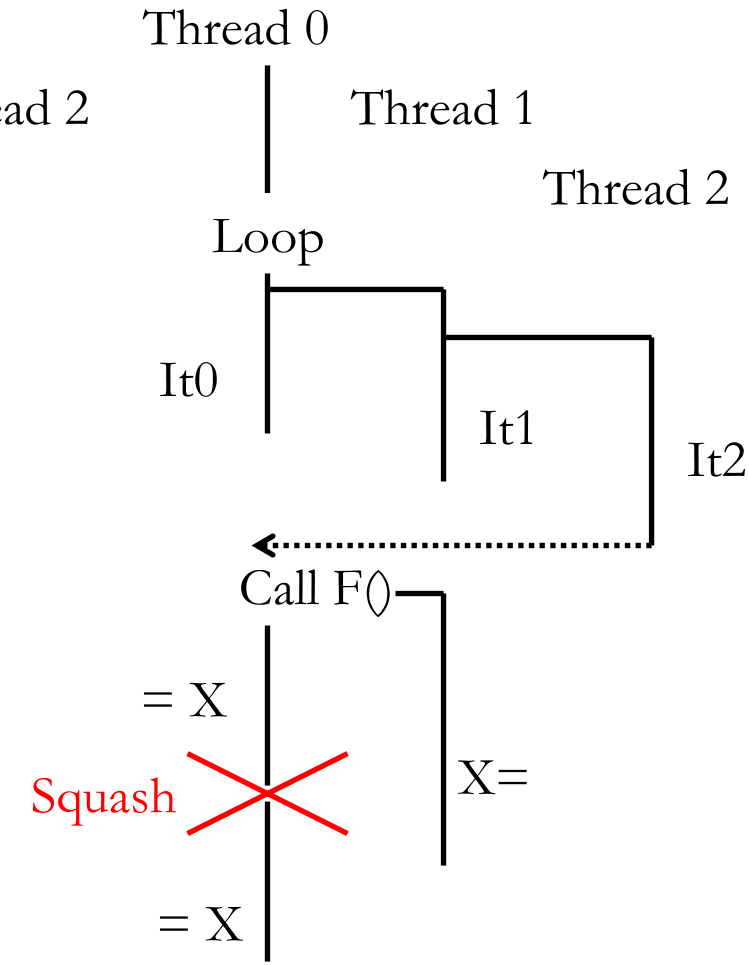
Multithreading Basics: Helper Threads





Speculative Multithreading (SM)





Different Reaction Modes

- Reactions when a monitoring function returns FALSE (indicating an error):
  - ReportMode: report the error and continue
  - BreakMode: pause right after the triggering access
  - RollbackMode: rollback to the most recent checkpoint (need checkpoint support)

Detect Triggering Accesses

- When to detect?
  - Reads: during the read operation
  - Stores: during the pre-touch operation
- How to detect?
  - Checking RWT in parallel with TLB lookup
  - Checking WatchFlags in load/store queues
  - Checking WatchFlags in the caches
- When to trigger (executing the monitoring function)?
  - At the retirement of the triggering access
  - Use a *Trigger* bit in ROB

iWatcher: Main Idea

- Associate a monitoring function with a watched memory location
  - At a triggering access to a watched location, the associated monitoring function(s) are triggered by hardware and executed
- Use SM to reduce overhead and support rollback
  - Execute the main thread speculatively in parallel with monitoring function(s)
  - Use SM to buffer state for rollback in case of errors reported by monitoring function(s)

iWatcher User Interface

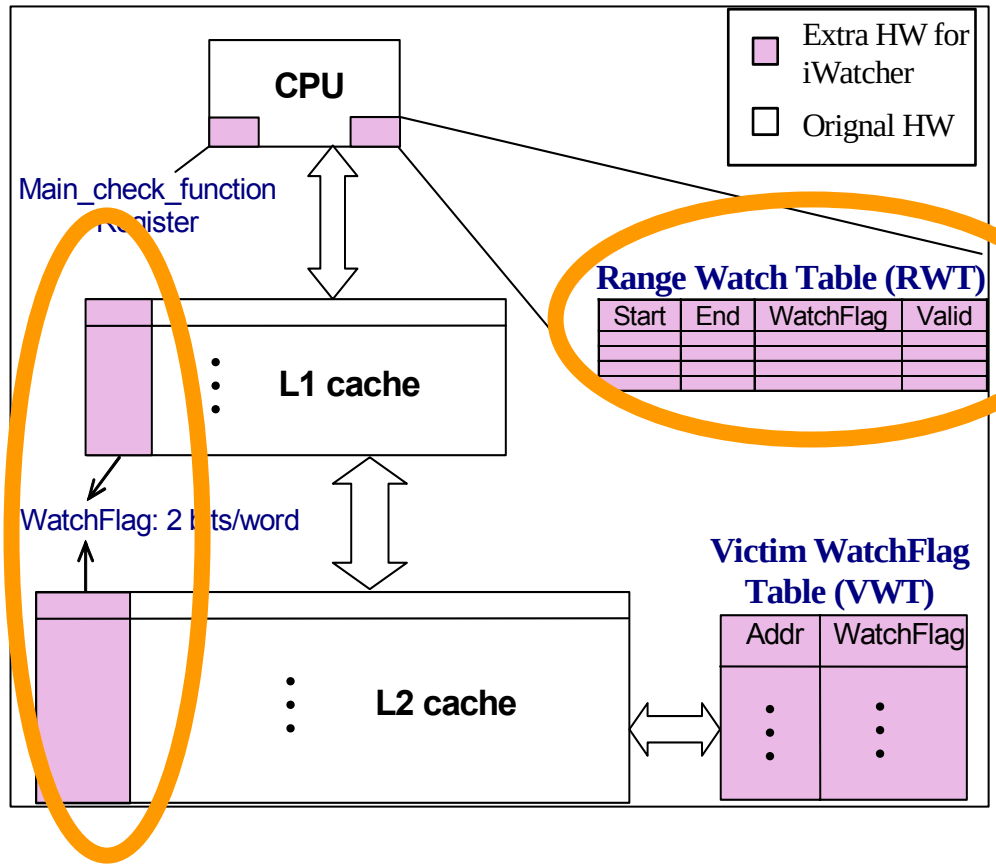
- Turn on/off monitoring for a memory location
  - iWatcherOn (MemAddr, Length, WatchFlag, ReactMode, MonitorFunc, Param1, Param2, …, ParamN)
  - iWatcherOff (MemAddr, Length, WatchFlag, MonitorFunc)
  - A global switch
    - EnableiWatcher
    - DisableiWatcher

iWatcher Design Overview

- Hardware:
  - Detecting triggering accesses
  - Triggering the main monitoring function
- Software
  - Manage associations between watched locations and monitoring functions
  - Call the appropriate monitoring function upon a triggering access

Watch a Memory Region

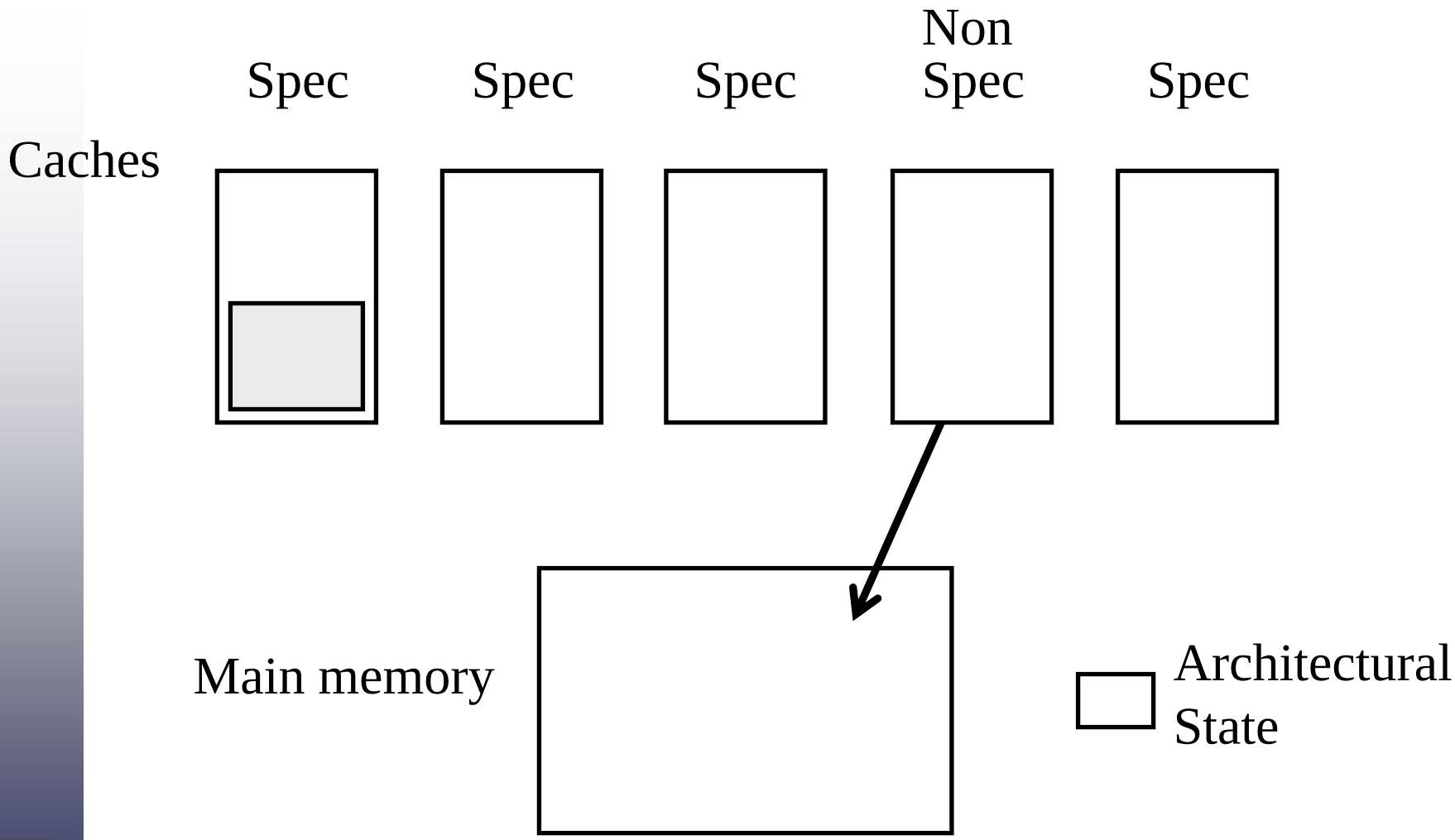
- Hardware
  - Large region: allocate an RWT entry
  - Small region: set WatchFlags in caches
- Software
  - Add monitoring function info to check table



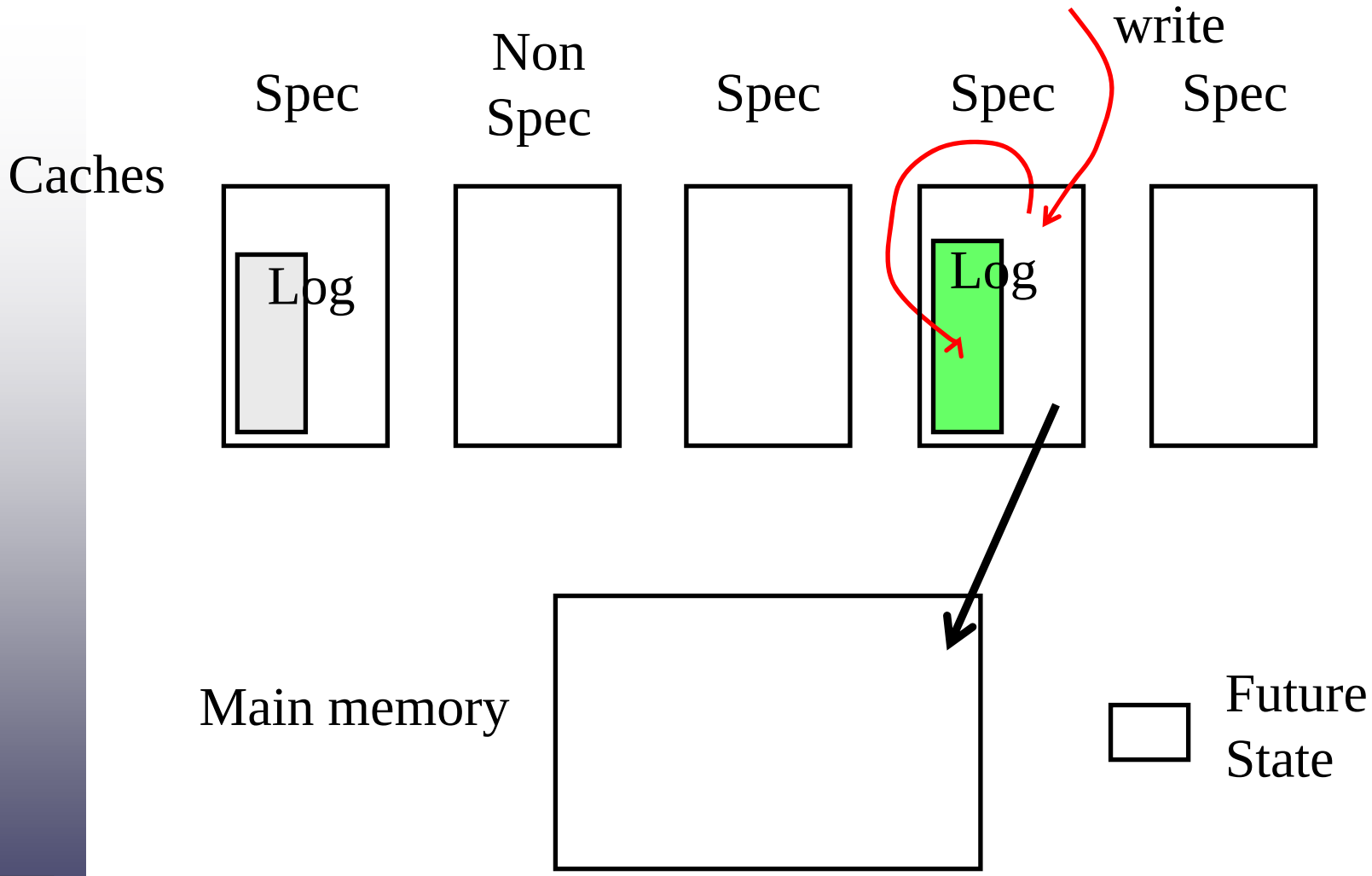
Repair: Pattern Matching

- Analysis resulted in a detailed signature
  - Instruction & data addresses, data values, timing, etc.
- Pattern-match it with a library of common races:
  - Suggest repair to programmer, or
  - Download bug-specific patch, or
  - Try to automatically re-introduce missing ordering
- Squash chunks, re-execute with corrections

Merging of Task State: Architectural MM



Merging of Task State: Future MM



Implementation Details

- When entering the speculative section:
  - Fence-type instruction that creates a checkpoint of the register state
- While executing speculative section:
  - Buffer all memory updates in the cache -- cannot update memory
  - Mark cache lines read and written
  - Monitor for errors or violations
- If an error or violation occurs:
  - Invalidate updated cache lines, reset marks, restore the register checkpoint
- Successful end of speculation:
  - Reset marks
  - Allow updated cache lines to be displaced to memory

Evaluation for Data Races [ISCA03]

- 4-processor chip multiprocessor
- Splash2 applications

- Want to know:
  - Overhead
  - Effectiveness

Main Results

- Low overhead in error-free execution: **6% avg**
- Highly effective: Detect, Analyze & Correct race bugs
  - Existing races
    - Synchronization through plain variables
    - Other existing data races
  - Induced races
    - Remove lock
    - Remove barrier

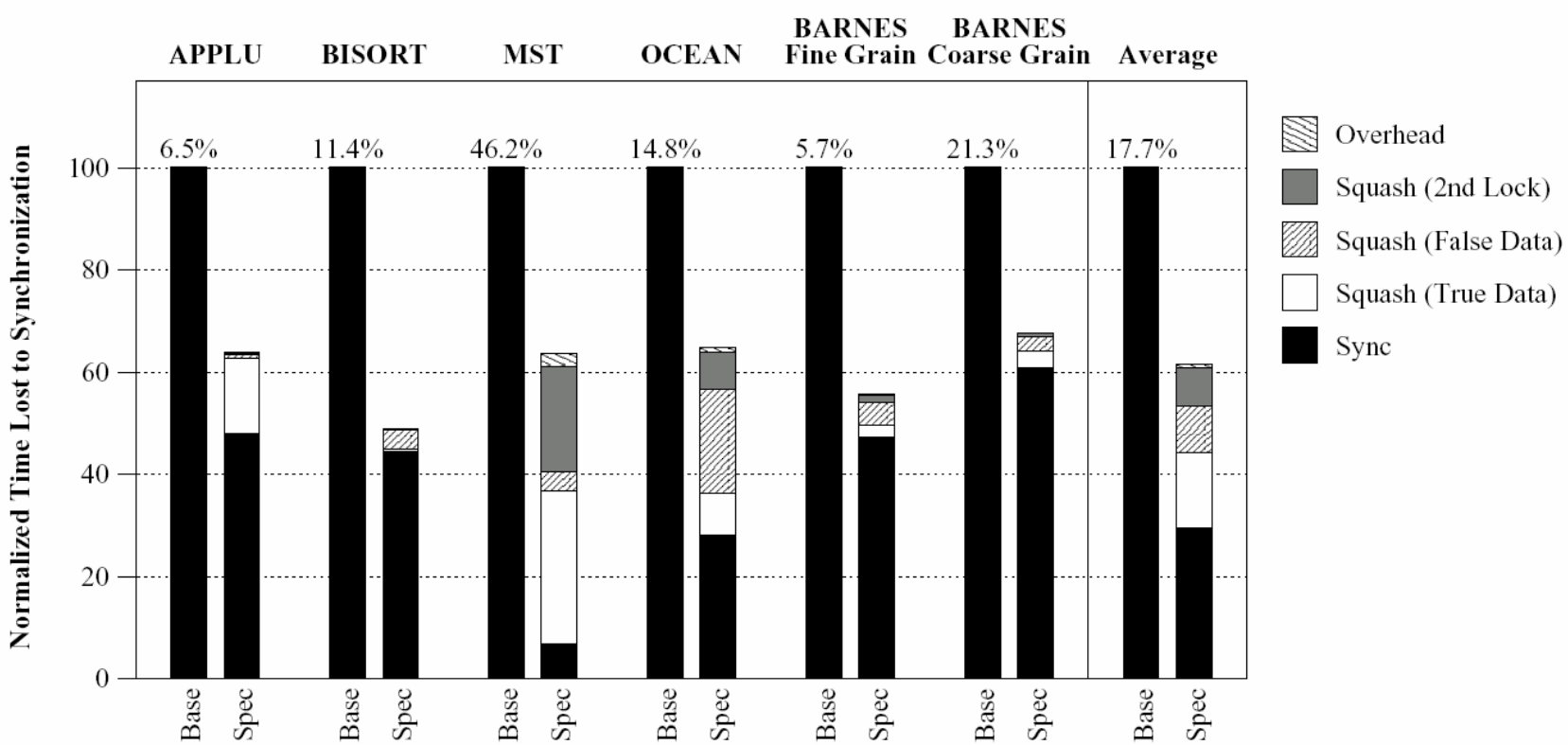
How Good ReEnact is to:

	Detect	Rollback	Analyze	Match
Sync through plain variables	✓	✓	✓	✓
Other Existing Data Races	✓	✓	✓	No
Induced Bugs: Removed Lock	✓	✓	✓	✓
Induced Bugs: Removed Barrier	✓	~	~	~

Spec Synchronization Evaluation [ASPLOS02]

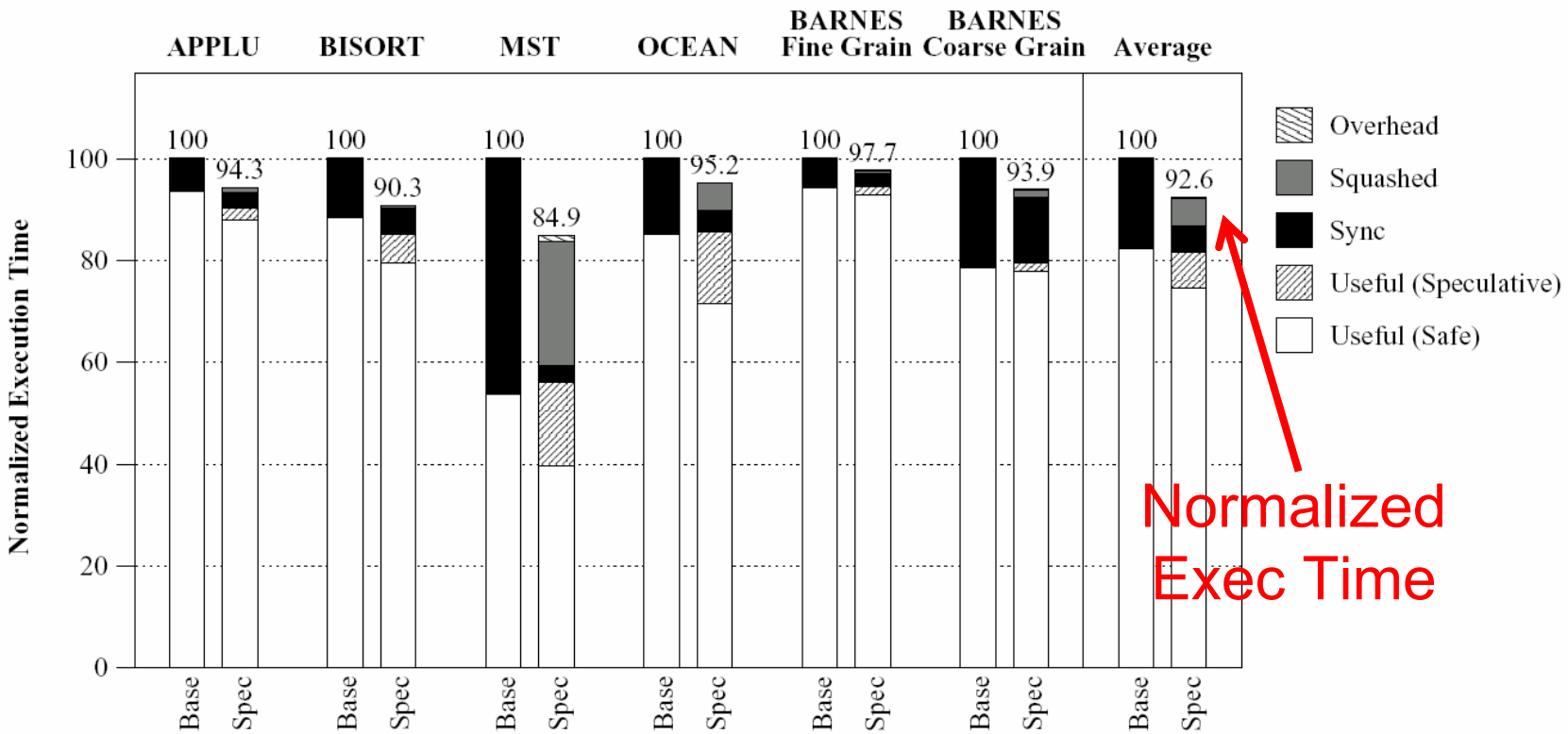
- Mix of parallel codes
- Parallelization:
  - Compiler [16 processors] (*applu*)
  - Annotated [16 processors] (*mst*, *bisort*)
  - Hand [64 processors] (*ocean*, 2×*barnes*)

Sync Time Reduction



Large reduction: 40%
Room for improvement

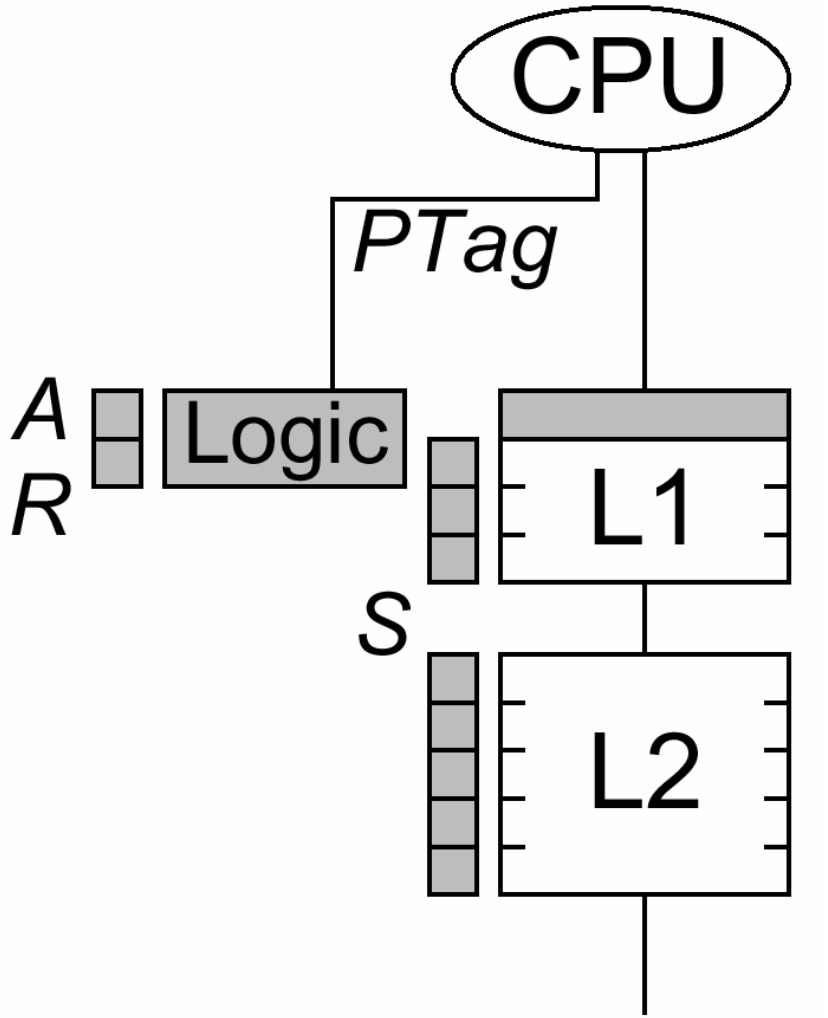
Execution Time Reduction



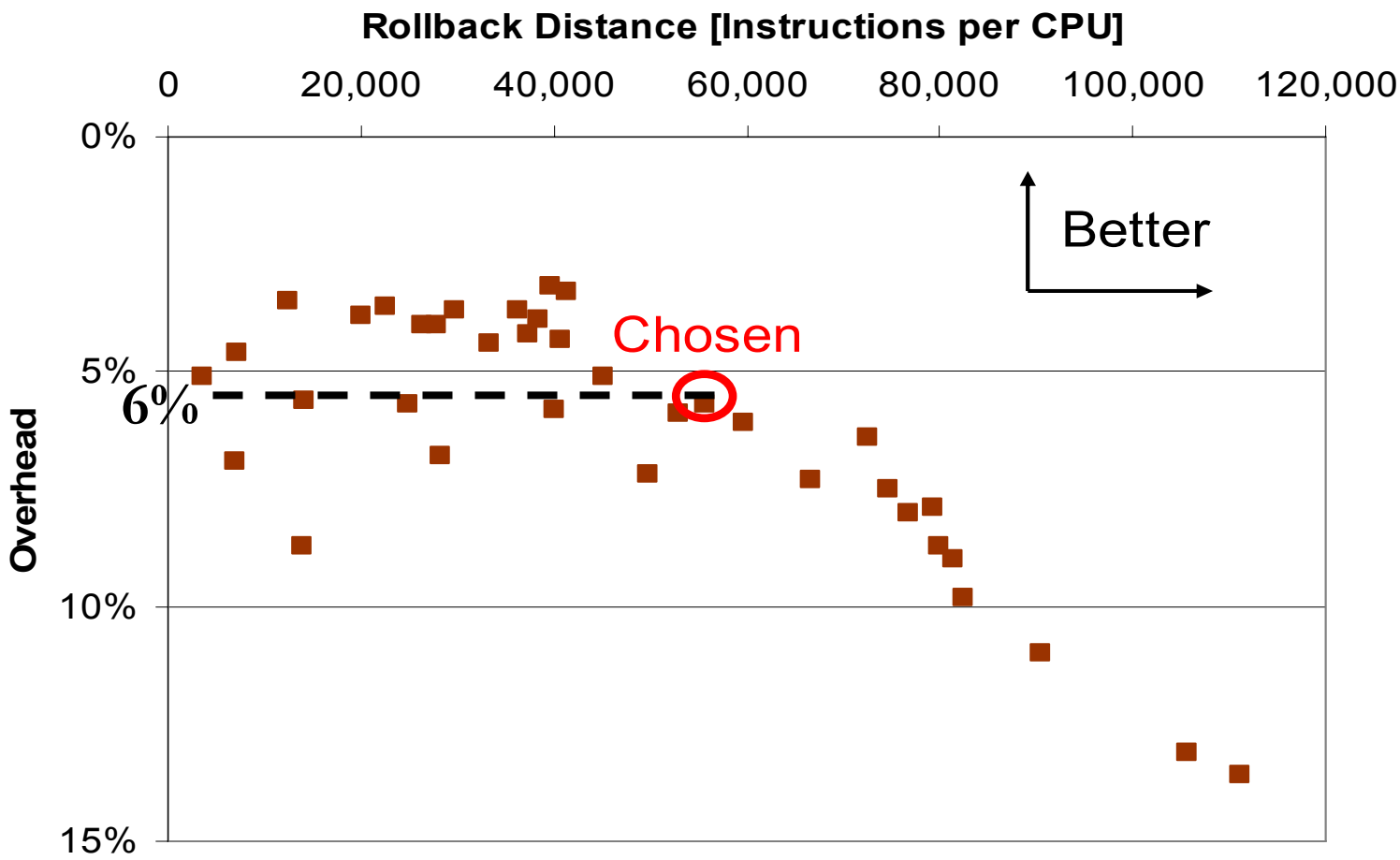
Across-the-board reduction

Speculative Synchronization Unit

- Extends cache controller
- Simple hardware:
  - 1 spec bit/line
  - Some control logic



ReEnact: Overhead



Speculative Lock Request

- Processor side:
  - Program SSU for speculative lock
  - Checkpoint register file
- SSU side:
  - Initiate T&T&S loop on lock variable
- Use caches as speculative buffer (like TLS)
  - Set *Speculative* bit in lines accessed speculatively

Lock Acquire

- SSU acquires lock (T&S successful)
  - Clears all *Speculative* bits → one-shot commit
  - Becomes idle
- Release (store) later by processor

Release while Speculative

- Processor issues release, SSU still active
  - SSU intercepts release (store) by processor
  - SSU toggles *Release* bit – "already done"
- When lock becomes available later
  - SSU:
    - Does not perform T&S
    - Clears all *Speculative* bits → one-shot commit

Memory Access Conflict

- External coherence actions
  - Request to safe line: service normally
  - Request to spec line: squash thread
    - Invalidate lines marked *Speculative+Dirty* → one-shot squash
    - Roll back & restart at sync point
- Safe threads never squashed → forward progress
- All safe-to-spec in-order dependences tolerated

Speculative Flags and Barriers

- Flag spin: Test only – no T&S
  - Handle like "Release while Speculative" case
- Barrier: leverage flag spin support
  - Update thread counter
  - If not last one, spin on flag speculatively

References

SPECULATIVE MULTITHREADING (THREAD LEVEL SPECULATION)

H. Akkary and M. Driscoll.
A Dynamic Multithreading Processor.
Intl. Symp. on Microarchitecture, pages 226--236, Dec. 1998.

M. Cintra, J. Martinez, and J. Torrellas.
Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors.
Intl. Symp. on Computer Architecture, pages 13--24, June 2000.

R. Figueiredo and J. Fortes.
Hardware Support for Extracting Coarse-grain Speculative Parallelism in Distributed Shared-memory Multiprocesors.
Proc. Intl. Conf. on Parallel Processing, September 2001.

M. Frank, W. Lee, and S. Amarasinghe.
A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics.
Tech. Rep., MIT/LCS Technical Memo MIT-LCS-TM-619, July 2001.

M. Franklin and G. Sohi.
ARB: A Hardware Mechanism for Dynamic Reordering of Memory References.
IEEE Trans. Computers, 45(5):552--571, May 1996.

M. Garzaran, M. Prvulovic, J. Llaberia, V. Vinals, L. Rauchwerger, and J. Torrellas.
Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors.
International Symposium on High Performance Computer Architecture, Feb. 2003.

M. Garzaran, M. Prvulovic, J. Llaberia, V. Vinals, L. Rauchwerger, and J. Torrellas.
Using Software Logging to Support Multi-Version Buffering in Thread-Level Speculation.
International Conference on Parallel Architectures and Compilation Techniques, Sept. 2003.

References

S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi.
Speculative Versioning Cache.
Intl. Symp. on High-Performance Computer Architecture, pages 195--205, February 1998.

M. Gupta and R. Nim.
Techniques for Speculative Run-Time Parallelization of Loops.
Proc. Supercomputing 1998}, November 1998.

L. Hammond, M. Willey, and K. Olukotun.
Data Speculation Support for a Chip Multiprocessor.
Intl. Conf. on Arch. Support for Prog. Lang. and Oper. Systems, pages 58--69, October 1998.

T.Knight.
An Architecture for Mostly Functional Languages.
ACM Lisp and Functional Programming Conf., pages 500--519, August 1986.

V. Krishnan and J. Torrellas.
A Chip-Multiprocessor Architecture with Speculative Multithreading.
IEEE Trans. on Computers, pages 866--880, September 1999.

P. Marcuello and A. Gonzalez.
Clustered Speculative Multithreaded Processors.
Proc. 1999 Intl. Conf. on Supercomputing, pages 365--372, June 1999.

M. Prvulovic, , M. Garzaran, L. Rauchwerger, and J.Torrellas.
Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization.
Intl. Symp. on Computer Architecture, pages 204--215, July 2001.

L. Rauchwerger and D.Padua.
The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization.
Conf. on Prog. Lang. Design and Implementation, pages 218--232, June 1995.

References

P. Rundberg and P. Stenstrom.
Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors.
4th Workshop on Multithreaded Execution, Architecture and Compilation, December 2000.

G. Sohi, S. Breach, and T. Vijaykumar.
Multiscalar Processors.
Intl. Symp. on Computer Architecture pages 414--425, June 1995.

J. Steffan, C. Colohan, A. Zhai, and T. Mowry.
A Scalable Approach to Thread-Level Speculation.
Annual Intl. Symp. on Computer Architecture, pages 1--12, June 2000.

J.Steffan, C. Colohan, and T. Mowry.
Architectural Support for Thread-Level Data Speculation.
Tech. Rep., CMU-CS-97-188, Carnegie Mellon University, November 1997.

M. Tremblay.
MAJC: Microprocessor Architecture for Java Computing.
Hot Chips, August 1999.

J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew.
The Superthreaded Processor Architecture.
IEEE Trans. on Computers, 48(9):881--902, September 1999.

Y. Zhang.
Hardware for Speculative Run-Time Parallelization in DSM Multiprocessors.
Ph.D. Thesis, Dept. of Elec. and Comp. Engineering, Univ. of Illinois at Urbana-Champaign, May 1999.

Y. Zhang, L. Rauchwerger, and J.Torrellas.
Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors.
Intl. Symp. on High-Performance Computer Architecture, pages 135--139, January 1999.

References

SPECULATIVE SYNCHRONIZATION

J. Martinez and J. Torrellas.
Speculative Synchronization: Applying Thread-Level Speculation to Parallel Applications
International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002.

CHERRY

J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas,
Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors
International Symposium on Microarchitecture, November 2002.

REENACT

M. Prvulovic and J. Torrellas.
ReEnact: Using Thread-Level Speculation to Debug Data Races in Multithreaded Codes.
International Symposium on Computer Architecture, June 2003.

IWATCHER

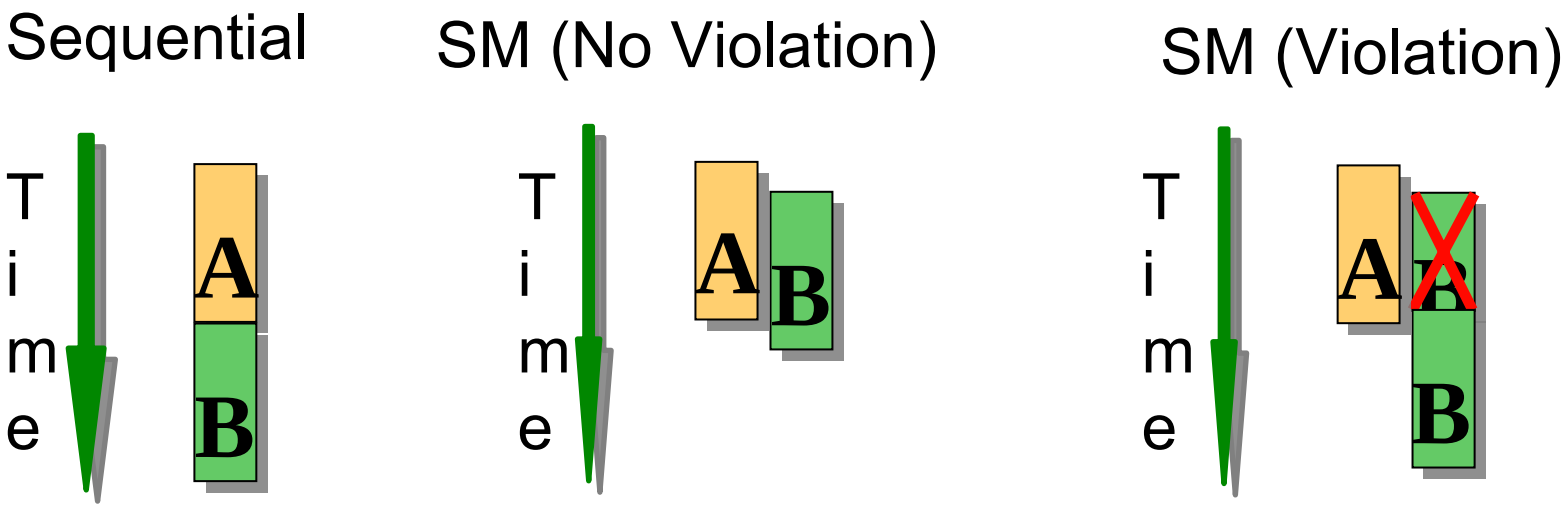
P. Zhou, F. Qin, W. Liu, Y. Zhou and J. Torrellas.
iWatcher: Efficient Architectural Support for Software Debugging
International Symposium on Computer Architecture, June 2004.

Boosting Machine Performance with Speculative Multithreading

Josep Torrellas

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

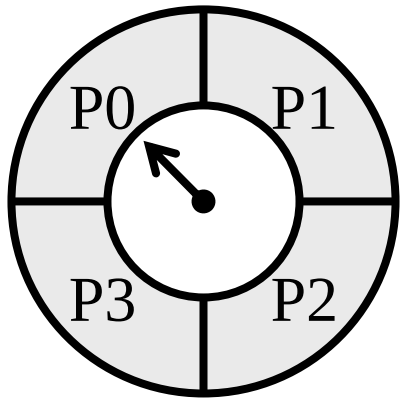
How Speedups are Possible

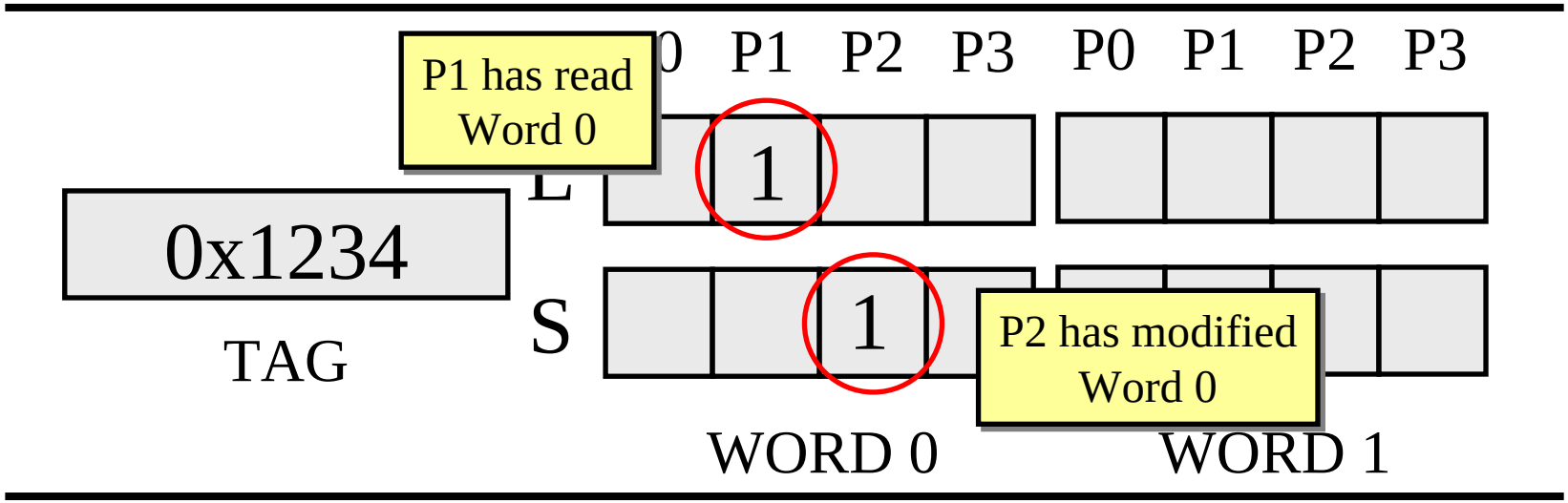


Different Reaction Modes

- Reactions when a monitoring function returns FALSE (indicating an error):
  - ReportMode: report the error and continue
  - BreakMode: pause right after the triggering access
  - RollbackMode: rollback to the most recent checkpoint (need checkpoint support)

Memory Disambiguation Table (MDT)



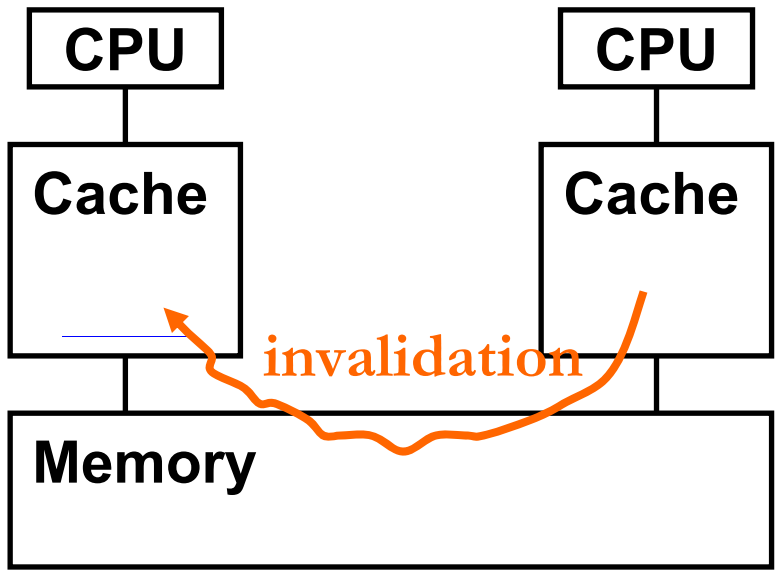


Important Features

- Concurrency possible even if conflicts
  - All in-order safe-to-spec conflicts tolerated
- No order among spec threads → simpler HW
- No programming effort
  - Retargetted macros/directives
- Can coexist with conventional sync at run-time

TLS: Primitive for Software Debugging

- Undo group of tasks (window of buggy code, hopefully)
- Re-execute those tasks only
- Re-execution of tasks is deterministic even under parallelism
- Bonus: detect bugs that appear as communication (e.g. Data Races)

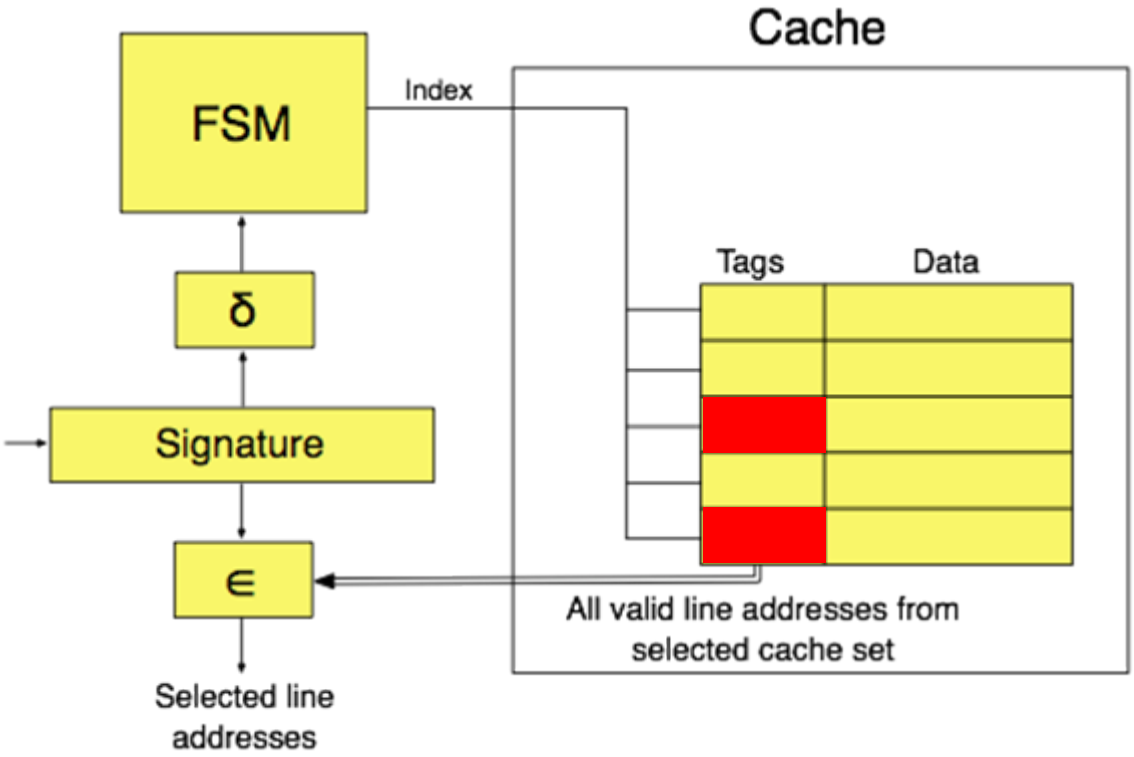


Breaking Code into Chunks

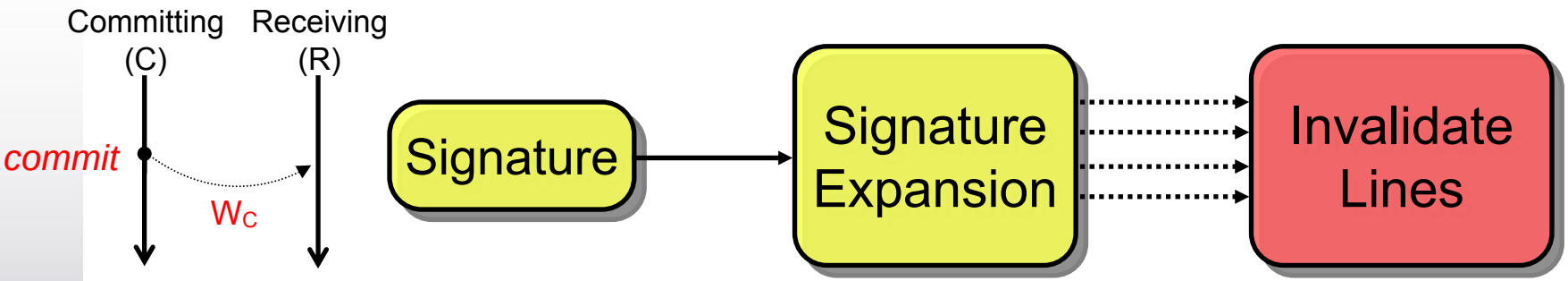
- New chunk begins ⇒ save register state
- Undo (squash) recent chunks if needed
  - Invalidate cache lines, restore saved register state
  - Enables rollback of chunk
- Commit old chunks
  - Allow displacement from cache, free saved register state
  - Makes room for buffering more recent chunks
  - Cannot undo committed chunks

Composed Operation: Signature Expansion

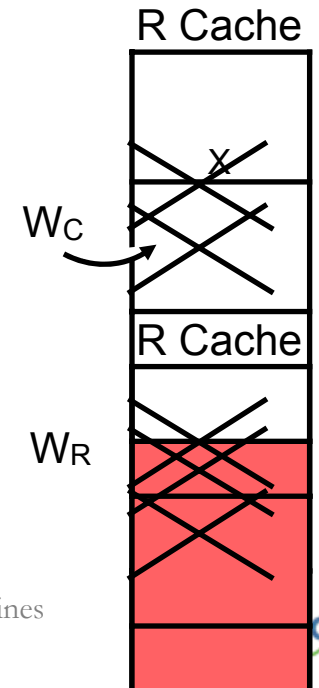
- Select lines in the cache that belong to the signature
  - used in bulk invalidations



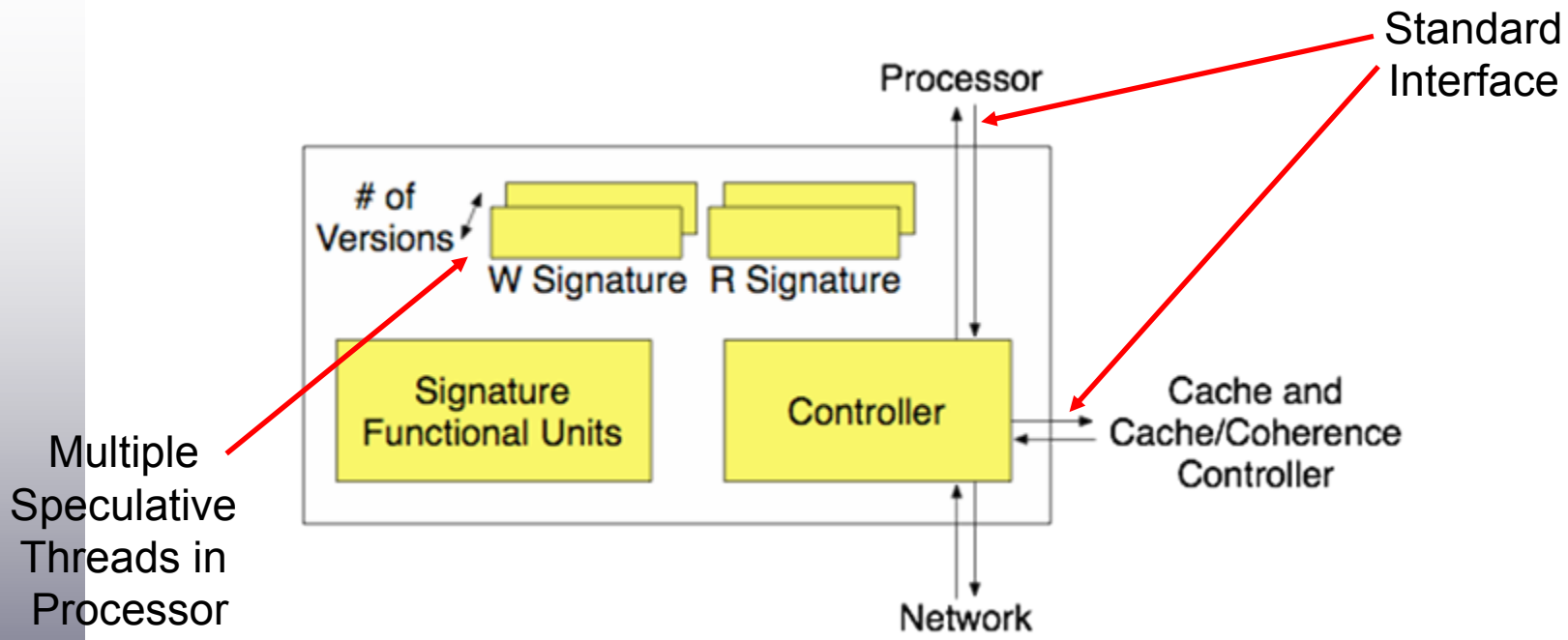
Bulk Invalidation



- 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)

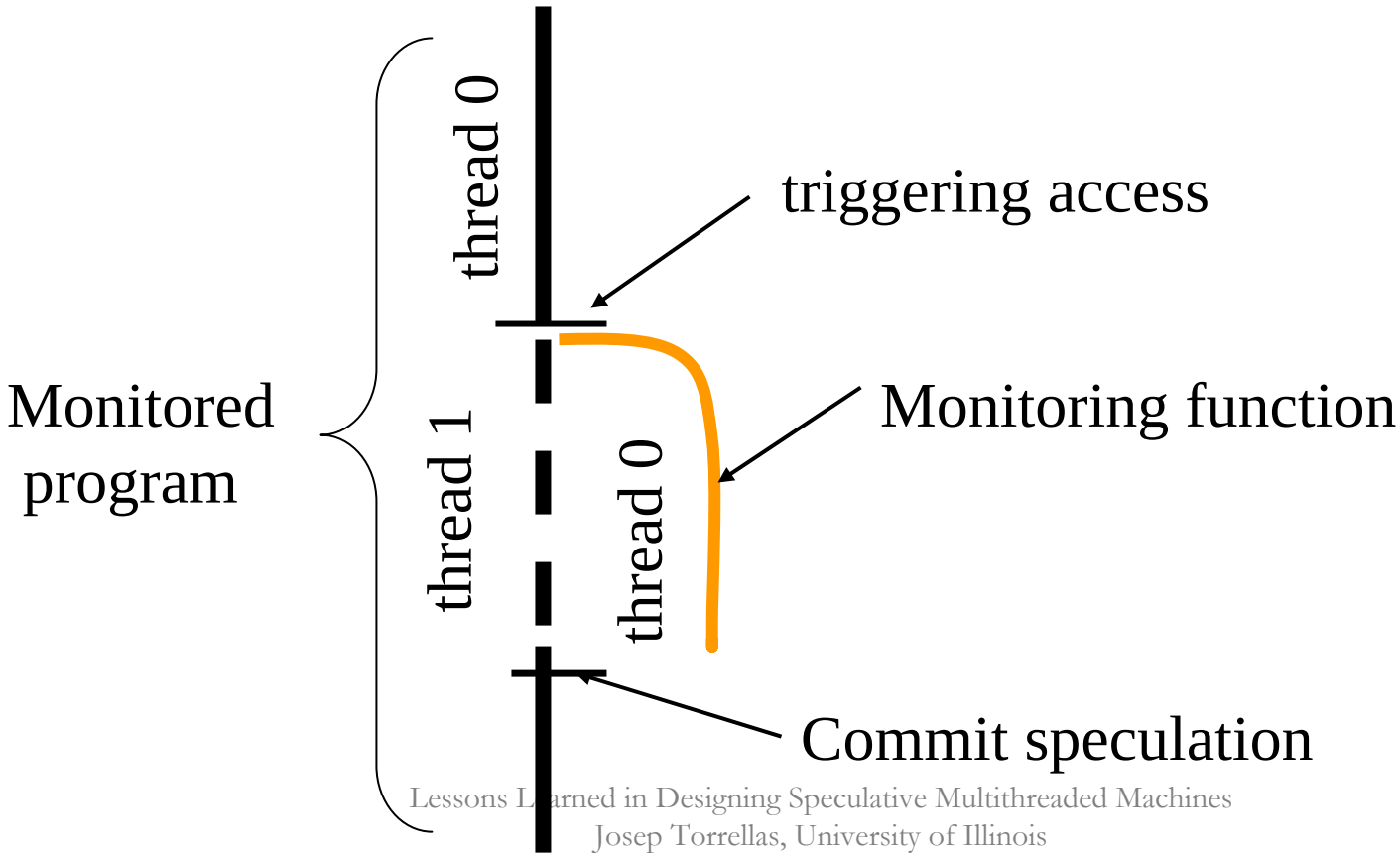


Bulk Disambiguation Module

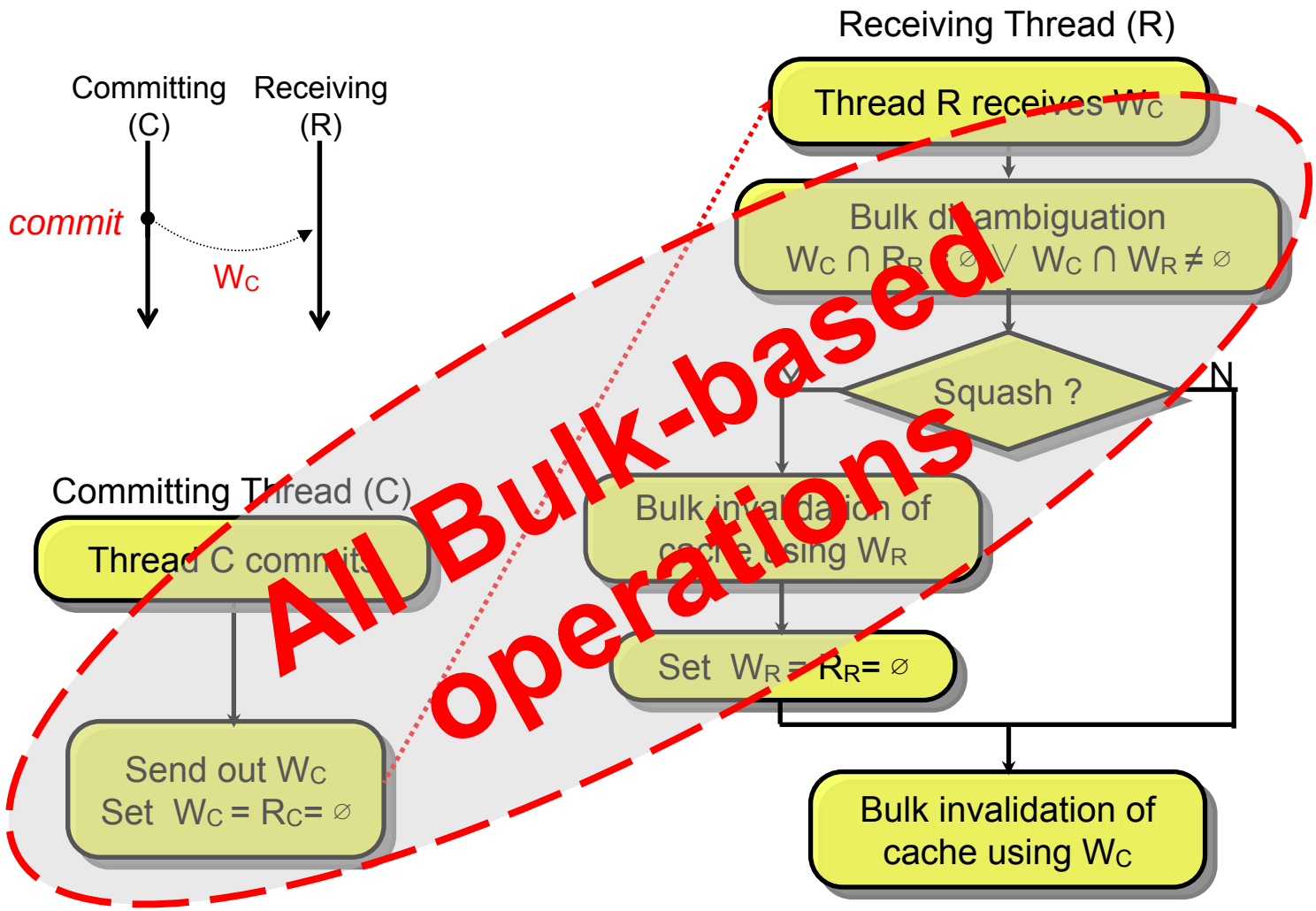


Monitoring Functions

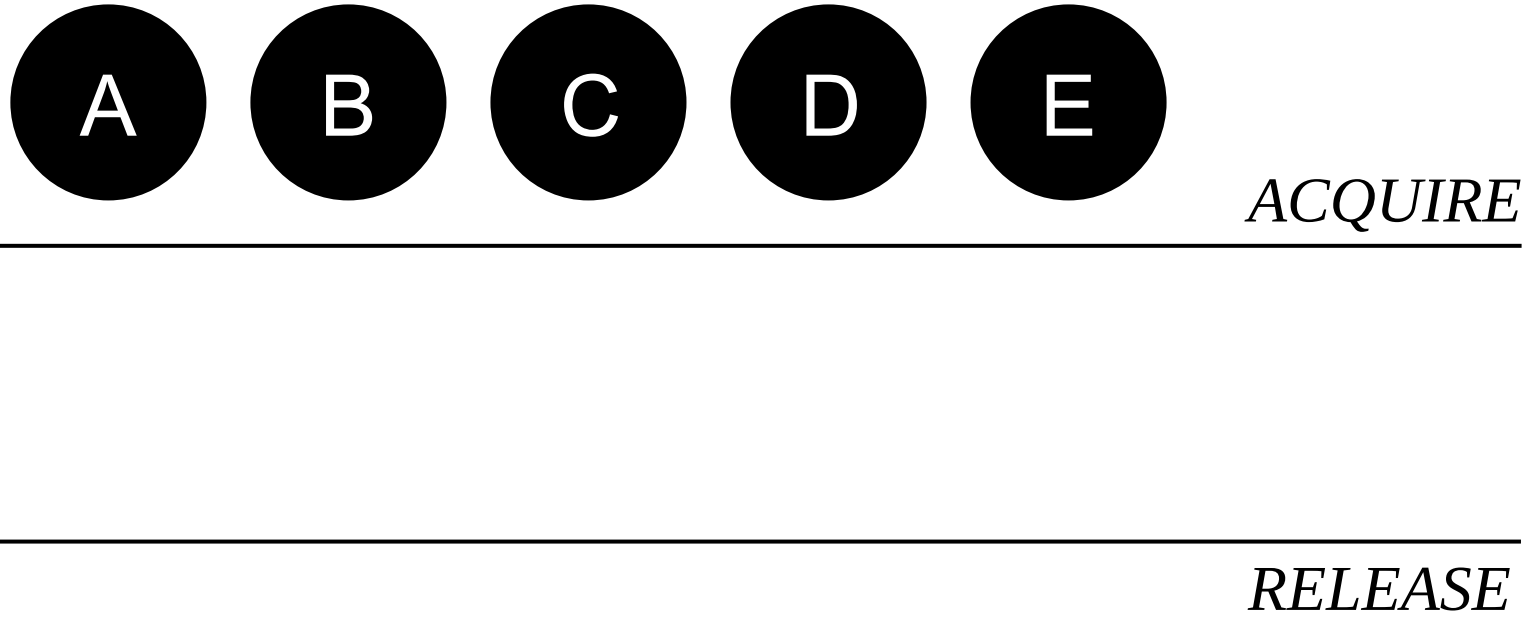
- Triggered by hardware
- Executed in parallel with main program by TLS (optionally)
- Can have any side-effects



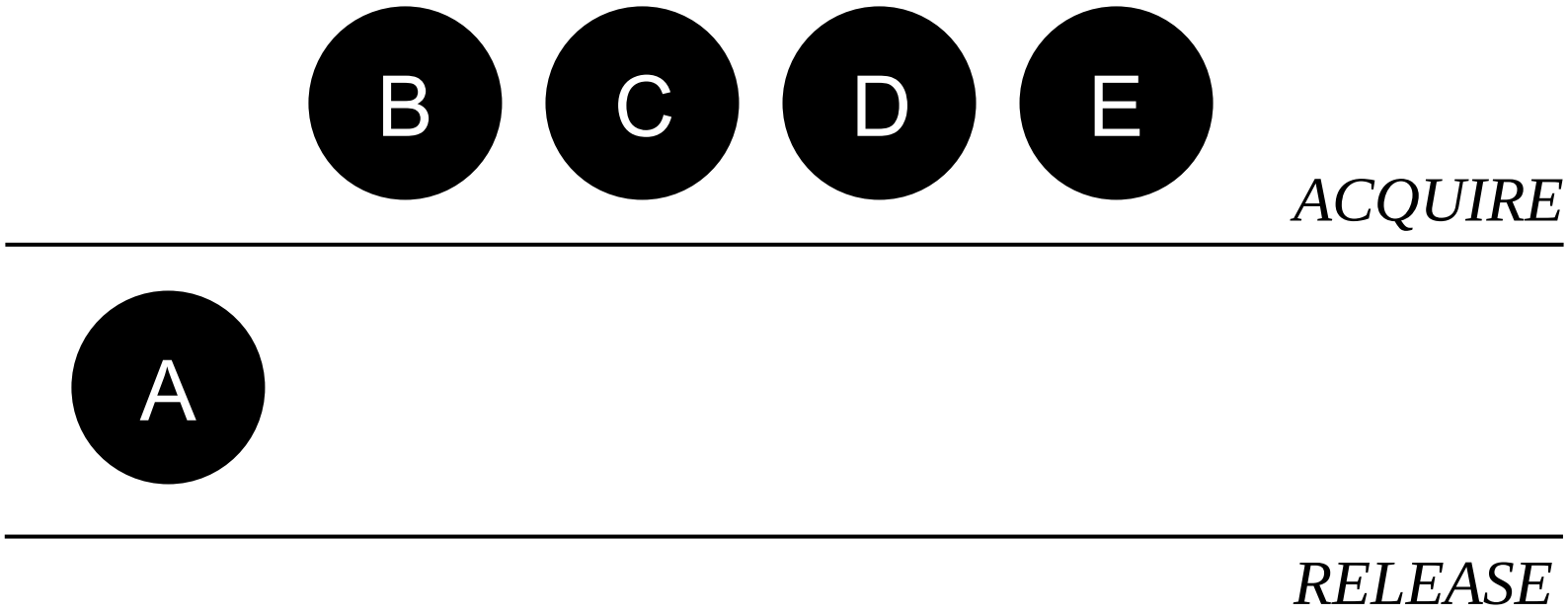
Commit Process



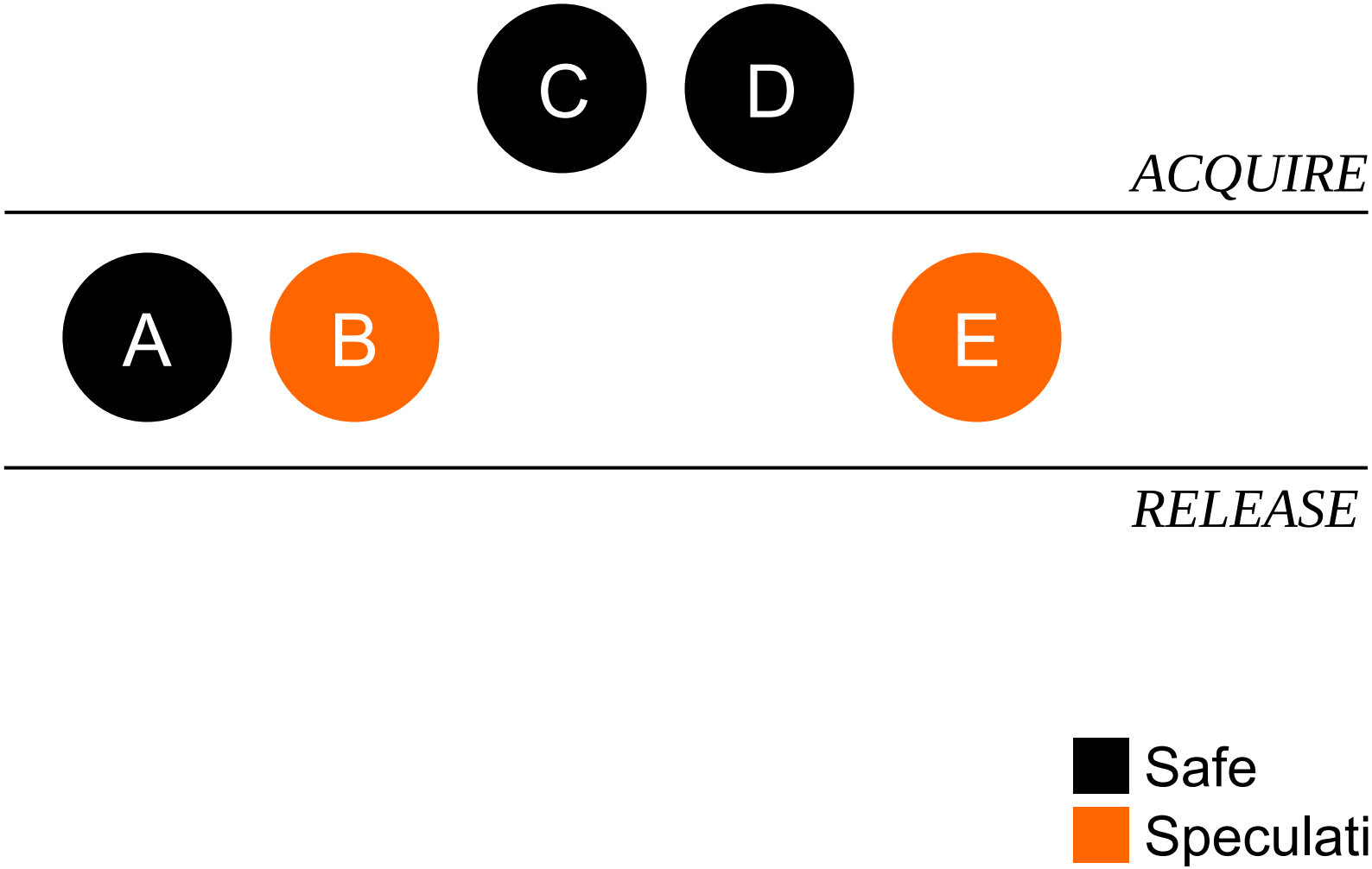
Speculative Lock



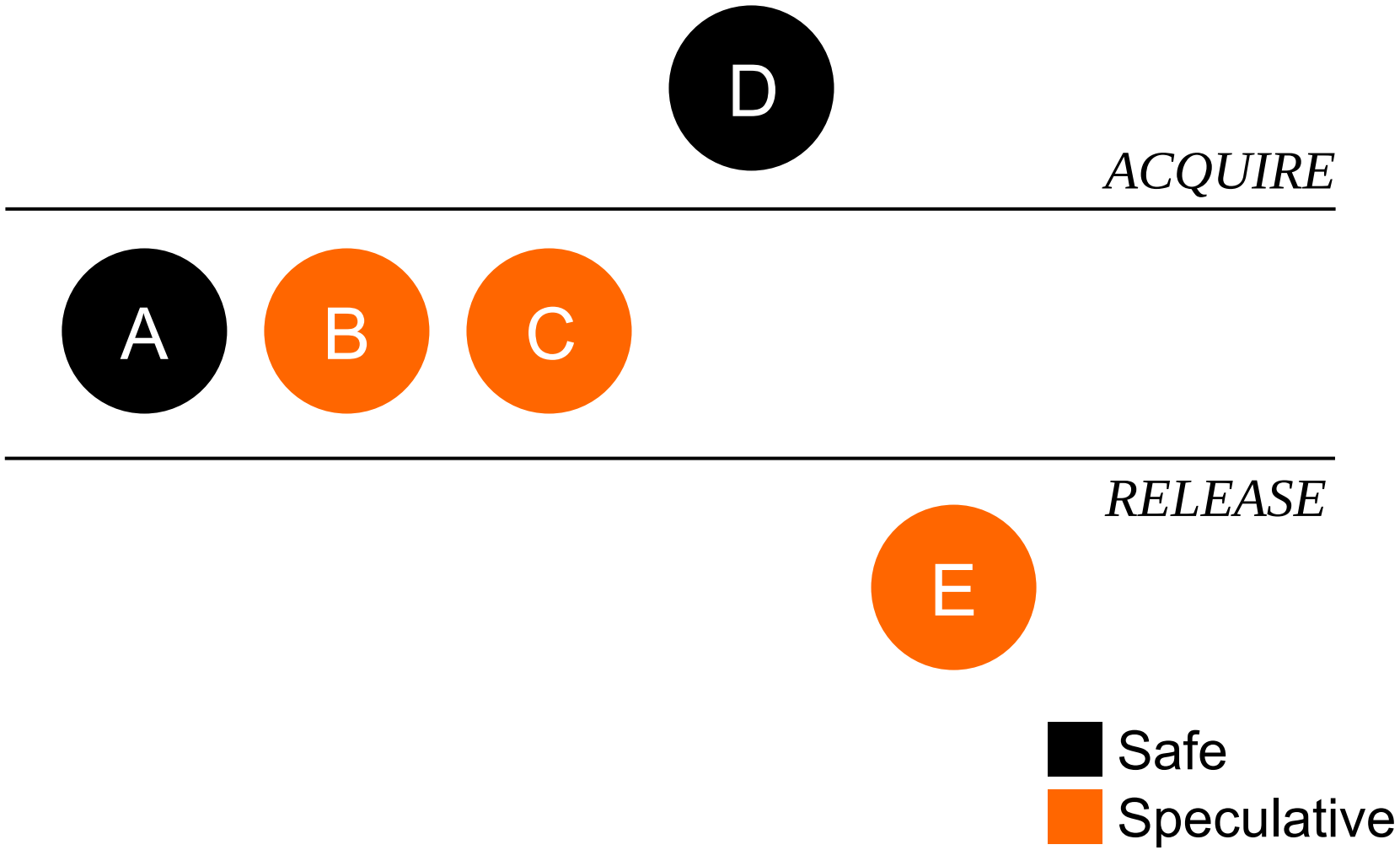
Speculative Lock



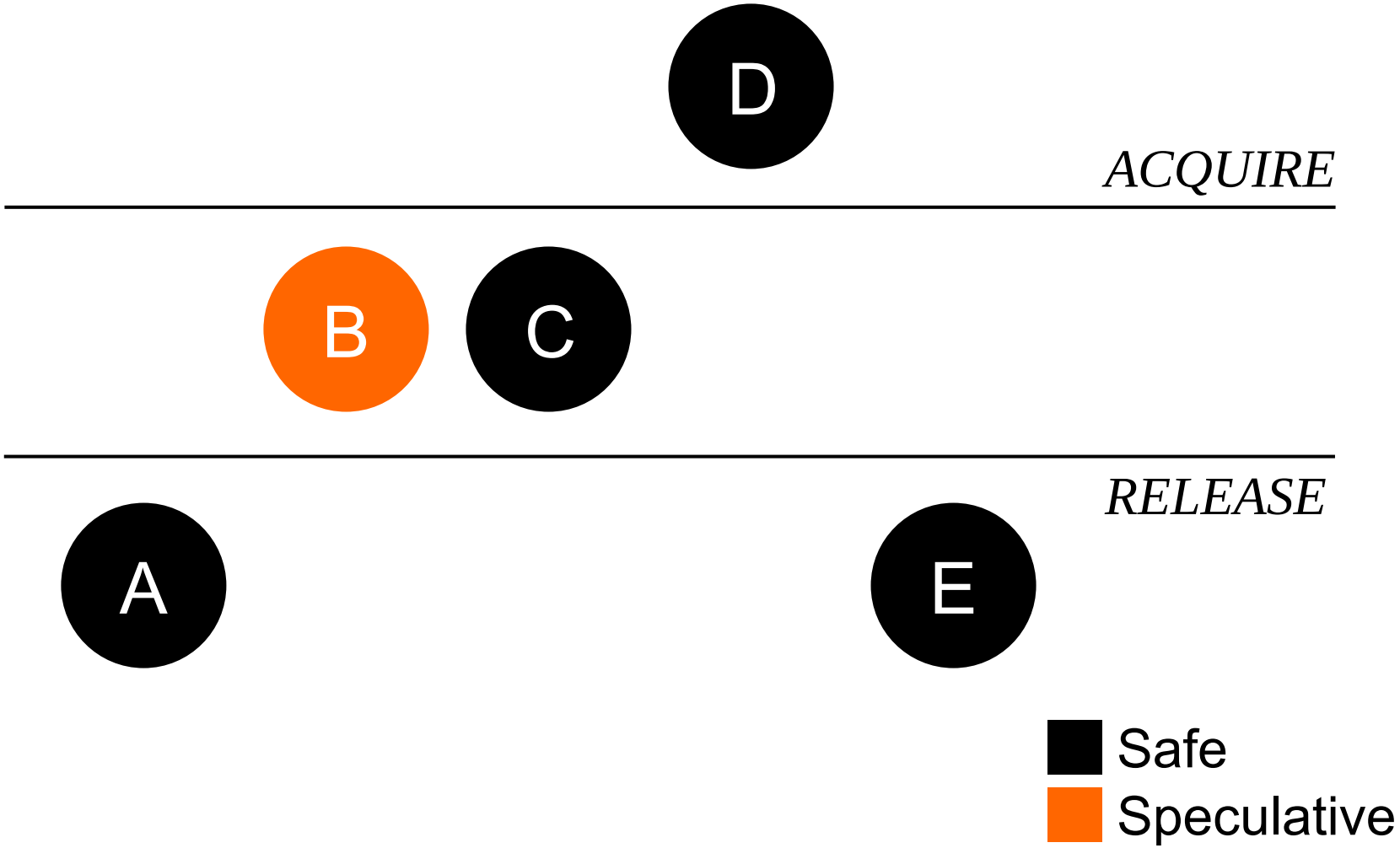
Speculative Lock



Speculative Lock



Speculative Lock



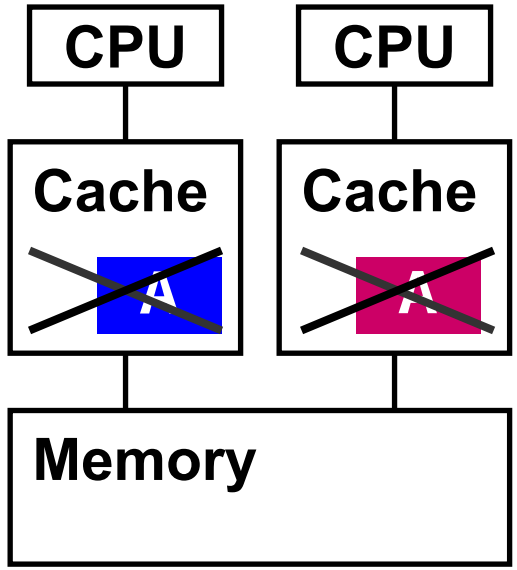
Analysis: Find Race Signature

Thread X

LD A
ST A

Thread Y

LD A
ST A



1. Rollback

2. Put a watchpoint on accesses to data address A

3. Re-execute assuming order: after

Significance: Where is the Bug?

- No need to insert invariant checks
- Find it immediately

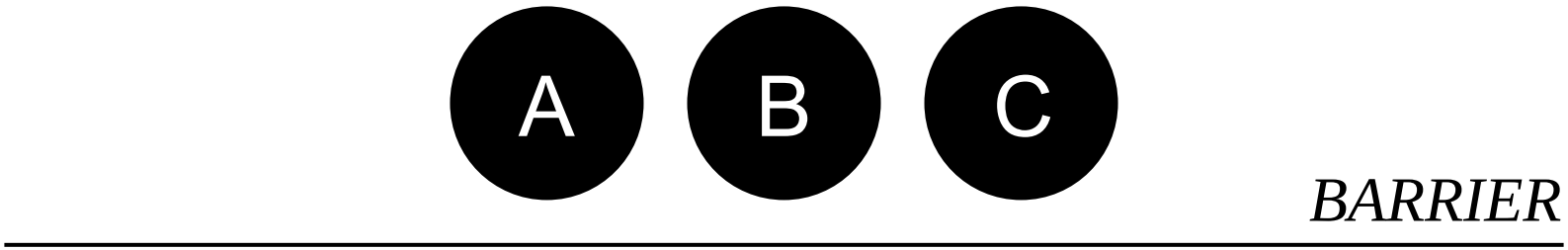
```
int x, *p;       /* invariant: x=1 or x=0*/
Watch(&x, &Monitor);
…
p = ...;   /* a bug: p points to x incorrectly */
*p = 5;   /* line A: a triggering access, bug detected with IWatcher*/
…
Assert (x==1 || x==0);    /*line B: bug detected without IWatcher */
```

```
bool Monitor (int *x) {
      return(*x==1|| *x==0)
}
```

Monitoring Functions

- Triggered by hardware
- Executed in parallel with main program by TLS (optionally)
- Can have any side-effects

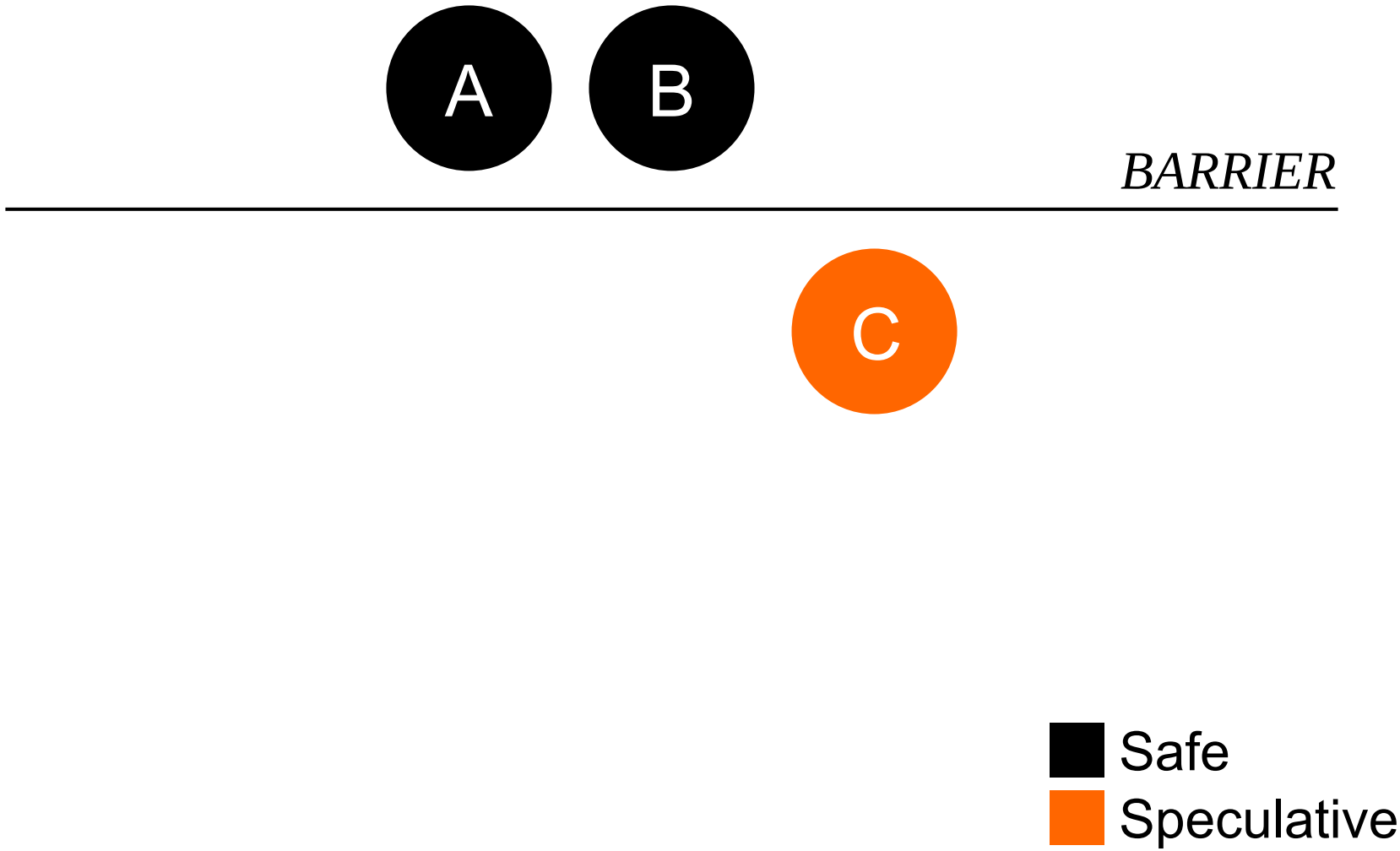
Speculative Barrier



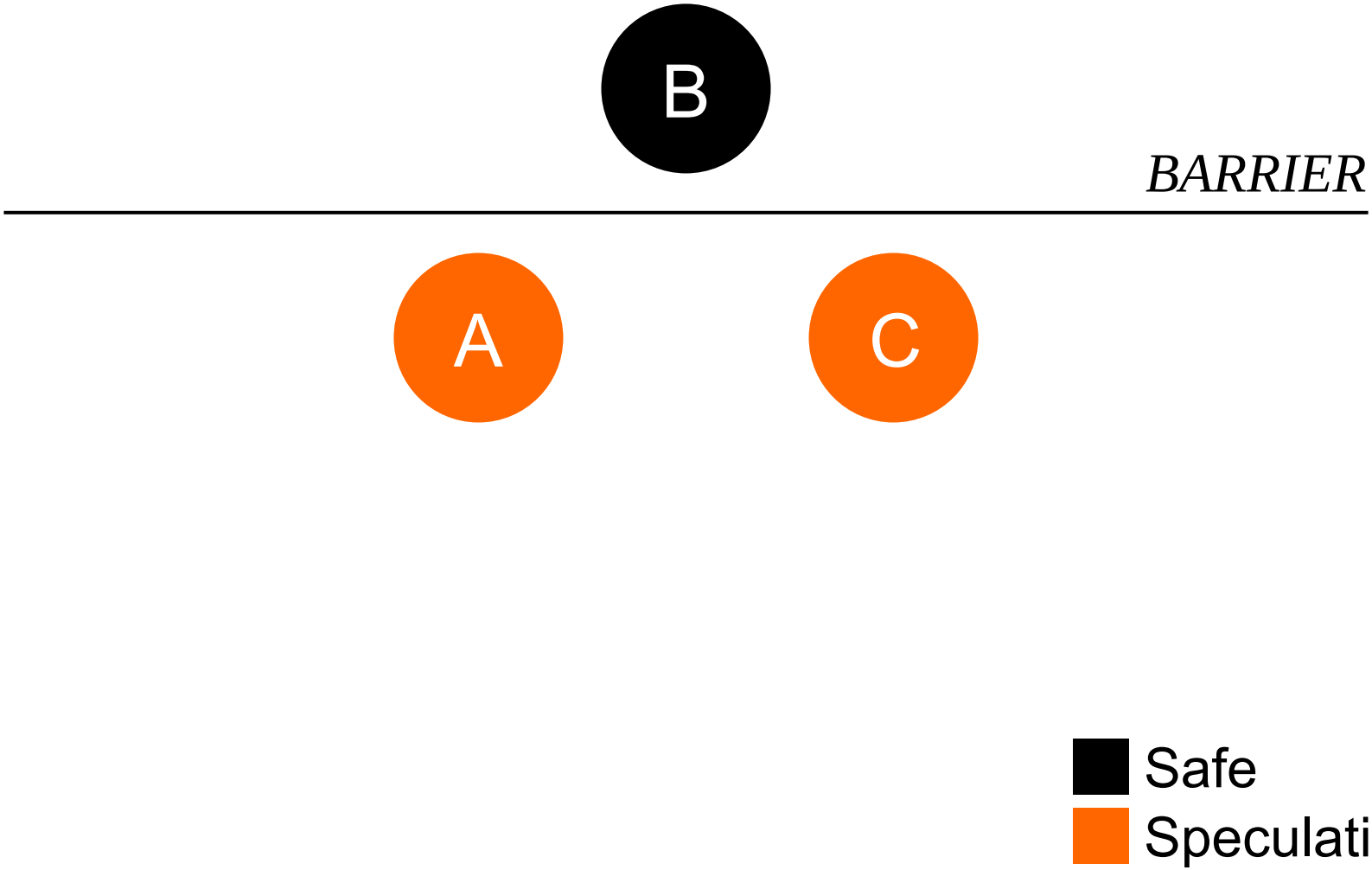
Safe

Speculative

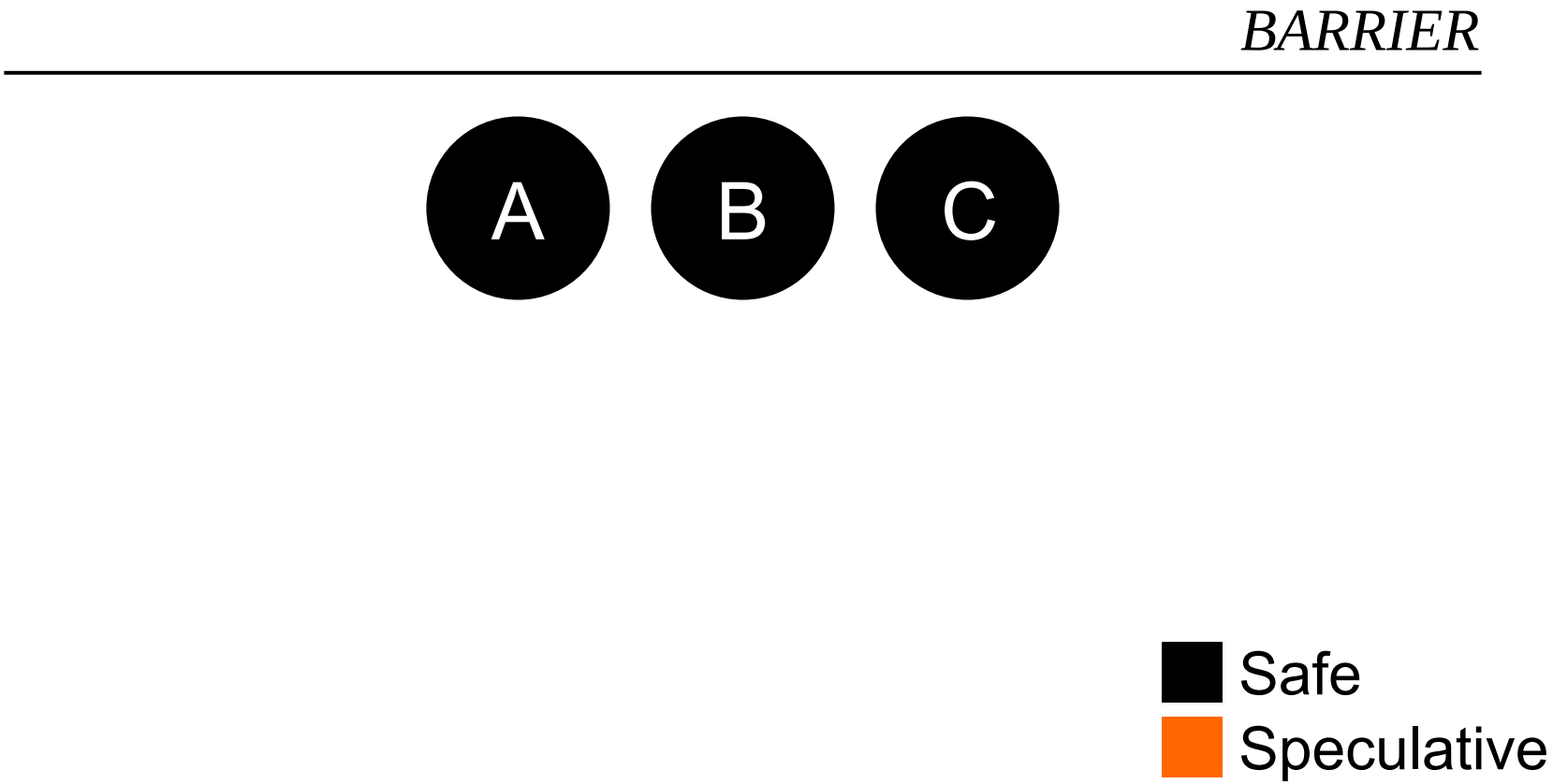
Speculative Barrier



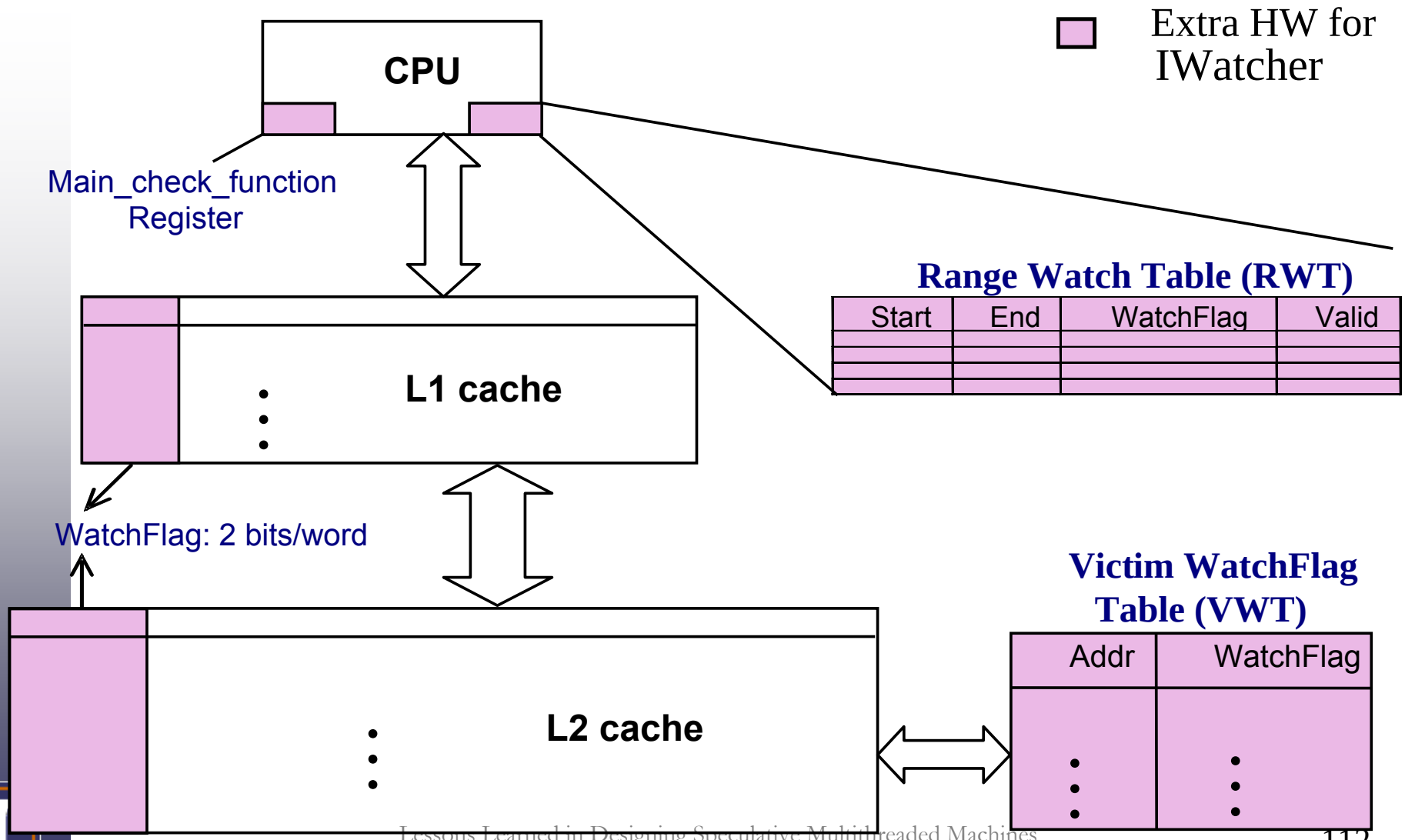
Speculative Barrier



Speculative Barrier



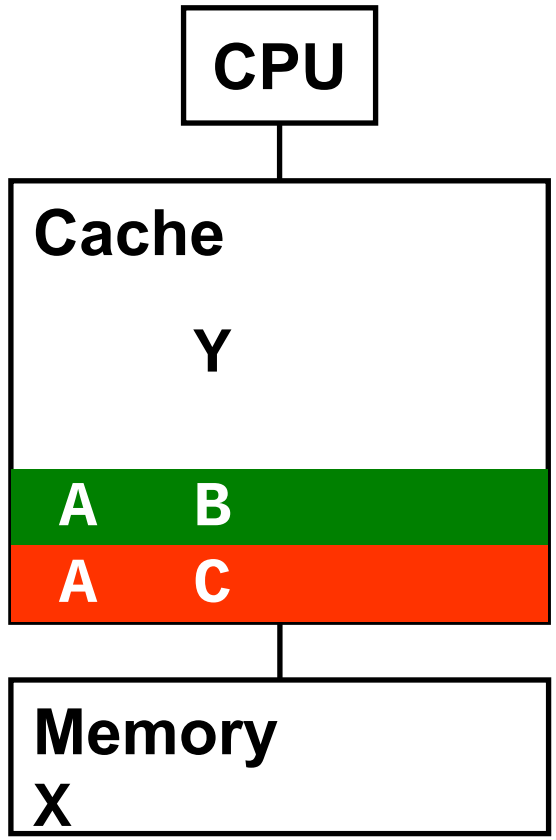
IWatcher Hardware



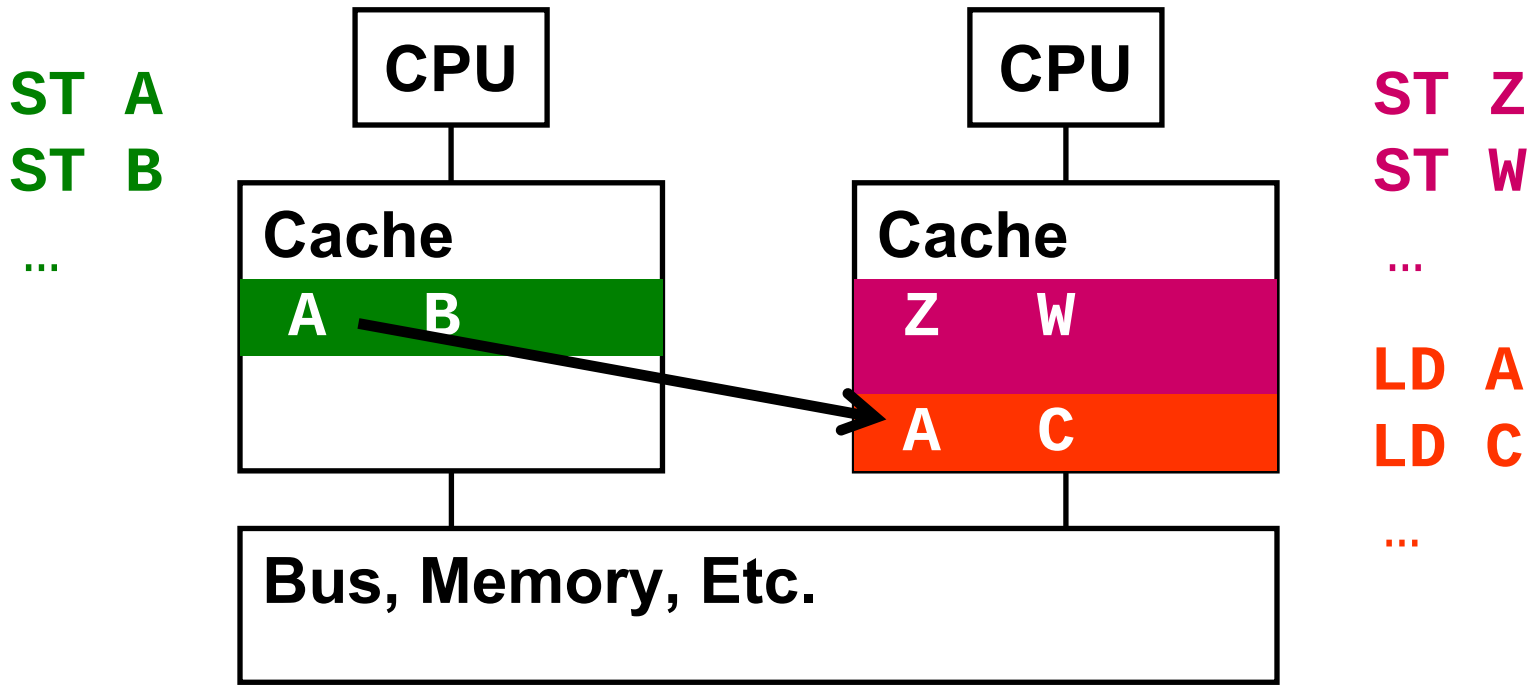
Chunk Commit

Dynamic Instructions

```
ST  X
ST  Y
ADD
 ...
ST  A
ST  B
ST  A
 ...
ST  A
ST  C
```



Ordering Chunks



Chunk ■ "happens before" chunk ■

Advantages

- On the fly – debug each problem as it is found
- Always on – usable in **production** runs
  - Low overhead in bug-free execution
- Debug multi-threaded code
  - Forces deterministic re-execution

Speculative Synchronization

- If a data collision is detected, speculative task is:
  - Squashed
  - Rolled back to the synch point

- Maintain 1 or more *safe threads* → forward progress
  - Lock: owner
  - Flag: producer
  - Barrier: lagging threads

Significance: Where is the Bug?

- No need to insert invariant checks
- Find it immediately

```
Watch(&x, &Monitor);
…
p = ...;   /* a bug: p points to x incorrectly */
*p = 5;   /* line A: a triggering access, bug detected with IWatcher*/
…
Assert (x==1 || x==0);    /*line B: bug detected without IWatcher */
```