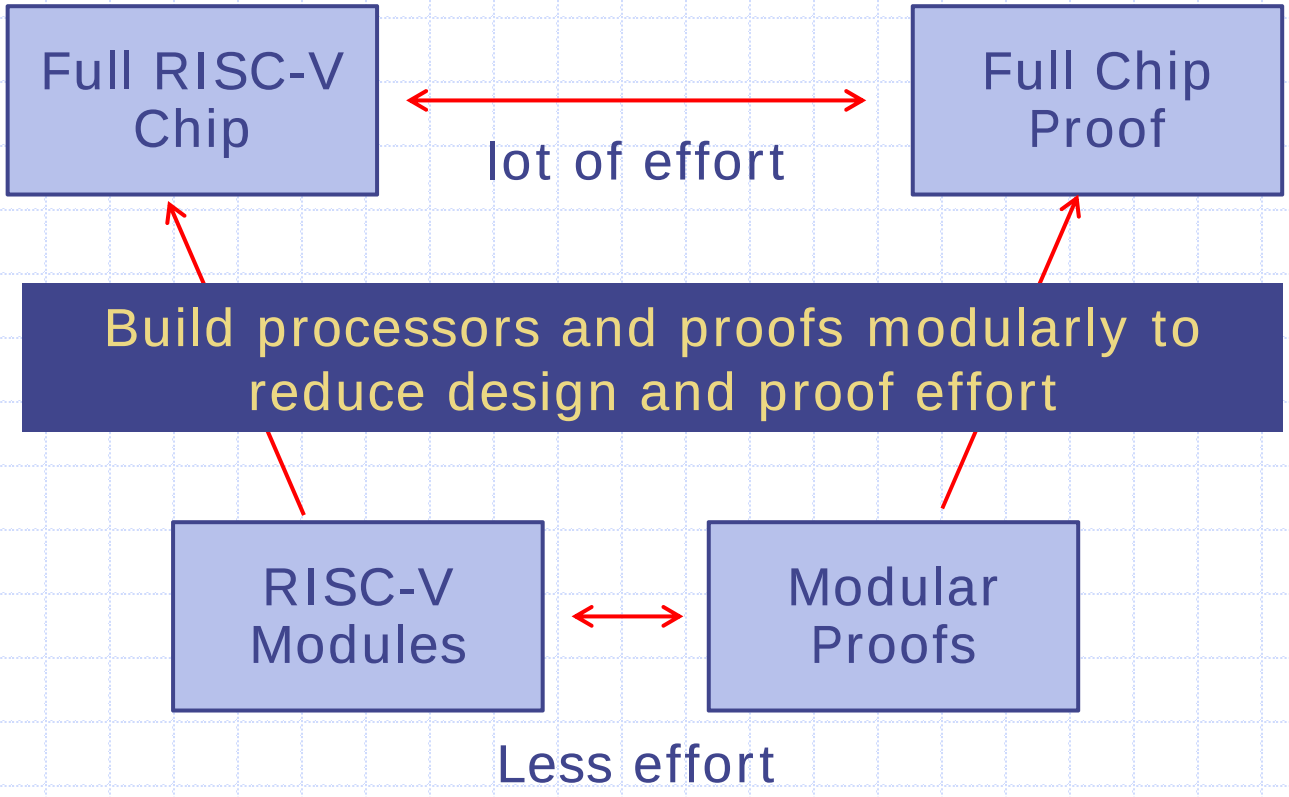
A Memory Model for RISC-V

Arvind *(joint work with Sizhuo Zhang and Muralidaran Vijayaraghavan)*
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Barcelona Supercomputer Center, Barcelona
June 27, 2017

MIT's Riscy Expedition: Chips with Proofs with *Adam Chlipala*



Andy Wright

Sizhuo Zhang

Thomas Bourgeat

Joonwon Choi

Murali Vijayaraghavan

Jamey Hicks

Current Riscy Offerings

www.github.com/csail-csg/riscy

- Building Blocks for Processor Design:
  - Riscy Processor Library
  - Riscy BSV Utility Library
- Reference Processor Implementations:
  - Multicycle
  - In-Order Pipelined
  - Out-of-Order Execution
- Infrastructure:
  - Connectal
  - Tandem Verification

One low-power RISC-V chip with security accelerators for IOT applications had been taped out (with Chandrakasan)

A flexible way of designing processors leveraging Bluespec System Verilog (BSV)

Plan

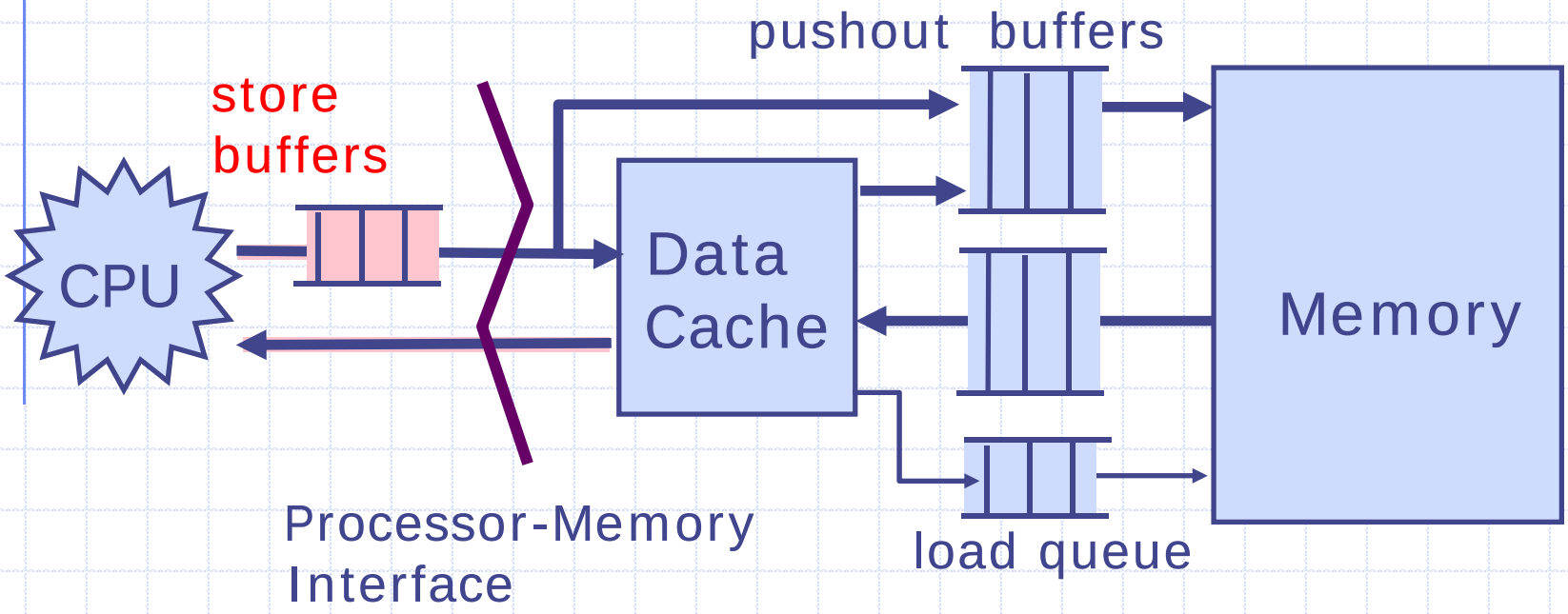
- What is the memory model debate about?
- Two weak-memory model proposals for RISC-V

General Observations

- Memory models in use were never designed – they "emerged" when people started building shared memory machines
  - IBM 370, SUN, Intel, ARM, ...
- "Emerged": Just about every correct and popular microarchitectural and compiler optimization becomes (programmatically) visible in a multiprocessor setting
- A memory-model specifies which program behaviors are legal and which are not

Goal: Specify a memory model for RISC-V to guide architects and programmers

Optimizations & Memory Models



Architectural optimizations that are correct for uniprocessors, often violate SC and result in a new memory model for multiprocessors

Example: Store Buffers

Process 1	*Process 2*
Store(x,1);	Store(flag,1);
r_1 := Load(flag);	r_2 := Load(x);

Suppose Loads can bypass stores in the store buffer

Is it possible that both r_1 and r_2 are 0 simultaneously?

Not possible in SC but allowed in the TSO memory model (IBM 370, Sparc's TSO, Intel)

Initially, all memory locations contain zeros

Memory Fence Instructions

- A programmer needs instructions to prevent undesirable Load-Store reorderings
  - Intel : MFENCE; Sparc: MEMBAR, ...
  - Meaning - All instructions before the fence must be completed before any instruction after the fence is executed

What does it mean for a store instruction to be completed?

Insertion of fences is a significant burden for the programmer and compiler writer

A hack in IBM 370 ISA

Process 1	*Process 2*
Store(x,1);	Store(flag,1);
r_3 := Load(x);	r_4 := Load(flag);
r_1 := Load(flag);	r_2 := Load(x);

IBM 370 did not want to change the instruction set – so they stipulated that a load immediately preceded by a store will act as a barrier

The meaning of the program will change if the middle (dead) load is deleted by an optimizer!

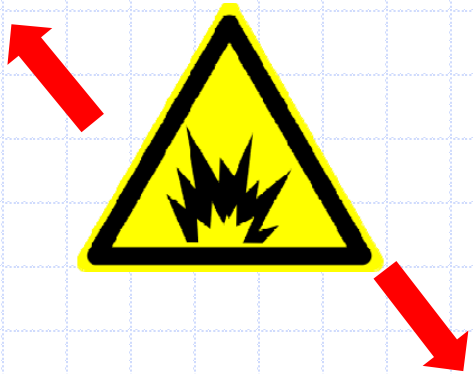
There were several such hacks

Memory Model Landscape

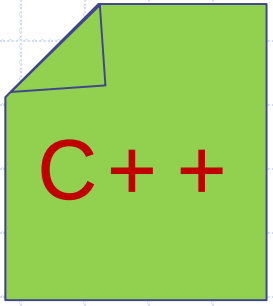
- Sequential Consistency (SC)
  - Easy to understand and formalize; no fences
  - All parallel programming is built on SC foundations
  - No ISA supports it exclusively
- Total Store Order (TSO)
  - Loads can jump over stores; operationally can be explained in terms of Store buffers
  - Easy to understand and formalize; one fence
  - Intel ISA supports it ⇒ lots of legacy code
- Weaker memory models
  - RMO, RC, Alpha, POWER, ARM, ...
  - No two models agree with each other
  - Experts don't agree on definitions

Weak Memory Models

Architects find SC & TSO constraining



Programmers hate weak memory models



Different Viewpoints

- Architects: Out-of-order and speculative execution is the backbone of modern processors
  - ⇒ Results in reordering of loads and stores
  - ⇒ Extra hardware to detect SC/TSO violations
  - ⇒ Not all violations affect program correctness
- Programmers: Difficult to understand, implementation-driven weak memory models ARM, POWER, RMO, Alpha, etc.
  - Insertion of model-dependent fences difficult
    - Extra fences ⇒ bad performance
    - Too few ⇒ errors (often latent); undesirable behaviors
  - Automatic insertion of minimal number of fences is impossible

Definitions are awful

POWER ***sync*** fence: Any access in group *A* (instructions before the fence in P1) are performed with respect to any processor before any access in group *B* (instructions after the fence in P1).
The fence is *cumulative* and it implies:

- Group *A* also includes all accesses by any processor that have been performed *w.r.t.* P1 before the fence is executed
- Group *B* also includes all accesses by any processor that are performed after a load executed by that processor has returned the value of a store in *B*.

Weak Memory Model Debate

- The subtleties cannot be handled without formalisms – informal natural language descriptions in the manuals just won't do
- In the last 10 years researchers with training in formal methods have jumped into the fry, mostly from outside the architecture community
  - Architects are gasping…
  - Formal people often do not understand what is implementable
  - Too much reliance on *litmus tests*

Current practice

- Develop an axiomatic model based on informal company documentation and empirical observations to determine allowed and disallowed behaviors
- Summarize observations as a set of *litmus tests*, each test is a multithreaded program
  - 2 to 4 threads, small straight-line codes (2 to 6 instructions)
- Use formal tools (mostly model checking) to show if a multithreaded program with fences shows only legal behaviors

RISC-V Memory Model debate

- Stick to TSO
  - The programming community loves it
  - Most architects barf at the idea because they think they will lose performance
- Adopt a cleaned up weak memory model
  - Specify via a “simple” axiomatic model
  - Specify via a “simple” operational model
  - The two definitions must match
  - Don’t restrict implementations

Requires research!

Performance issues

- Naïve viewpoint: If a memory model does not allow a particular instruction reordering then the microarchitecture cannot do it
  - demonstrably false, look at Intel implementations
- Fact 1: In-order pipelines
  - No instruction reordering ⇒ No memory model issues
- Fact 2: All modern OOO pipelines are similar
  - ROB, store buffers, cache hierarchies, …
  - Rely on speculation machinery to squash unwanted memory behaviors

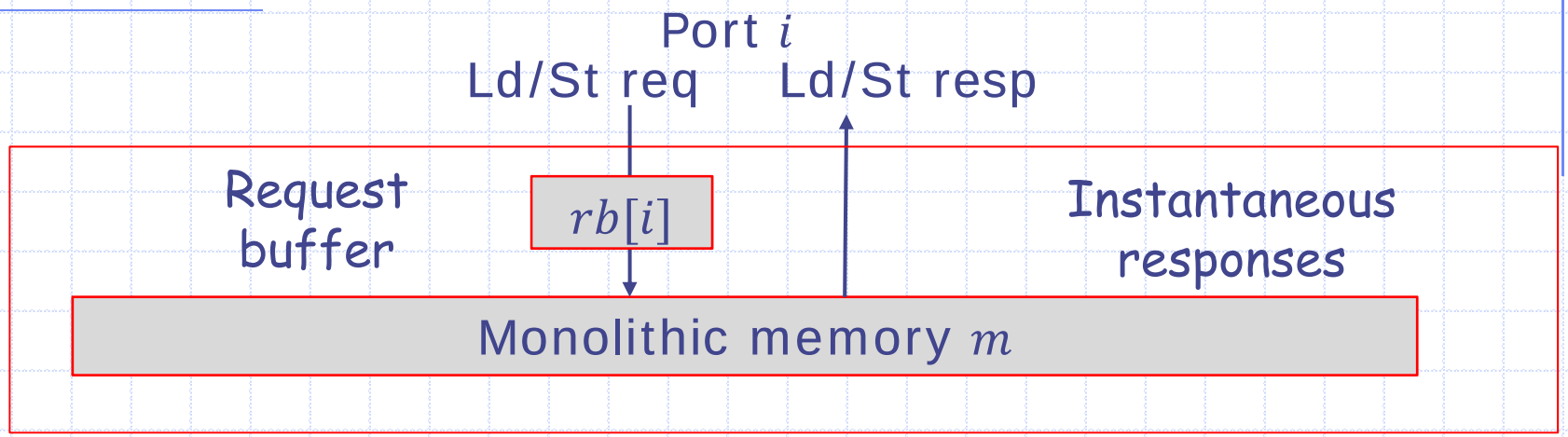
No proper studies exist to show the advantage of weak memory models or the hardware overhead of preserving TSO

Weak memory models: Technical issues

- Atomic vs Non-Atomic memory subsystems
- Should Load-Store reordering, i.e., a store is allowed to be issued to memory before previous loads have completed, be permitted?
- Which same address dependencies must be enforced?
  - Load a ; Load a ;
  - Store a, Load a ;
- How many different fences should be supported?
  - Different fences can have different performance implications

Even TSO allows this reordering

Atomic memory systems

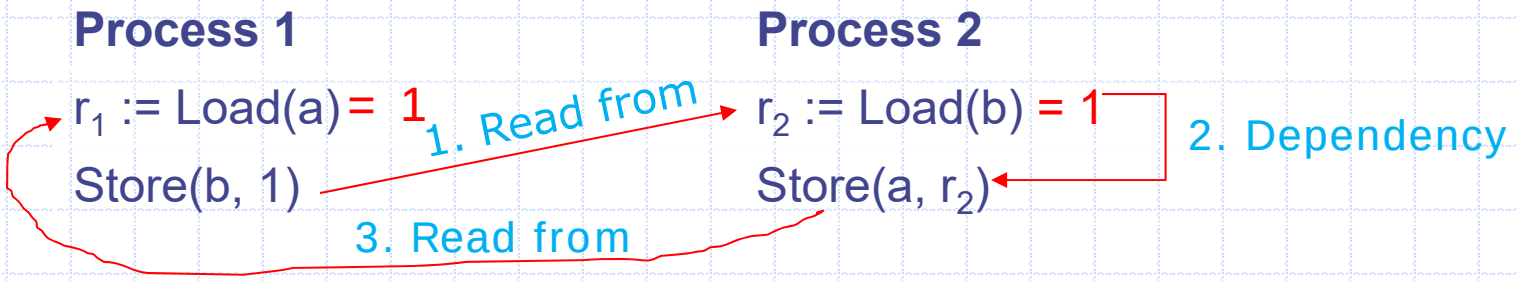


- Add a request to rb
- Later process the oldest request for any address on any port

Consensus: RISC-V memory model definition will rely only on atomic memory

Example: Ld-St Reordering

- Permitting a store to be issued to the memory before previous loads have completed, allows load values to be affected by future stores in the *same* thread



  - Load a misses in local cache
  - Store a is written to memory
  - Load b reads the latest value
  - Store a is written to memory
  - Load a reads the latest value

Load-Store Reordering

- Nvidia says it cannot do without Ld-St reordering
- Although IBM POWER memory model allows this behavior, the server-end POWER processors do not perform this reordering for reliability, availability and serviceability (RAS) reasons
- MIT opposes the idea because it complicates both the operational and axiomatic definitions, and MIT estimates no performance penalty in disallowing Ld-St reordering

Nevertheless MIT has worked diligently to come up with a model that allows Ld-St ordering

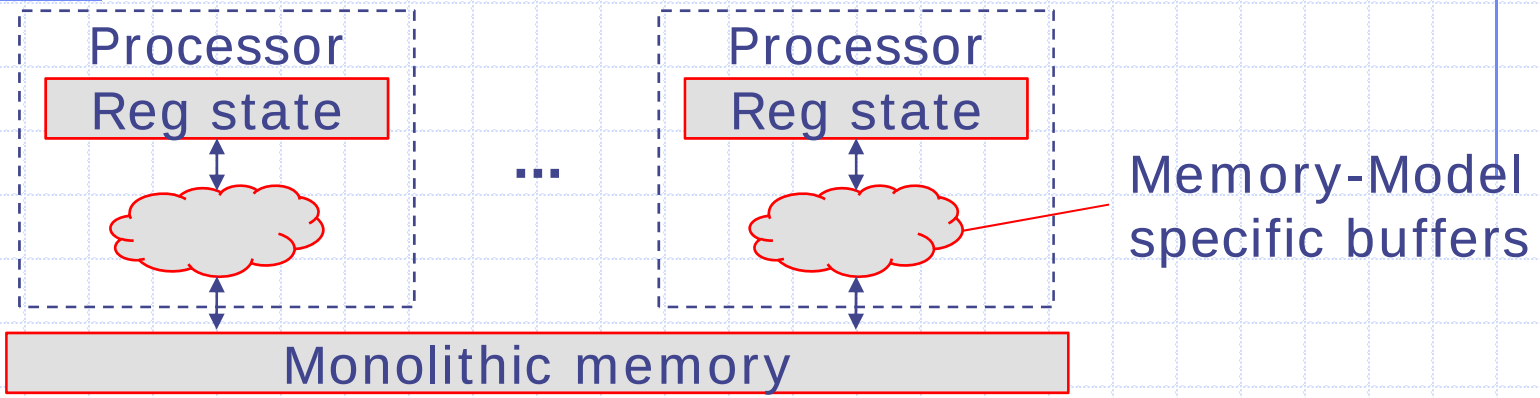
WMM: MIT proposal [PACT2017]

- Philosophy: Develop a weak memory model that does not rule out any hardware optimizations (WMM)

> Even for multithreaded programs, let programmers think in terms of sequential execution of threads. However some loads and stores are for communication and may be followed or preceded by fences.

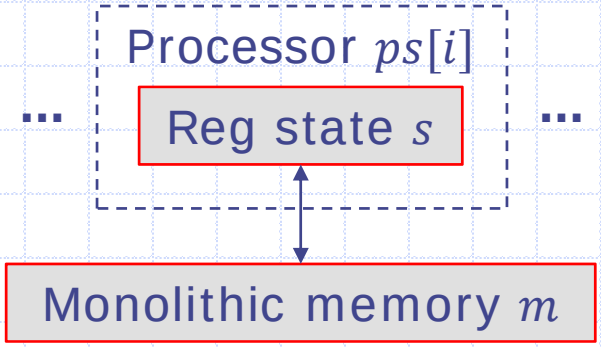
- Suffer the pain of inserting fences once; the code should work on any reasonable machine

Instantaneous Instruction Execution (to simplify definitions)



- Instructions execute in-order and instantaneously; processor state is always up-to-date
- Monolithic memory processes loads and stores instantaneously
- Data moves between processors and memory *asynchronously* according to some background rules

SC in I²E

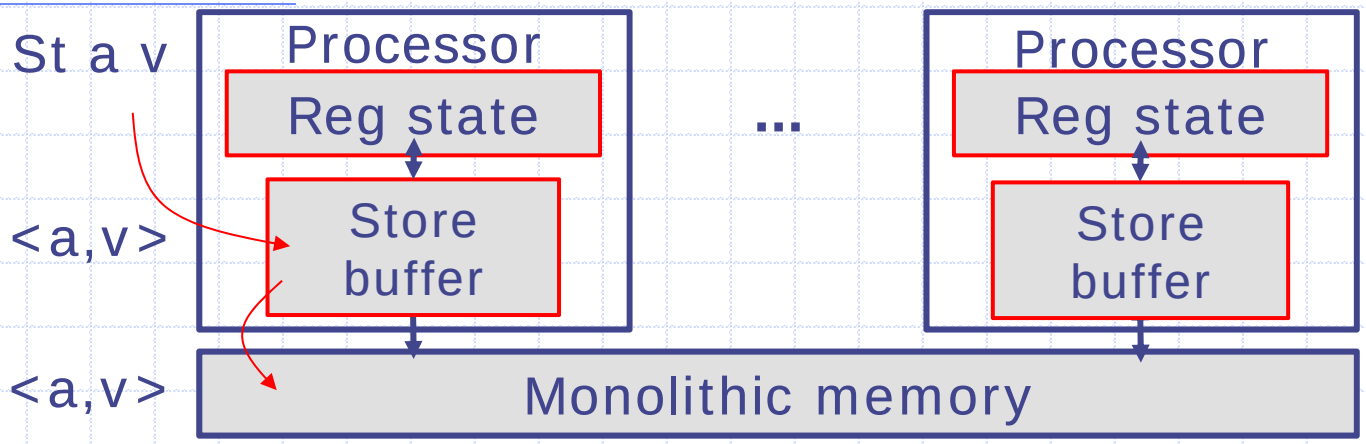


- Pick a processor, execute its current instruction instantaneously and update the register state
  - A Load reads the memory instantaneously
  - A Store updates the memory instantaneously

SC allows no reordering of instructions

Dijkstra 1966, Lamport 1973

TSO in I²E



simple and vendor independent

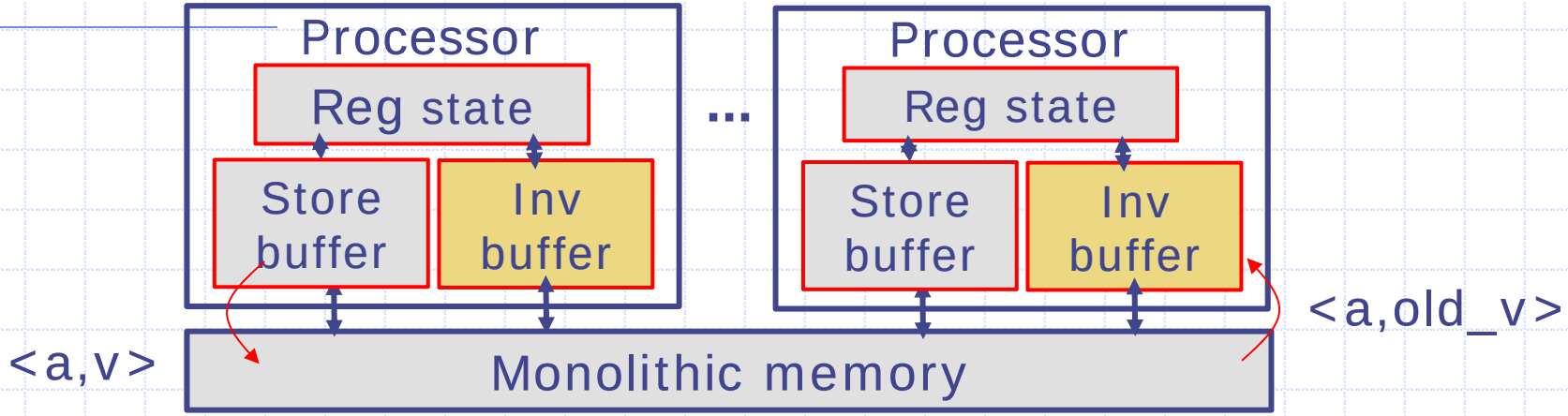
- A store first goes into the Store buffer (SB)
- A load reads the youngest corresponding entry from SB before reading the memory
- A store is dequeued from the SB in FIFO order to update the monolithic memory (*background rule*)
- A *commit fence* stalls local execution until SB is empty

per address ⇒ PSO

TSO allows loads to overtake stores

WMM: Also allows load-Load reordering

Sizhuo Zhang, Murali Vijayaraghavan, Arvind



- Introduce *Invalidation Buffers* (IB), a conceptual device to make stale values visible
- Whenever *<a,v>* from SB is moved to the memory, the old value for *a* in memory is inserted into IB of all other processors and all values for *a* are purged from the local IB
- Values in IB and memory can be read by a load if the address is not found in the SB; staler values than the one read are purged from IB
- A *Reconcile fence* clears the invalidation buffer
- A *Commit fence* clears the store buffer

Intuitive Understanding of WMM

- Allowed reorderings
  - A load can overtake loads (to different addresses), stores and Commit fences
  - A Store can overtake stores (to different addresses)
- Reconcile stops younger loads from reading stale values (Acquire semantics)
- Commit advertises older stores globally (Release semantics)

Fences for Common Paradigms:

Producer-consumer by signaling

Global Memory	`int *data = new int[8];` `int *flag = new int;`
Thread 1	Thread 2

`data[0] = 100;`
`...`
`data[7] = 800;`
`Commit;`
`*flag = 1;`
	`while(*flag != 1) {};`
	`Reconcile;`
	`int d0 = data[0];`
	`...`
	`int d7 = data[7];`

- Reconcile prevents d0~d7 from reading stale values in *ib*
- Commit prevents stores to data[0~7] staying in *sb*

Fences for Common Paradigms:

Properly Synchronized Programs

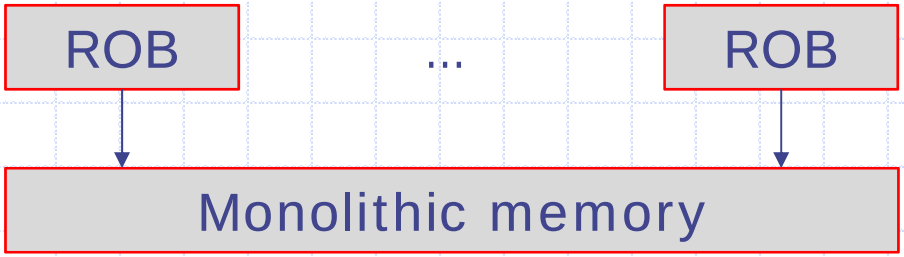
Global Memory `mutex_t mutex;`	
Thread 1	Thread 2

`mutex.lock();`
`Reconcile;`
`// critical section`
`Commit;`
`mutex.unlock()`
	`mutex.lock();`
	`Reconcile;`
	`// critical section`
	`Commit;`
	`mutex.unlock();`

- Critical sections are preserved by locks

Model X: Also allows Ld-St reordering

Sizhuo Zhang, Murali Vijayaraghavan, Arvind



- Each processor is an unbounded ROB with a perfect branch predictor
- Instructions in ROB are marked as *done* or *!done*
  - ALU or branch instructions are executed when operands are available and marked as done
  - Loads get their values either by bypassing in ROB or by reading the monolithic memory
  - Stores update the monolithic memory

The operational model also works for WMM with minor modifications

Model X: General considerations

- No speculative stores
- Enforces the ordering between two consecutive loads for the same address (same as WMM)
- Enforces data dependencies (WMM does not)

Process 1	**Process 2**
Store(a, 1)	r_1 := Load(b) = a
Commit	r_2 := Load(r_1) = 0
Store(b, a)	

WMM allows load-value prediction

WMM: A load value is predicted at fetch time

Rule to execute load inst i

- Address has been computed
- All older Reconcile fences have been done
- Check for same address operations: Search the ROB from i towards the oldest instruction for the first not-done memory instruction with the same address
  - If a not-done load is found, then i cannot be executed
  - If a not-done store to a is found then if the data for the store is ready, then execute i by bypassing the data from the store, and mark i as done; otherwise, i cannot be executed.
  - If nothing is found then execute i by reading the monolithic memory, and mark i as done

WMM: if the loaded value differs from the previously predicted value, then kill the load

Rule to execute store inst *i*

- Address and data of *i* have been computed
- All older fences have been done
- All older branches have been done
- All older loads and stores have computed their addresses
- All older loads and stores for the same address have been done
- Update the monolithic memory and mark *i* as done

WMM: When all older loads (not just to the same address) have been done

Rule to kill speculative loads

- Compute the address of a load or store instruction i
- Search ROB from i towards the youngest instruction for the first memory instruction with the same address
  - If the instruction found is a done load, kill it

Formal Results

- We have provided axiomatic definitions for both WMM and Model X
- We have also proven the following theorems for both models :

Soundness: $Model_{operational} \subseteq Model_{axioms}$

Completeness: $Model_{axioms} \subseteq Model_{operational}$

Summary

- RISC-V memory model debate is not settled; in spite of lot of research by the Memory Model Committee (Chair Dan Lustig), the *community* may vote for TSO
- We have only been discussing the base memory model without the systems instructions (fences for TLBs and self modifying codes)
- We have also not touched the topic of communication between the processors and accelerators

Please voice your opinions by joining the online discussions

Thanks!

Extras

Model X rules

- Fetch an instruction
  - Fetch the next instruction into ROB; predict the next PC

    WMM: if the fetched instruction is a load, predict its value
- Execute a reg-to-reg or branch instruction
  - When source operands are ready
  - Mark the instruction as done
  - If branch is mispredicted previously, then flush ROB
- Compute store address when source operands are ready
- Execute a Commit fence
  - When all previous memory instructions and fences are done
  - Mark the fence as done
- Execute a Reconcile fence
  - When all previous loads and fences are done
  - Mark the fence as done

Compilation from C++11 to WMM

C++ operations	WMM instructions
Non-atomic Load / Load Relaxed	Ld
Load Consumed / Load Acquire	Ld; Reconcile
Load SC	Commit; Reconcile; Ld; Reconcile
Non-atomic Store / Store Relaxed	St
Store Released / Store SC	Commit; St

- C++11 introduces atomic variables in addition to the ordinary (non-atomic) ones
  - Non-atomic variables are accessed by non-atomic Ld/St
  - Atomic variables can be accessed by Ld/St with different semantics (e.g. load acquire and store release)

Atomic read-modify-write

- Directly load from and store into the monolithic memory
- SB should not contain the address
- The address should be purged from IB

Insertion of fences in racy programs is difficult

Lock free enque

```
void enq(queue_t *queue, value_t value) {
  node_t *tail; node_t *next;
  node_t *node = new node_t;
  node->value = value; node->next = NULL; F1: Commit;
  while (true) {
    L1: tail = queue->tail; F2: Reconcile;
    L2: next = tail->next; F3: Reconcile;
    L3: if (tail == queue->tail)
      if (next == NULL) {
        if (CAS(&tail->next, next, node)) break;
      } else CAS(&queue->tail, tail, next)
  }
  CAS(&queue->tail, tail, node);
}
```