Sequential Consistency & TSO

Core C1	Core C2
data = 0, flag ≠ SET	
S1: **store** data = NEW	
S2: **store** flag = SET	L1: **load** *r1* = flag
	B1: if (*r1* ≠ SET) goto L1
	L2: **load** *r2* = data;

Will *r2* always be set to NEW?

Core C1	Core C2
data = 0, flag ≠ SET	
S1: **store** data = NEW	
S2: **store** flag = SET	L1: **load** *r1* = flag
	B1: if (*r1* ≠ SET) goto L1
	L2: **load** *r2* = data;

Will *r2* always be set to NEW?

NO

Core C1	Core C2
data = 0, flag ≠ SET	
S1: **store** data = NEW	
S2: **store** flag = SET	L1: **load** *r1* = flag
	B1: if (*r1* ≠ SET) goto L1
	L2: **load** *r2* = data;

Will *r2* always be set to NEW?

S1 and S2 may get reordered

Reordering

Store-Store	• Non FIFO write buffer
	• Examples:
	  • 1st store misses cache, 2nd hits
	  • 2nd store coalesces with earlier store

Load-Load	• Out-of-order execution • Can have same effect as store-store
Load-Store	• Can cause incorrect behaviors, such as load after mutex unlock
Store-Load	• FIFO write buffer • Out-of-order execution

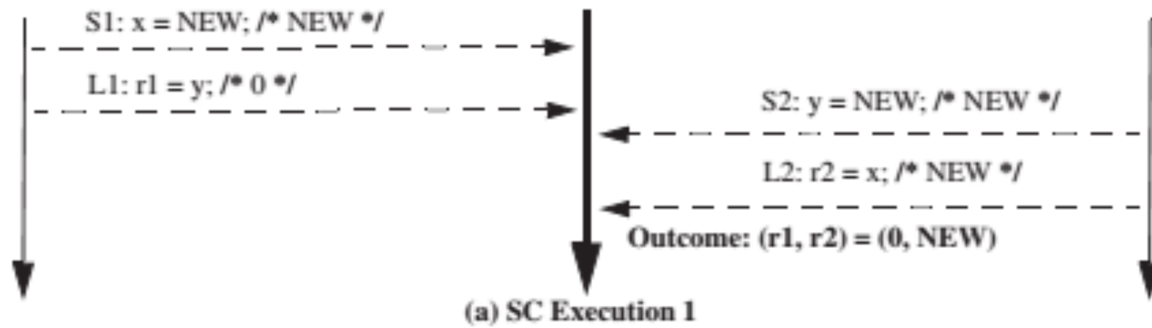
What is a memory consistency model?

A *memory consistency model* is a specification of the allowed behavior of multithreaded programs executing with shared memory.

Sequential Consistency (SC)

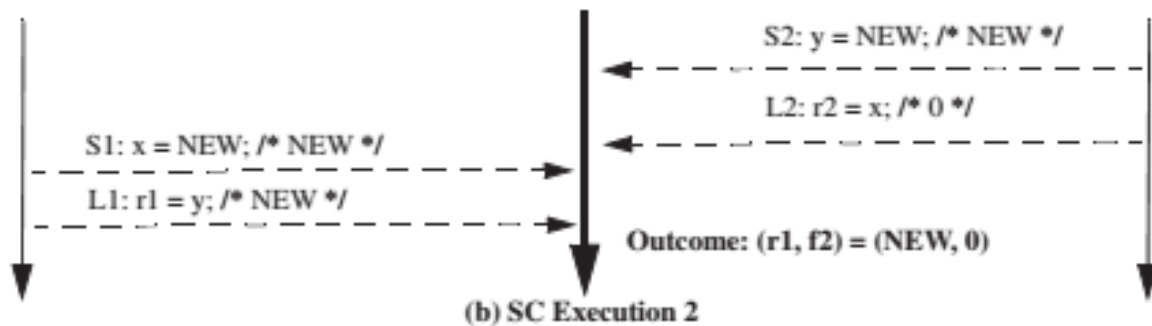
- The most intuitive MC model is sequential consistency.
- First formalized by L. Lamport:
  - How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–91, Sept. 1979.
- A single core is sequential if "*the result of an execution is the same as if the operations had been executed in the order specified by the program*."
- A multiprocessor is sequentially consistent if "*the result of any execution is the same as if the operations of all cores were executed in some sequential order, and the operations of each core appear in this sequence in the order specified by its program*."
- The total order of operations is called memory order
- In SC, memory order respects each core's program order, but other consistency models may permit memory orders that do not always respect the program orders.

SC/Non-SC Example

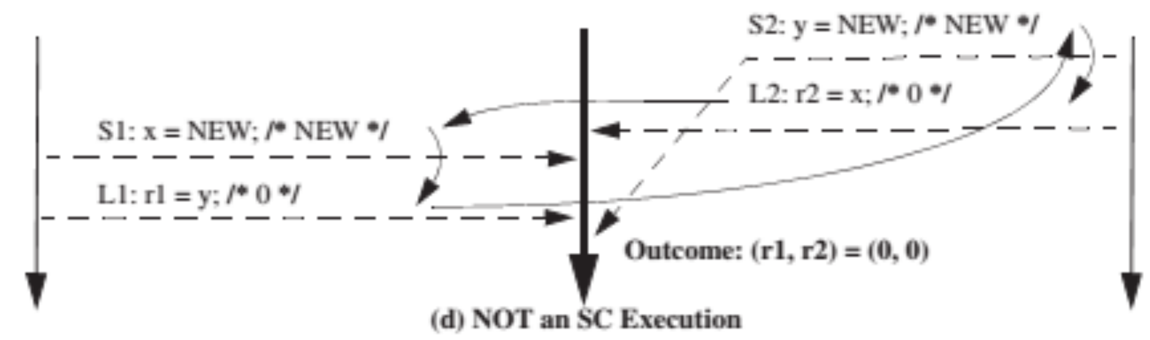


(a) SC Execution 1

(c) SC Execution 3



(b) SC Execution 2



(d) NOT an SC Execution

Formalism

- All cores insert their loads and stores into the memory order (<m) respecting their program order (<p), regardless of whether they are to the same or different addresses (i.e., a=b or a≠b).
  - If L(a) <p L(b) ⇒ L(a) <m L(b) /* Load→Load */
  - If L(a) <p S(b) ⇒ L(a) <m S(b) /* Load→Store */
  - If S(a) <p S(b) ⇒ S(a) <m S(b) /* Store→Store */
  - If S(a) <p L(b) ⇒ S(a) <m L(b) /* Store→Load */

- Every load gets its value from the last store before it (in global memory order) to the same address:
  - Value of L(a) = Value of MAX <m {S(a) | S(a) <m L(a)},
    where MAX <m denotes “latest in memory order.”

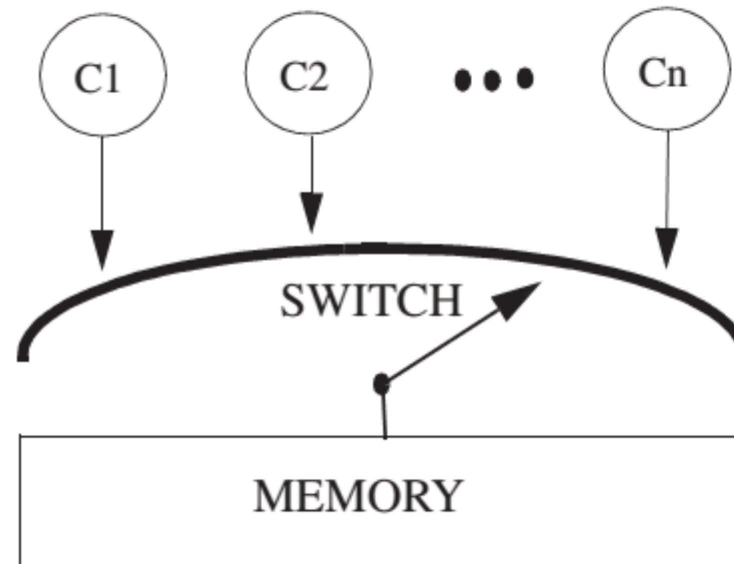
SC Ordering Summary

		Operation 2		
		Load	Store	Atomic RMW
Operation 1	Load	✓	✓	✓
	Store	✓	✓	✓
	Atomic RMW	✓	✓	✓

✓ - **Order is enforced**
✗ - **Order not enforced**

Naïve SC Implementation

- Option 1 - Just run everything on a single core
- Option 2 – memory access through a “switch” serializing memory accesses:



Each core Ci seeks to do its next memory access in its program order <p.

The switch selects one core, allows it to complete one memory access, and repeats; this defines memory order <m.

Total Store Order (TSO) – Motivation (1)

- Processors use *write buffers* to hold committed stores until the memory system can process them.
- A store enters the write buffer when the store commits, and a store exits the write buffer when the block to be written is in the cache in a read–write coherence state.
- A store can enter the write buffer before the cache has obtained read–write coherence permissions
  - The write buffer hides the latency of a store miss.
  - Stores are common, being able to avoid stalling on most of them is an important benefit.
- For a single-core processor
  - a write buffer can be made invisible by ensuring that a load returns the value of the most recent store even if one or more stores to are in the write buffer.
  - This can be done by *bypassing* the value of the most recent store as determined by program order,
  - Or by stalling a load if a store to the same address is in the write buffer.

Total Store Order (TSO) – Motivation (2)

- When building a multicore processor, it seems natural to use multiple cores, each with its own bypassing write buffer
  - Assume that the write buffers continue to be architecturally invisible as before.
  - This assumption is wrong!

Core C_1	**Core C_2**
S1: x = NEW	S2: y = NEW
L1: r1 = y	L2: r2 = x

- Example:
  - Core C_1 executes store S1, buffers the NEW value in its write buffer.
  - Core C_2 executes store S2, buffer the NEW value in its write buffer.
  - Both cores perform L1 and L2 ➔ obtain the old values!
  - Finally, both cores' write buffers update memory with NEW

TSO

- The option chosen by SPARC and x86* was to abandon SC.
- Instead, implement an MC model that allows use of a FIFO write buffer in each core.
- The new model, TSO, allows the outcome “(r1, r2) = (0, 0)”!
- Behaves like SC for many programming idioms and is well defined in all cases, but can be surprising.

* Not entirely TSO

TSO Formalism (1)

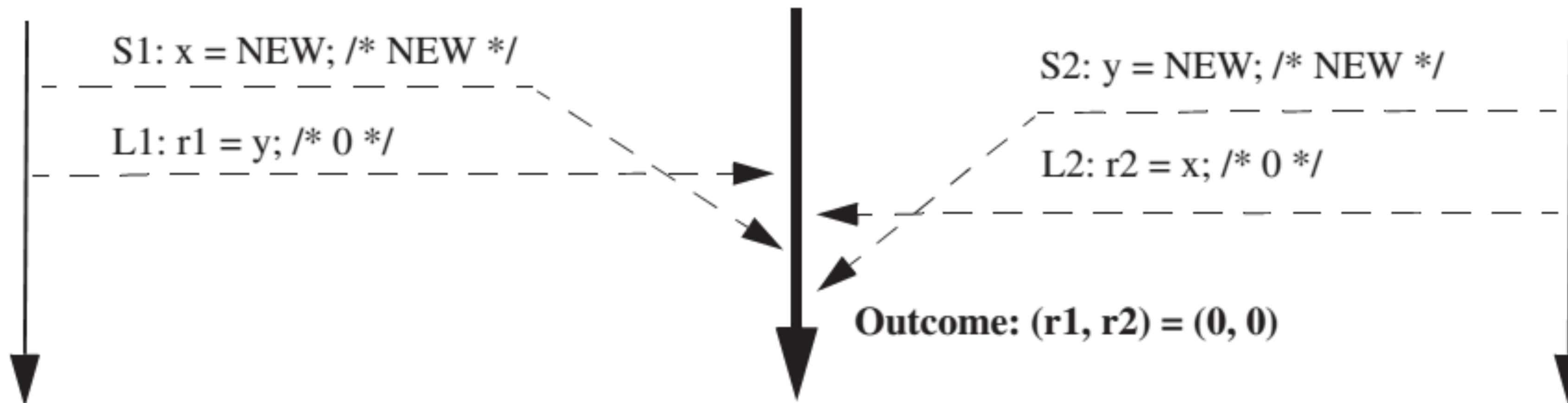
- All cores insert their loads and stores into the order <m respecting their program order, regardless of whether they are to the same or different addresses (i.e., a=b or *a≠b*).
  - If L(a) <p L(b) ⇒ L(a) <m L(b) /* Load→Load */
  - If L(a) <p S(b) ⇒ L(a) <m S(b) /* Load→Store */
  - If S(a) <p S(b) ⇒ S(a) <m S(b) /* Store→Store */
  - ~~If S(a) <p L(b) ⇒ S(a) <m L(b)~~ /* Store→Load */: Enable FIFO write buffer
- Every load gets its value from the last store before it (in global memory order) to the same address:
  - Value of L(a) = Value of MAX <m {S(a) | S(a) <m L(a) **or S(a)<p L(a)**}
    (<m "last in memory order.", <p "last in program order")
  - The value of a load is the value of the last store to the same address that is either (a) before it in memory order or (b) before it in program order (but possibly after it in memory order)
  - Option (b) taking precedence - write buffer bypassing overrides the rest of the memory system.

TSO Formalism (2)

- Store → Load addressed with FENCEs
- Executing a FENCE on core C_i ensures that C_i's memory operations before the FENCE, in program order, get placed in memory order before C_i's memory operations after the FENCE.
- FENCEs (memory barriers) are rarely used in TSO because TSO usually "does the right thing".
- FENCEs play an important role for relaxed models.

TSO Example

Core C_1	Core C_2
S1: x = NEW	S2: y = NEW
L1: r1 = y	L2: r2 = x

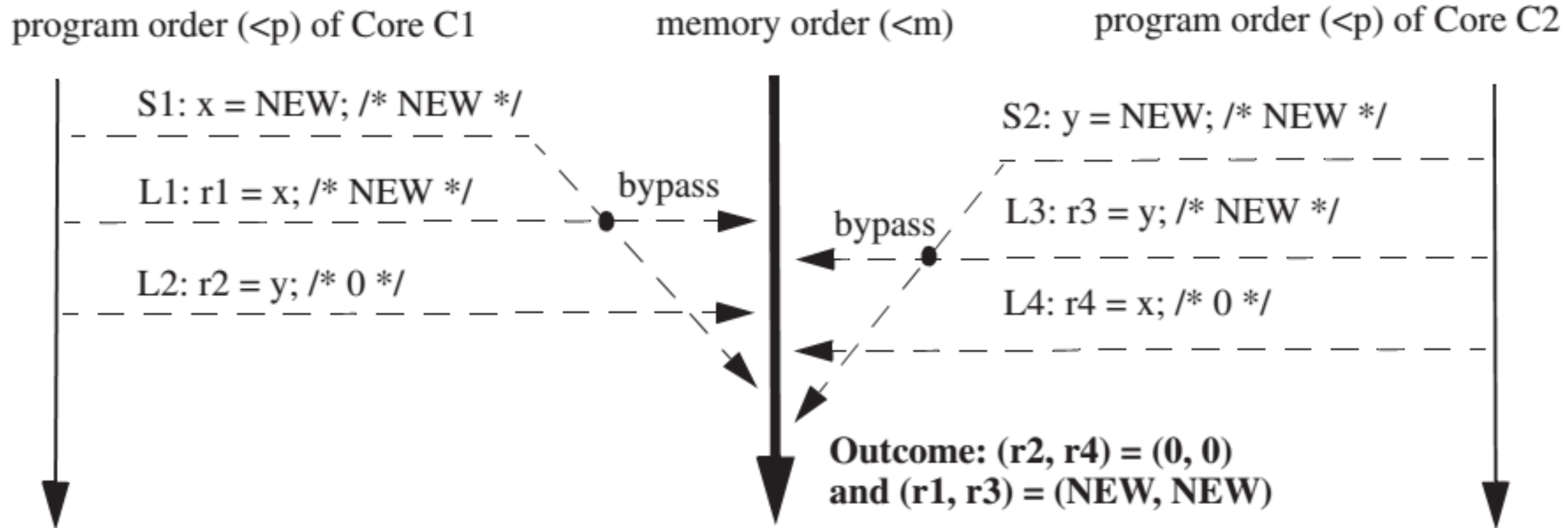


(d) TSO Execution, but NOT an SC Execution

TSO Bypass Example

- Can r1 or r3 be set to 0 if r2=r4=0?
- No, must always be set to NEW.

Core C_1	Core C_2
S1: x = NEW	S2: y = NEW
L1: r1 = x	L3: r3 = y
L2: r2 = y	L4: r4 = x



TSO Atomics

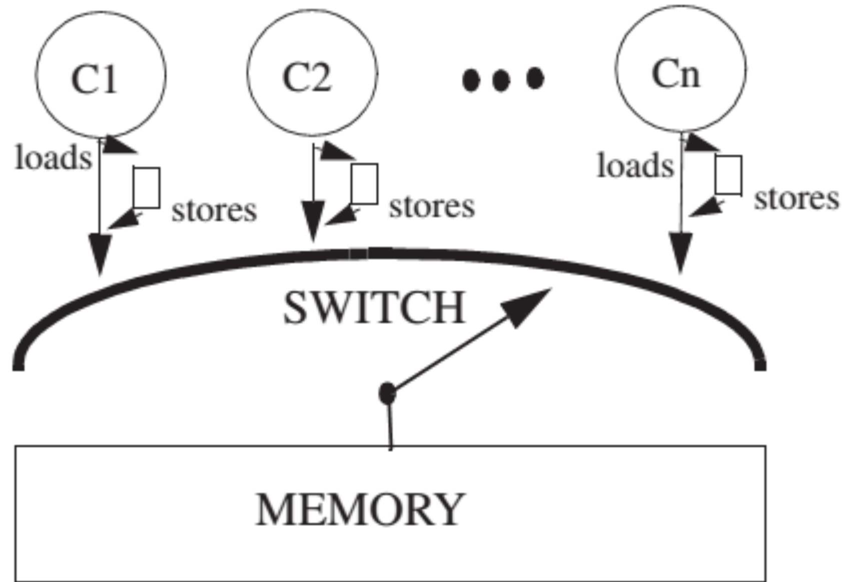
- A RMW is an atomic load-store.
- RMW cannot be reordered with earlier stores or loads due to TSO rules:
  - Load part cannot be executed before earlier loads
  - Load part cannot be executed before earlier store, as the RMW operation is atomic and this will reorder the store-half before the store as well, which is not allowed.
- This means the RMW cannot start until the write buffer has been drained, e.g. effectively a fence.
  - **Even more: it requires exclusive RW coherence permissions on the address, which are held for the entire duration of the RMW.**
  - Optimization – If the all entries in the write-buffer are already in exclusive RW, no need to drain buffer.

TSO Ordering Summary

		Operation 2			
		Load	Store	Atomic RMW	FENCE
Operation 1	Load	✓	✓	✓	✓
	Store	B	✓	✓	✓
	Atomic RMW	✓	✓	✓	✓
	FENCE	✓	✓	✓	✓

✓ - Order is enforced
B - Requires bypassing
✘ - Order not enforced

TSO Naïve Implementation



This implementation is the same as for Figure3-3, except that each core Ci has a FIFO write buffer that buffers stores until they go to memory.

(a) A TSO Implementation Using a Switch

Fences can be implemented by draining the write buffer.

Analyzing MCs

- A good memory consistency model should possess 3 Ps:
  - *Programmability*: A good model should make it easy to write MT programs. The model should be intuitive to most users, even those who have not read the details. It should be precise, so that experts can push the envelope of what is allowed.
  - *Performance*: A good model should facilitate high-performance implementations at reasonable power, cost, etc. It should give implementors broad latitude in options.
  - *Portability*: A good model would be adopted widely or at least provide backward compatibility or the ability to translate among models.

3 Ps for SC and TSO

- *Programmability*:
  - SC is the most intuitive.
  - TSO is close because it acts like SC for common programming idioms.
    Subtle non-SC executions can bite programmers and tool authors.
- *Performance*: For simple cores, TSO can offer better performance than SC, but speculation can help SC.
- *Portability*: SC is widely understood, while TSO is widely adopted.

What's Next

- Thursday – Relaxed Consistency models

DEC Alpha

Core C1	Core C2
p=&x, x=1, y = 0	
y=1	i=*p
MemoryBarrier()	
p=&y	
i can be 0!	

http://www.cs.umd.edu/~pugh/java/memoryModel/AlphaReordering.html

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	AMD64	IA-64	zSeries
Loads reordered after loads	✓	✓	✓	✓	✓					✓	
Loads reordered after stores	✓	✓	✓	✓	✓					✓	
Stores reordered after stores	✓	✓	✓	✓	✓	✓				✓	
Stores reordered after loads	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Atomic reordered with loads	✓	✓		✓	✓					✓	
Atomic reordered with stores	✓	✓		✓	✓	✓				✓	
Dependent loads reordered	✓										
Incoherent instruction cache pipeline	✓	✓		✓	✓	✓	✓	✓		✓	✓

Q&A

Bugs?

- Beyond the need for preserving program semantics, is there any other benefit/reason related to memory consistency models?
- Are there any major software errors/accidents known that were caused by TSO because the intuitively model for programmers is SO?

- Islam, Mohammad Majharul, and Abdullah Muzahid. "Characterizing Real World Bugs Causing Sequential Consistency Violations." *HotPar*. 2013.
  - Found lots of bugs in Apache, GCC, Mozilla, MySQL, JVM, Cilk.

Cost

- Considering that experimental tests show TSO is the memory consistency model implemented by x86 processors. Why do Intel/AMD not reveal their x86 MC? Are they unable to satisfy it in every case? Do they not know for sure if it will hold in every case? Do they not want to give this advantage to their competitor?
- Is there any general consensus on which model is adequate/ideal for current workloads?
- What is the actual performance cost in implementing SC over TSO, amortized with fence costs, especially with regards to SC languages such as Java volatiles?
- What are the implications on GPU chip area if a TSO-like implementation is done? (Because GPUs have 100s of "CUDA cores")
- Naeem, Abdul, Axel Jantsch, and Zhonghai Lu. "Architecture support and comparison of three memory consistency models in NoC based systems."*Digital System Design (DSD), 2012 15th Euromicro Conference on*. IEEE, 2012.
  - ***The average speedup for the RC, PSO and TSO models in the 8x8 network under different application workloads is increased by 34.3%, 10.6% and 8.9%, respectively, over the SC model. The area cost for the TSO, PSO and RC models is increased by less than 2% over the SC model at the interface to the processor.***

GPU

- Do common GPU architectures implement FENCE instructions and if so, is there a significant associated bottleneck?
  - void __threadfence();
    void __threadfence_block();
    void __threadfence_system();
  - Slow – not quantified anywhere formally
- Considering GPGPU programs that we have used relied heavily on memory being accessed, GPGPUs surely need to have a consistency and coherence model in place. So, how does the GPGPU memory consistency work?

x86 TSO

- Is x86 TSO?
  - Intel streaming SMID ISA extension supports write-combining and weakly ordered MC, with no-ordering whatsoever with regard to other WC and non-WC instructions.
- In Implementing x86 TSO, they state that a shared write buffer with thread tags is more common than individual per-core write buffers. Why is this method more common? How is a centralized write buffer with TSO ordering better than both an individual buffer per-core and a centralized buffer that provides store bypassing across cores?

Other

- Can you explain in more detail how program order and memory order work when considering out-of-order execution / branch predication / cache memory latencies?
- In 3.9, they describe how you can achieve SC in atomic operations by denying consistency requests before the completion of the final store, and describe this as an improvement over the "naive" method of blocking memory accesses. I don't see how these cases are different; if other threads don't get their requests to that block until the store finishes, how is this an improvement?
- Could you explain the two checks presented by Gharachorloo et al. under Dynamically Scheduled Cores in detail?
- "Importantly, the eviction of a cache block—due to a coherence invalidation or to make room for another block—that contains a load's address in the address queue squashes the load and all subsequent instructions, which then re-execute." Could you explain this point?
- Is there any case in which a FENCE instruction would make sense in an SC implementation, or are they always effectively no-ops?
  - Yes?