

A Memory Model for RISC-V

Sizhuo Zhang,

Muralidaran Vijayaraghavan,

Arvind



RISC-V Workshop, November 29, 2016

Why not SC/TSO?

- ◆ They both have simple specifications, both axiomatically and operationally
- ◆ But simple implementations have low performance
 - Strict ordering requirements for memory instructions
 - To improve performance, one must monitor coherence invalidation traffic to potentially squash executed loads

Why not POWER/ARM?

- ◆ Their operational models expose too much microarchitectural details
 - Branch speculation, OOO execution, rollback etc are exposed in the memory model specification!
- ◆ Their axiomatic models are too complex with no well-understood relation to microarchitecture
 - One cannot say with confidence if a particular microarchitectural implementation obeys the model

Why not RMO?

- ◆ RMO's dependency requirements are too strict

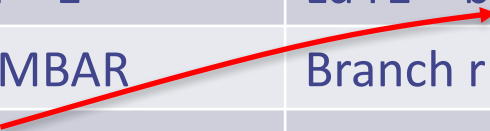
Thread 1	Thread 2
St a = 1	Ld r1 = b
MEMBAR	Branch r1 != 1 goto exit
St b = 1	St c = 1
	Ld r2 = c
	r3 = a + r2 - 1
	Ld r4 = [r3]
	exit:

Initially everything's 0

Why not RMO?

- ◆ RMO's dependency requirements are too strict

Thread 1	Thread 2
St a = 1	Ld r1 = b (1)
MEMBAR	Branch r1 != 1 goto exit
St b = 1	St c = 1
	Ld r2 = c
	r3 = a + r2 - 1
	Ld r4 = [r3]
	exit:

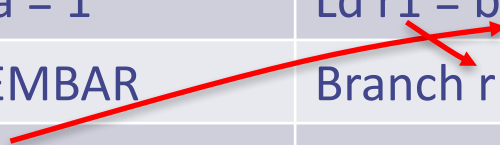


Initially everything's 0

Why not RMO?

- ◆ RMO's dependency requirements are too strict

Thread 1	Thread 2
St a = 1	Ld r1 = b (1)
MEMBAR	Branch r1 != 1 goto exit (1)
St b = 1	St c = 1
	Ld r2 = c
	r3 = a + r2 - 1
	Ld r4 = [r3]
	exit:

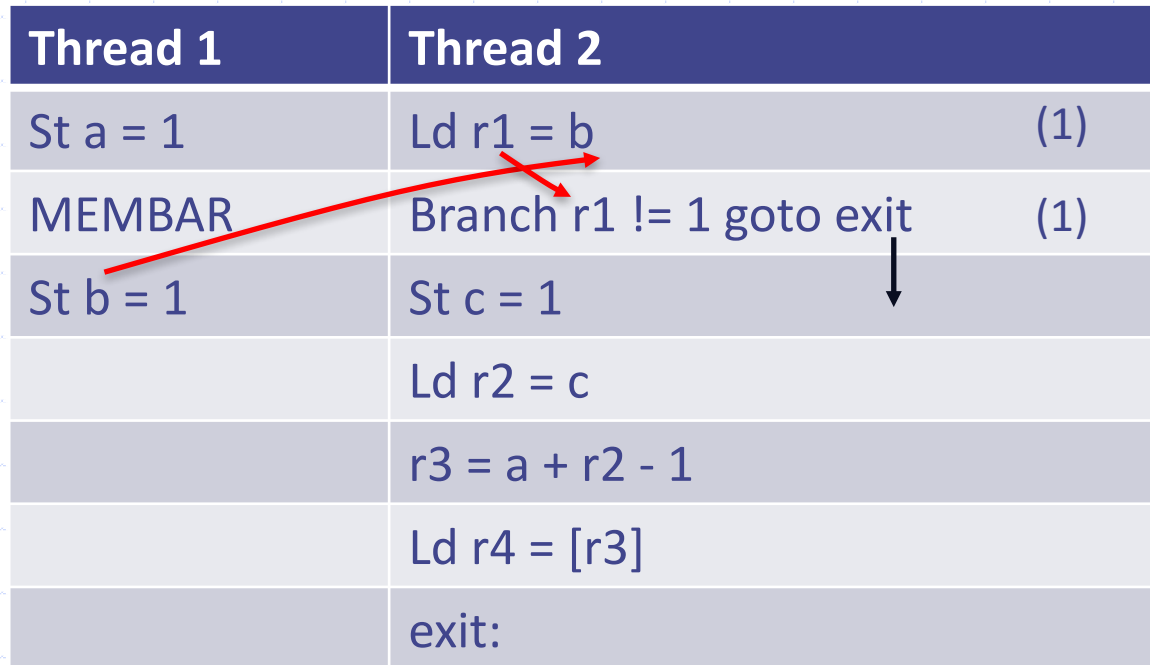


Initially everything's 0

Why not RMO?

- ◆ RMO's dependency requirements are too strict

Thread 1	Thread 2
St a = 1	Ld r1 = b (1)
MEMBAR	Branch r1 != 1 goto exit (1)
St b = 1	St c = 1
	Ld r2 = c
	r3 = a + r2 - 1
	Ld r4 = [r3]
	exit:



Initially everything's 0

Why not RMO?

- ◆ RMO's dependency requirements are too strict

Thread 1	Thread 2
St a = 1	Ld r1 = b (1)
MEMBAR	Branch r1 != 1 goto exit (1)
St b = 1	St c = 1
	Ld r2 = c (1)
	r3 = a + r2 - 1
	Ld r4 = [r3]
	exit:

Initially everything's 0

Why not RMO?

- ◆ RMO's dependency requirements are too strict

Thread 1	Thread 2
St a = 1	Ld r1 = b (1)
MEMBAR	Branch r1 != 1 goto exit (1)
St b = 1	St c = 1
	Ld r2 = c (1)
	r3 = a + r2 - 1 (a)
	Ld r4 = [r3]
	exit:

Initially everything's 0

Why not RMO?

- ◆ RMO's dependency requirements are too strict

Thread 1	Thread 2
St a = 1	Ld r1 = b (1)
MEMBAR	Branch r1 != 1 goto exit (1)
St b = 1	St c = 1
	Ld r2 = c (1)
	r3 = a + r2 - 1 (a)
	Ld r4 = [r3] (1)
	exit:

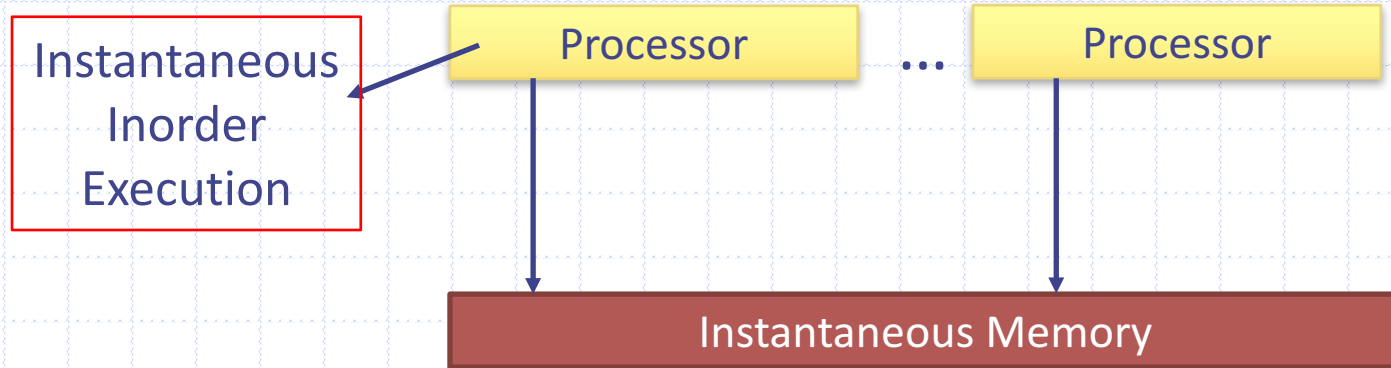
Initially everything's 0

Properties for a new memory model

- ◆ Simple specification without microarchitectural details like Branch speculation, OOO execution, rollback, etc
- ◆ But establish correspondence to microarchitecture implementations
- ◆ Weaker than SC/TSO for high performant, simple implementations
- ◆ Inclusion of sufficient fences to force SC-like behavior when necessary

Our proposal for RISC-V memory model: WMM

Simple operational specification like SC, TSO, PSO

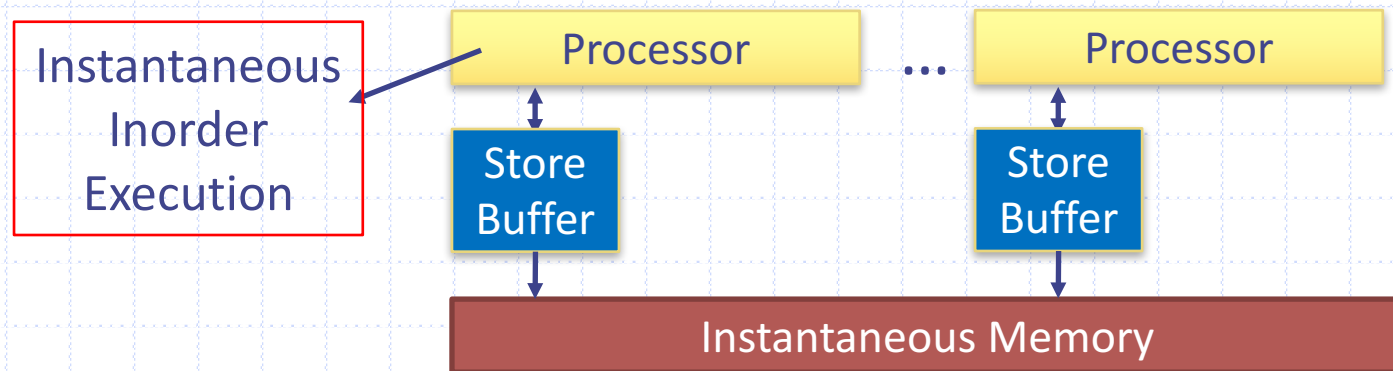


SC:

- Stores update memory instantly
- Load reads memory instantly

Our proposal for RISC-V memory model: WMM

Simple operational specification like SC, TSO, PSO

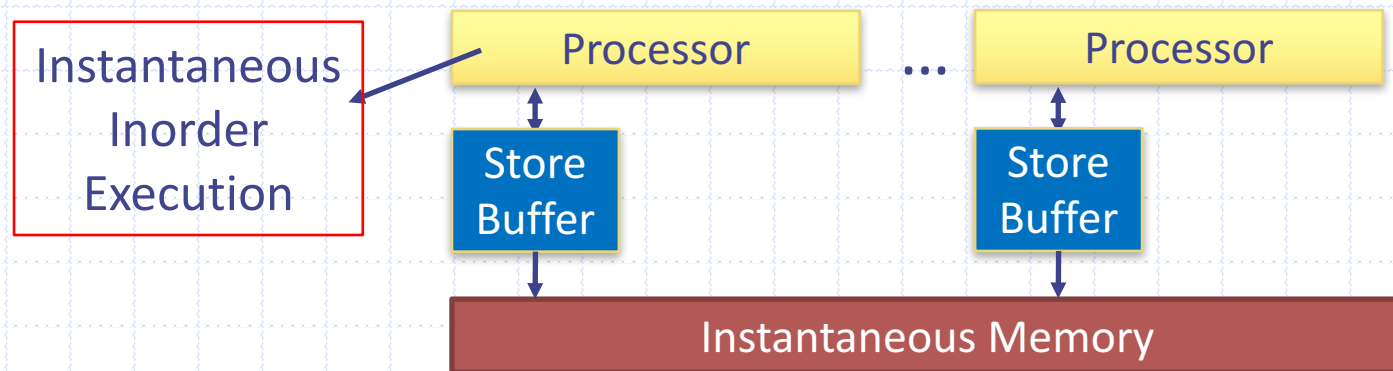


TSO:

- Stores are dequeued in order
- When a store is dequeued from store buffer, it updates memory instantly
- Load reads the youngest store from store buffer, or (if not present) memory instantly

Our proposal for RISC-V memory model: WMM

Simple operational specification like SC, TSO, PSO

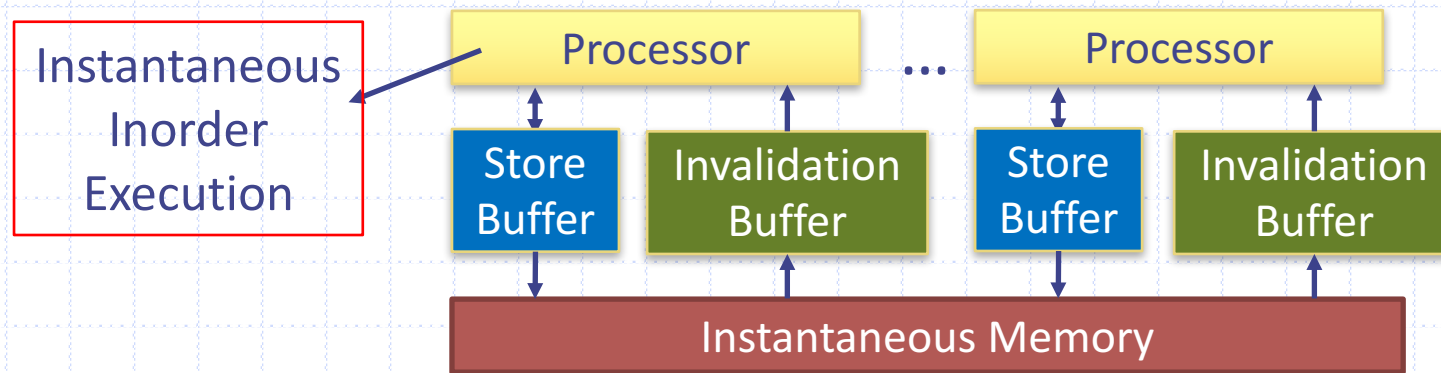


PSO:

- Stores are dequeued in order **only for same address**
- When a store is dequeued from store buffer, it updates memory instantly
- Load reads the youngest store from store buffer, or (if not present) memory instantly

Our proposal for RISC-V memory model: WMM

Simple operational specification like SC, TSO, PSO



WMM:

- Stores are dequeued in order only for same address
- When a store is dequeued from store buffer, it updates memory instantly, **removes address from own invalidation buffer and enters every other invalidation buffer instantly**
- Load reads the youngest store from store buffer, **or (if not present) oldest entry in invalidation buffer**, or (if not present) memory instantly
- **Oldest invalidation buffer entry can be thrown out any time**

Fences in WMM

- ◆ Acquire/Reconcile Fence : Clears Invalidation buffer
- ◆ Release/Commit Fence : Waits for Store buffer to be flushed (non-atomically)

Axiomatic Definition of WMM

◆ Memory order reordering axiom:

Can Reorder?		Second			
		Ld b	St b v'	Acq/Reconcile	Rel/Commit
First	Ld a	a!=b	No	No	No
	St a v	Yes	a!=b	Yes	No
	Acq/Reconcile	No	No	No	No
	Rel/Commit	Yes	No	No	No

- ## ◆ Load reads the younger (in memory order) of
- Latest store in memory order for that address OR
 - Latest store in program order (in that thread) for that address

St-St Fence: Commit

Ld-Ld Fence: Reconcile

St-Ld Fence: Commit+Reconcile

Ld-St Fence: Not needed

Implementing WMM

Formally Proven:

OOO + Single-threaded-correctness + In-order-commit
+ Value Prediction + Global Store Atomicity = WMM

- ◆ An executed load won't get squashed later as long as it doesn't overtake a reconcile or memory instruction to same address
 - No monitoring of coherence invalidations
 - Load address speculation allowed – squashed only if predicted address is wrong
- ◆ All instructions are committed in order
 - Stores cannot overtake loads
 - Prevents “out-of-thin-air” generation of values

Implementing WMM

Formally Proven:

OOO + Single-threaded-correctness + In-order-commit
+ Value Prediction + **Global Store Atomicity** = WMM

“Theoretically, the definition of the **aq** and **rl** bits allows for implementations without global store atomicity. When both **aq** and **rl** bits are set, however, we require full sequential consistency for the atomic operation which implies global store atomicity in addition to both acquire and release semantics. In practice, hardware systems are usually implemented with global store atomicity, embodied in local processor ordering rules together with single-writer cache coherence protocols.”

- ◆ Writeback coherent cache hierarchy typically satisfies Global Store Atomicity
- ◆ If L1 is write-through, easy to ensure Global Store Atomicity unless the core is SMT
 - SMT cores with L1 write-through caches implement a “non-multicopy-atomic” memory

Don't do it

Mapping C++11 to WMM

C++11	WMM
Non-atomic Load	Load
Load Relaxed	Load
Load Consume	Load; Acquire/Reconcile
Load Acquire	Load; Acquire/Reconcile
Load SC	Rel/Commit; Acq/Reconcile; Load; Acq/Reconcile
Non-atomic Store	Store
Store Relaxed	Store
Store Release	Release/Commit; Store
Store SC	Release/Commit; Store

Using operational specification of WMM makes it straightforward to derive/verify this mapping

Conclusion

- ◆ WMM is a memory model with simple specification and potentially high performant implementations
 - Blends well with RISC-V philosophy and should be used as the memory model for RISC-V

Advertisement: Formally verified RISC-V (subset of RV32I) multicore implementation in **Kami**, a hardware formal verification platform

Thank you! szzhang@mit.edu
vmurali@csail.mit.edu
arvind@csail.mit.edu



Backup



Why not RMO?

- ◆ RMO's dependency requirements are too strict

Thread 1	Thread 2
St a = 1	Ld r1 = b (1)
MEMBAR	Branch r1 != 1 goto exit (1)
St b = 1	St c = 1
	Ld r2 = c (1)
	r3 = a + r2 - 1 (a)
	Ld r4 = [r3] (1)
	exit:

Initially everything's 0

Why not Release Consistency ?

- ◆ Fences are not strong enough to give Sequential Consistency

Initially, everything is 0

Thread 1	Thread 2	Thread 3
St val = 1	Ld r1 = val (1)	Ld r2 = flag (1)
	Release	Acquire
	St flag = r1 (1)	Ld r3 = val (0)

Non-cumulative Fences

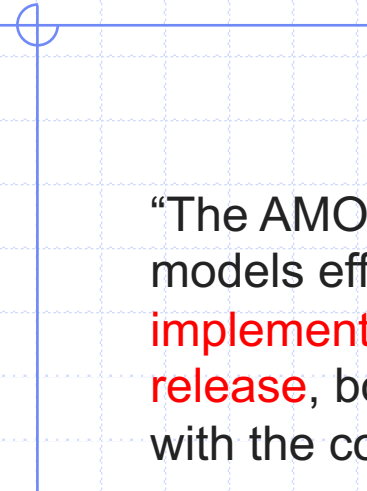
Out-of-thin-air issue

Thread 1	Thread 2
Ld r1 = x	Ld r2 = y
St y = R1	St x = 42

Initially everything is 0

Finally $x = y = r1 = r2 = 42$

- ◆ No processor can produce values out of thin air
 - But incomplete set of axioms seemingly allows this
- ◆ Insisting on in-order commits and advertising stores only after commit to other threads/processes takes care of this issue



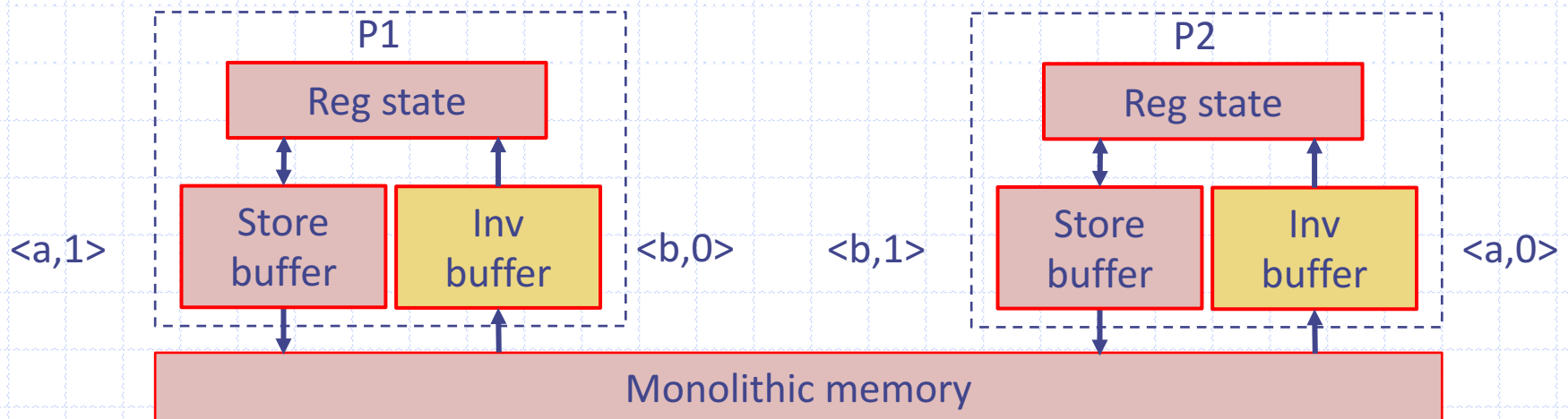
“The AMOs were designed to implement the C11 and C++11 memory models efficiently. Although the **FENCE R, RW** instruction suffices to **implement the acquire operation** and **FENCE RW, W** suffices to **implement release**, both imply additional unnecessary ordering as compared to AMOs with the corresponding aq or rl bit set.”

Litmus Tests for WMM

Test SB	
P1	P2
I1: St a 1 I2: Commit Reconcile I3: r1 = Ld b	I4: St b 1 I5: Commit Reconcile I6: r2 = Ld a
WMM allows: r1=0, r2=0	

WMM allows the behavior
- Ld overtakes St and Commit

Add Reconcile to forbid this



Litmus Tests for WMM

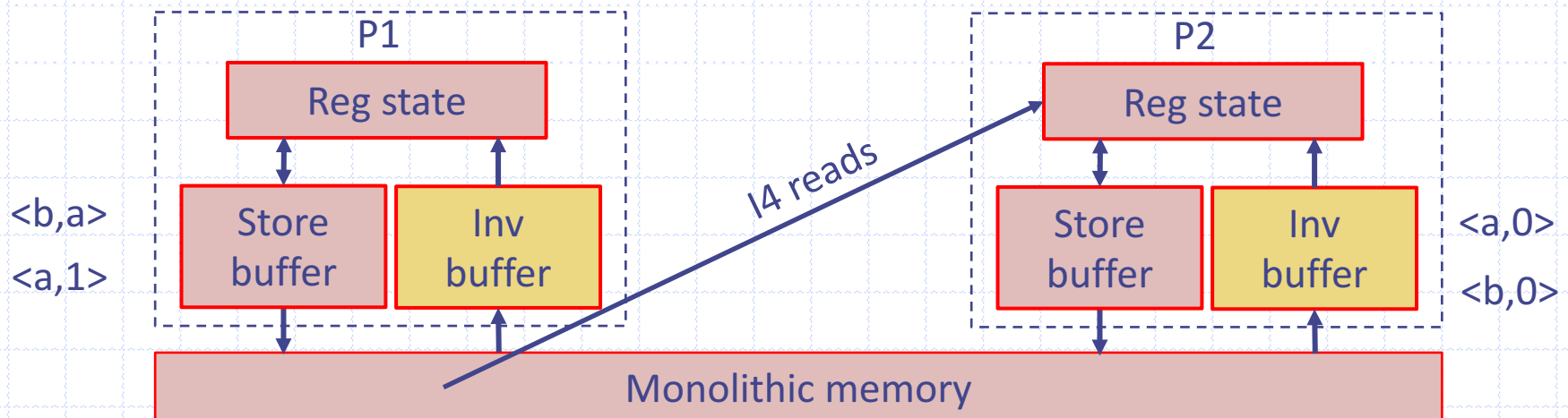
Test MP+data	
P1	P2
I1: St a 1 I2: Commit I3: St b a	I4: r1 = Ld b Reconcile I5: r2 = Ld r1
WMM allows: r1=a, r2=0	

WMM allows the behavior

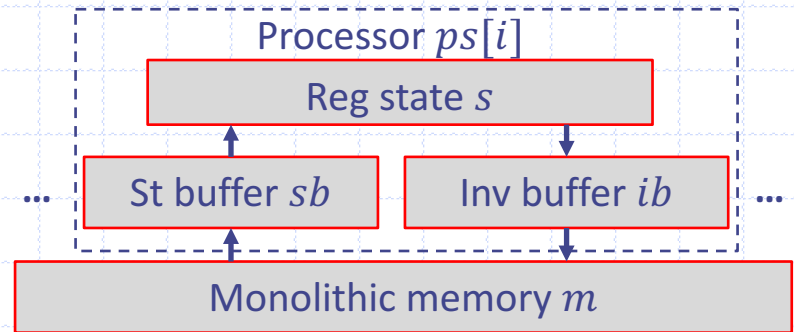
- Ld overtakes Ld
- No dependency ordering
- Can be caused by value prediction in hardware

Add Reconcile to forbid this

Out-of-thin-air is impossible because of I²E



WMM-S



- ◆ The same abstract machine structure as WMM
- ◆ Model non-multi-copy-atomic stores
 - Make a store from processor i visible to processor j before the store updates monolithic memory
 - Make a copy of the store from the sb of processor i , and insert the copy into the sb of processor j
 - Each store has a unique tag, copies have the same tag
- ◆ Dequeue a store from sb to monolithic memory
 - All copies are dequeued from sb
 - All copies have to be the oldest one for the store address in their respective sb
- ◆ Copying of a must be constrained for per-location SC
 - Each sb orders stores for a certain address as a list
 - Combining all such lists from all sb together forms a partial coherence order ($<_{co}$) of the store tags for that address
 - After copying, partial coherence order must be still acyclic

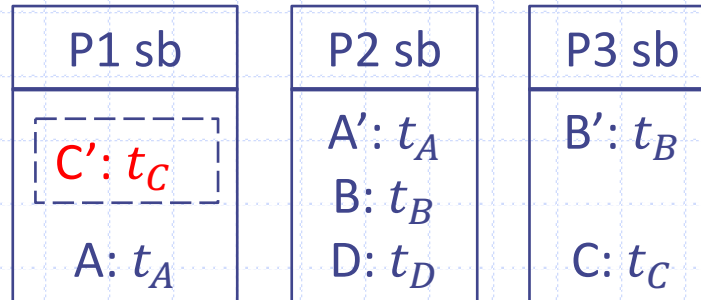
Store copy example

(Primes are copies)

Inserted later (younger)



Inserted earlier (older)



◆ Current partial coherence order

- $t_D <_{co} t_B <_{co} t_A$ and $t_C <_{co} t_B$
- t_D and t_C are unrelated

◆ If we copy C into sb of P1 as C'

- Create cycle: $t_A <_{co} t_C <_{co} t_B <_{co} t_A$
- Should not be allowed

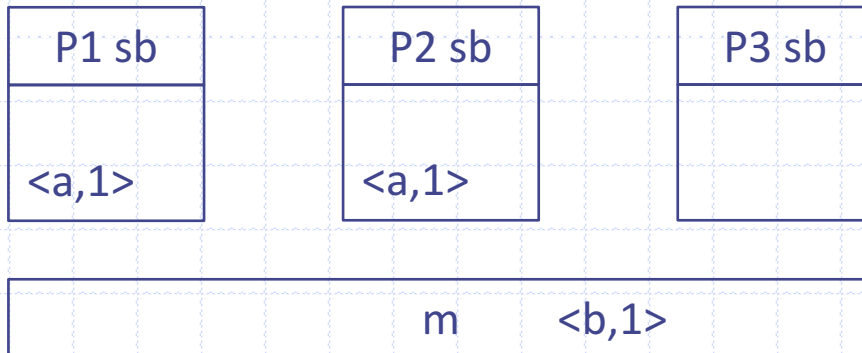
◆ If we copy A into sb of P2

- Create cycle: $t_A <_{co} t_A$

Litmus Tests for WMM-S

Test WRC		
P1	P2	P3
I1: St a 1	I2: r1 = Ld a Commit I3: St b r1	I4: r2 = Ld b I5: Reconcile I6: r3 = Ld a
WMM-S allows: r1=1, r2=1, r3=0		

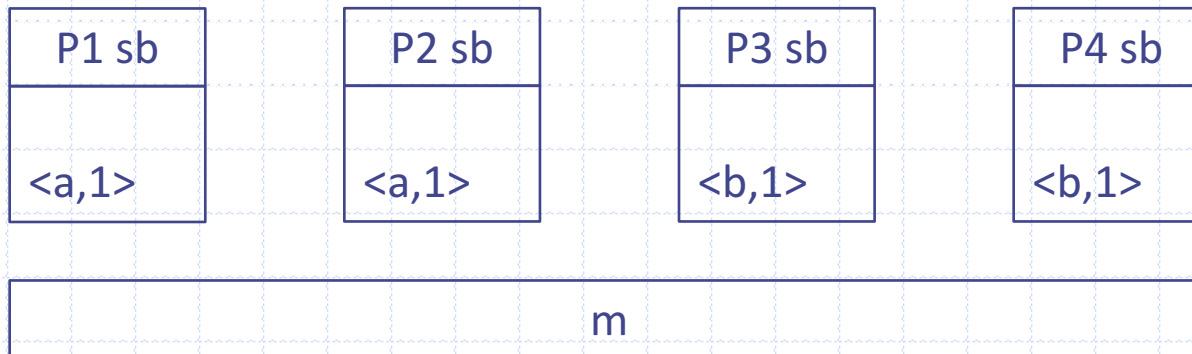
- Add Commit in P2 to forbid this behavior
- Commit globally advertises observed stores -- release
- Reconcile prevents loads from reading stale values -- acquire



Litmus Tests for WMM-S

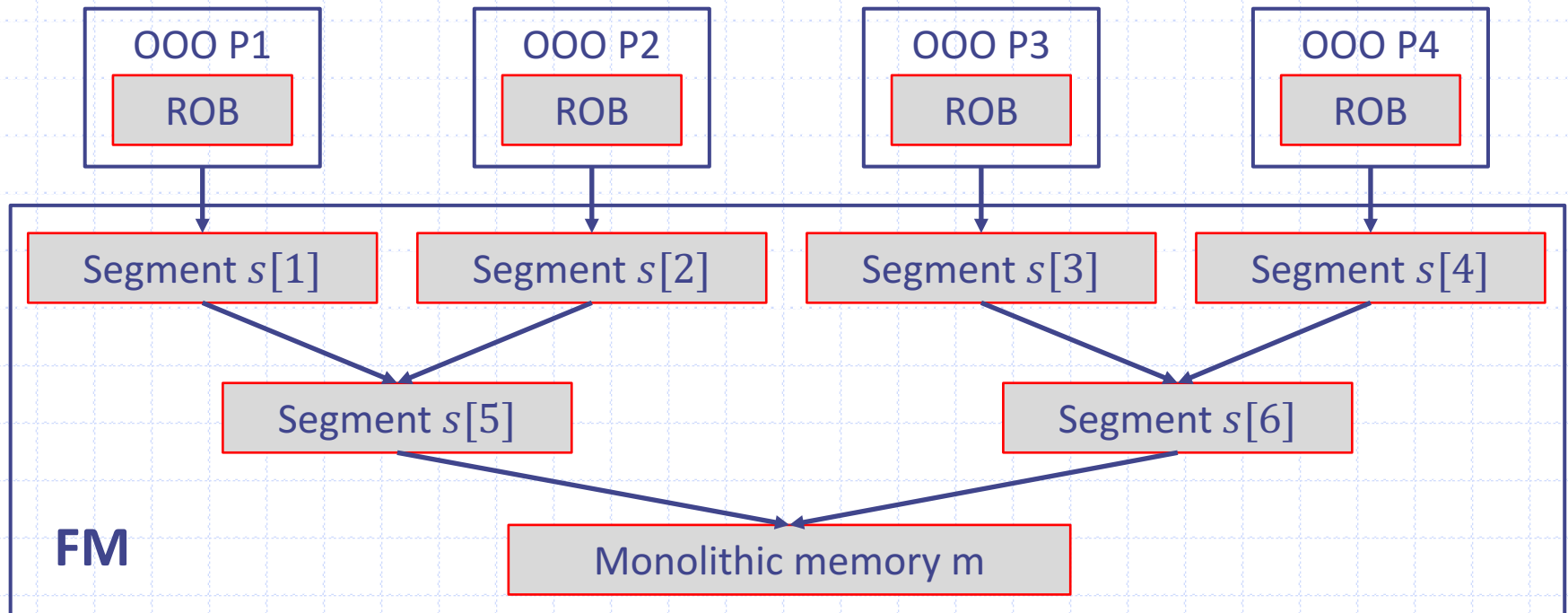
Test IRIW

P1	P2	P3	P4
I1: St a 1	I2: r1 = Ld a Commit I3: Reconcile I4: r2 = Ld b	I5: St b 1	I6: r3 = Ld b Commit I7: Reconcile I8: r4 = Ld a
WMM-S allows: r1=1, r2=0, r3=1, r4=1			



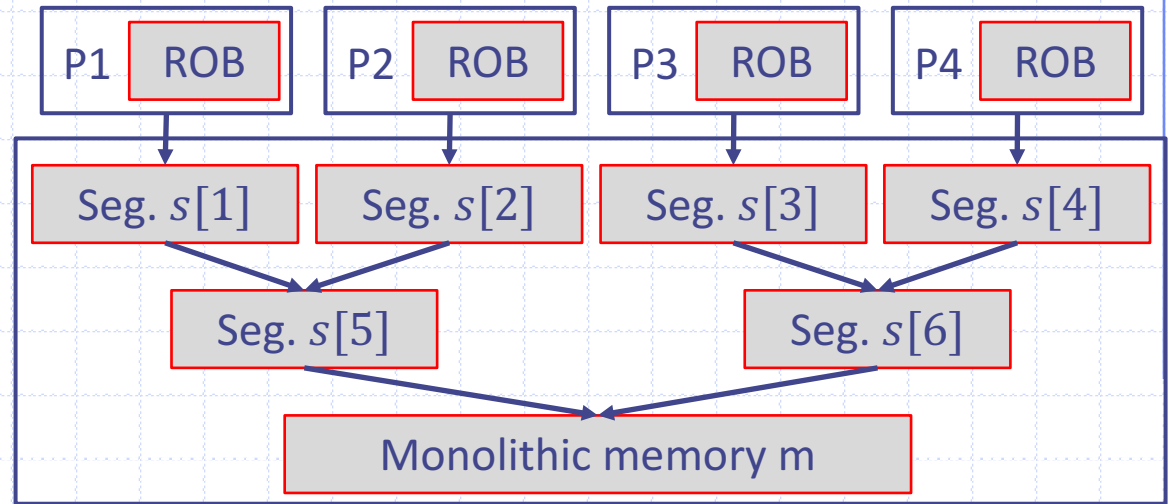
WMM-S Implementation

- ◆ WMM-S can be implemented using OOO + non-atomic memory system
 - e.g. memory system of the ARM Flowing Model (FM) ^[1]
 - We do not need store buffer in OOO, because FM has buffers



[1] Flur et al. "Modelling the ARMv8 architecture, operationally: concurrency and ISA", POPL 2016

FM+OOO



◆ Simplified version of FM (no fence in FM)

◆ Each segment is a buffer of memory requests

- Keeps FIFO ordering of requests to the same address
- Flow rule: The oldest request for some address in a segment can be moved to the parent segment or monolithic memory
- Bypass rule: A store can forward its data to a load, as long as there is no other request to the same address in between

◆ OOO commit

- store: directly insert into segment
- Commit fence: if any segment contains a store observed by the commits of the OOO processor, then we cannot commit the fence

A store observed by commits of P_i : either committed by P_i or returned by a load committed by P_i

CCM+OOO \subseteq WMM

FM+OOO \subseteq WMM-S

- ◆ How WMM/WMM-S simulates CCM/FM+OOO
 - When the monolithic memory in CCM/FM is updated by a store
 - ◆ WMM/WMM-S dequeues that store from *sb* to monolithic memory
 - When OOO commits an instruction
 - ◆ WMM/WMM-S executes that instruction
- ◆ When OOO P_i commits a load L for address a with result v
 - Consider where is v in CCM/FM+OOO when L commits
 - v is in monolithic memory of CCM
 - ◆ WMM executes L by reading monolithic memory
 - v has been overwritten by another store in monolithic memory
 - ◆ WMM has previously inserted $\langle a, v \rangle$ in to *ib* of $ps[i]$
 - ◆ Now WMM can execute L by reading *ib*
 - v is in store buffer of OOO P_i
 - ◆ If v has been observed by commits of P_i before L is committed, then WMM/WMM-S can execute L by reading local *sb*
 - ◆ Otherwise, WMM-S fires copy $\langle a, v \rangle$ into local *sb* and let L read it

Impact of Disallowing Ld-St Reordering

◆ Qualitative analysis

- Store buffer can already hide the store miss latency
- Stores are not on the critical path for single-thread performance
- In extreme cases, the speculative store queue may be filled up with uncommitted stores

◆ Quantitative evaluation

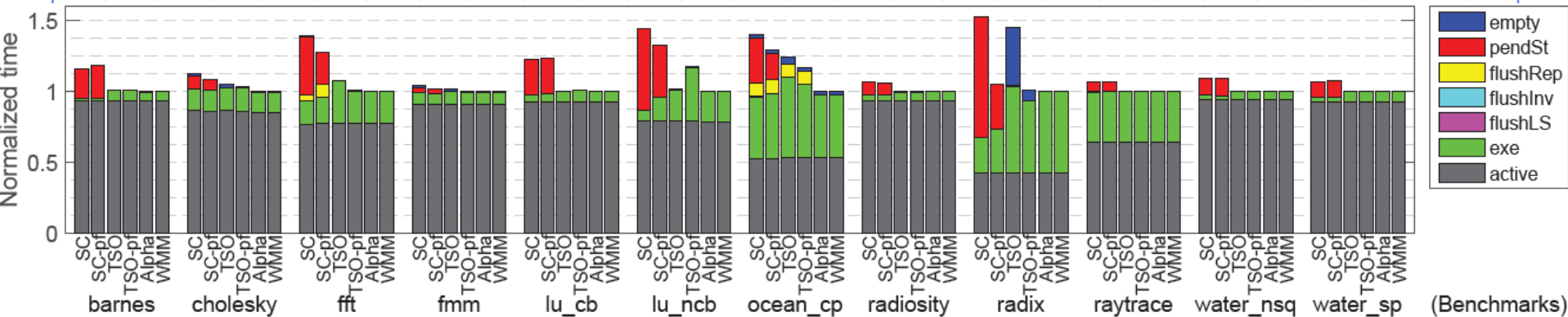
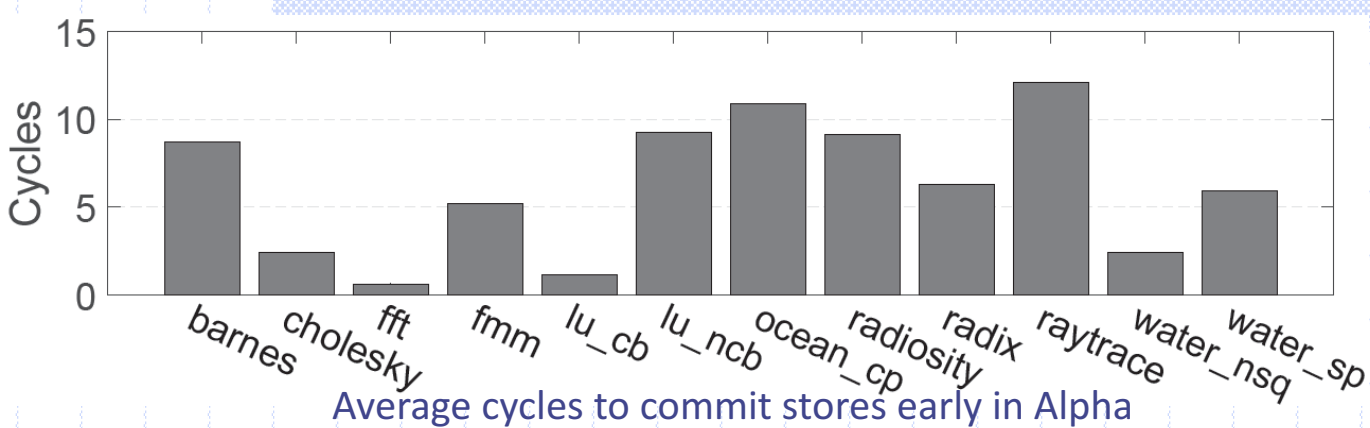
- Simulate 8-core multiprocessor using ESESC simulator
- Run SPLASH2x benchmarks
- Compare WMM, Alpha, and aggressive implementations of SC and TSO
- Alpha = WMM + Ld-St reordering
 - ◆ Try to find younger stores to commit when the instruction at the commit slot of the ROB cannot commit

Simulation Configuration

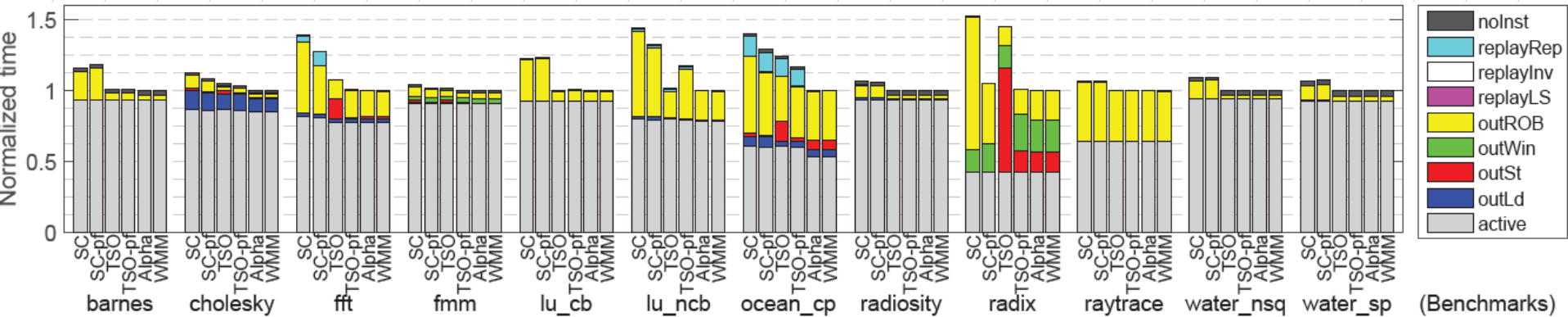
Cores	8 cores (@2GHz) with private L1 and L2 caches
L3 cache	4MB shared, MESI coherence, 64-byte cache line 8 banks, 16-way, LRU replacement, max 32 req per bank 3-cycle tag, 10-cycle data (both pipelined) 5 cycles between cache bank and core (pipelined)
Memory	120-cycle latency, max 24 requests

Frontend	fetch + decode + rename, 7-cycle pipelined latency in all 2-way superscalar, hybrid branch predictor
ROB	128 entries, 2-way issue/commit
Function units	2 ALUs, 1 FPU, 1 branch unit, 1 load unit, 1 store unit 32-entry reservation station per unit
Ld queue	Max 32 loads
St queue	Max 24 stores, containing speculative and committed stores
Store set	8192-entry store set ID table (SSIT) 256-entry last fetched store table (LFST)
L1 D cache	32KB private, 1 bank, 4-way, 64-byte cache line LRU replacement, 1-cycle tag, 2-cycle data (pipelined) Max 32 upgrade and 8 downgrade requests
L2 cache	128KB private, 1 bank, 8-way, 64-byte cache line LRU replacement, 2-cycle tag, 6-cycle data (both pipelined) Max 32 upgrade and 8 downgrade requests

Results



Normalized execution time and its breakdown at the commit slot of ROB



Normalized execution time and its breakdown at the issue port to ROB

Non-Atomic Memory

- ◆ Models for non-atomic memory is more complicated
- ◆ We are unclear about the performance advantage of non-atomic memory
 - Because our understanding of the microarchitectural sources for non-atomic memory is limited
 - POWER: shared write-through L1 due to SMT
 - ◆ Other sources in the hierarchy starting from L2?
 - ARM: no clue
 - ◆ Many litmus tests for non-atomic stores are not observable on hardware
 - ◆ WRC+addrs, WWC+addrs, IRIW+addrs (<http://diy.inria.fr/cats/model-arm/all.html>)
 - ◆ WRC+addrs (<http://www.cl.cam.ac.uk/~sf502/pop16/observations.pdf>)
- ◆ Only by understanding the microarchitectural reasons for non-atomic memory, are we able to analyze the benefit of it