

A Taxonomy of Multiprocessor Memory-Ordering Models

Gil Neiger
Intel Corporation
Microprocessor Research Labs

Multiprocessor Systems and Shared Memory

- Multiple processors share memory
- Memory managed by one or more memory controllers (controllers integrated with or discrete from processors)
- What is memory behavior under concurrent data access?
- Answer determined by system's *memory-ordering model*
- Defines order that processors perceive concurrent accesses
- Should be defined based on *ordering* not *timing* of accesses

Memory-Ordering Models

- Different platforms support different models
- Some platforms (e.g., SPARC) support multiple models
- Ideal: model specification is implementation independent: future implementations should continue to support model
- Reality: initial/planned implementations inform specification
- Some cases: model evolves with implementations and changing needs of software (these tend not to be published)

Specifying Memory-Ordering Models

- More complex models attempt to expose more performance
- Some specified only by description of (first) implementation
- Difficult to write programs to such specifications
- SW tends to be conservative, forgoing performance gains
- Verification of future (different) implementations difficult

Outline: Models and Issues Covered

- Memory-ordering models:
 - Sequential consistency
 - SPARC TSO
 - SPARC PSO
 - IA32 (Pentium®)
 - Itanium™ release consistency
- Implementation issues:
 - Store buffering
 - Store forwarding
 - Non-unique store order
 - Large multiprocessors and “mixed” models
- This talk informed by my experiences with Intel architects

Defining Memory-Ordering Models: Basics

- Program order (of a processor's operations):
per-processor order of memory accesses determined by SW
- Visibility order (of all operations):
order of memory accesses perceived by one or more processors: every load from a location returns value written by most recent previous store (or initial value 0 if none)
- Notation:
 - L = load; S = store
 - $X \rightarrow Y$ means "X precedes Y in program order"
 - $X \Rightarrow Y$ means "X precedes Y in the visibility order"

Simple Definition: Sequential Consistency

- Defined by Lamport in 1979
- Simplest model to define and understand*
- Execution is *sequentially consistent* if $X \rightarrow Y$ implies $X \Rightarrow Y$
- Visibility order must completely respect program order

*But most costly to implement

Sequential Consistency: Example

P0	P1
S(a)1	S(b)1
L(b)0	L(a)1

- Visibility order:
 $S(a)1 \Rightarrow L(b)0 \Rightarrow S(b)1 \Rightarrow L(a)1$
- Respects program order
- Each load returns proper value:
 - initial value (0) for L(b)
 - stored value (1) for L(a)

Implementation Issue: Store Buffering

- To improve performance, multiprocessors buffer stores
- Stores must wait for ownership of data item (cache line)
- Loads can read from local copy
- Result: later loads appear to “pass” earlier stores

Weaker Consistency Model: IBM 370

- Execution is IBM 370 if $X \Rightarrow Y$ whenever $X \rightarrow Y$ and
 - X is a load;
 - X and Y are both stores; or
 - X and Y access a common location (cache coherence)
- $X \rightarrow Y$ and $Y \Rightarrow X$ are allowed if
 - X is a store;
 - Y is a load; and
 - X and Y do not access a common location

Store Buffering: Example

P0	P1
S(a)1	S(b)1
L(b)0	L(a)0

- Visibility order:
 $L(b)0 \Rightarrow L(a)0 \Rightarrow S(a)1 \Rightarrow S(b)1$
- Not sequentially consistent
- Allowed by IBM 370 model
- Breaks Dekker's algorithm for mutual exclusion

Implementation Issue: Store Forwarding

- IBM 370 model:
Load “hitting” store buffer blocks until store “becomes visible”
- Many multiprocessors forward buffered store values to loads
- Impact on model:
 - such loads can be bypassed with stores that satisfy them; or
 - stores appear to become visible to storing processor first
- Example: SPARC TSO (total store order)

Weaker Consistency Model: SPARC TSO

- Execution's visibility order categorizes loads:
 - local loads: read values stored by same processors
 - non-local loads: read values stored by other processors
- Execution is TSO if $X \Rightarrow Y$ whenever $X \rightarrow Y$ and
 - X is a **non-local** load;
 - X and Y are both stores; or
 - X and Y access a common location
- $X \rightarrow Y$ and $Y \Rightarrow X$ are allowed if
 - X is a store or **local** load;
 - Y is a load; and
 - X and Y do not access a common location

Store Forwarding: Example

P0	P1
S(a)1	S(b)1
L(a)1	L(b)1
L(b)0	L(a)0

- Visibility order:

$L(b)0 \Rightarrow L(a)0 \Rightarrow S(a)1 \Rightarrow S(b)1 \Rightarrow L(a)1 \Rightarrow L(b)1$

- Not allowed by IBM 370 model
(e.g., L(b)0 passes L(a)1)
- Allowed by SPARC TSO (bypassed loads are local)
- Breaks Peterson's algorithm for mutual exclusion

Implementation Issue: Unordered Store Buffers

- SPARC TSO implementations:
 - Store buffers are linearly ordered
 - Any pair of stores become visible in program order
- Alternative: SPARC PSO (partial store order)
- Store visibility not constrained by program order

SPARC PSO

- Execution is PSO if $X \Rightarrow Y$ whenever $X \rightarrow Y$ and
 - X is a non-local load;
 - X is a load and Y is store; or
 - X and Y access a common location
- $X \rightarrow Y$ and $Y \Rightarrow X$ are allowed if
 - X and Y do not access a common location and either
 - (1) X is a store or
 - (2) X is a local load and Y is a load

Unordered Store Buffers: Example

P0	P1
S(a)1	L(b)1
S(b)1	L(a)0

- Visibility order:
 $S(b)1 \Rightarrow L(b)1 \Rightarrow L(a)0 \Rightarrow S(a)1$
- Not allowed by TSO (e.g., S(b)1 passes S(a)1)
- Allowed by SPARC PSO (stores can pass stores)
- Breaks “producer-consumer” code

Unique Store Order: Example

P0	P1	P2
S(a)1	L(b)1	L(a)1
S(b)1	L(a)0	L(b)0

- Not allowed by PSO:
 - P1's loads imply $S(b)1 \Rightarrow S(a)1$
 - P2's loads imply $S(a)1 \Rightarrow S(b)1$
- All models described so far require a single visibility order
- All observers agree on the order of any two stores*

*Does not include storing processors for TSO, PSO

Unique Store Order: Implementation Issues

- Some multiprocessors employ a single *memory bus*
- Order of stores corresponds to order seen on memory bus
- Single memory bus = single visibility order
- Larger and more recent multiprocessors use other topologies
- Examples:
 - Cluster systems with multiple buses and point-to-point interconnects
 - Multithreaded processors whose threads share a store buffer

Relaxing the Unique Store Order: IA32

- IA32 (Pentium®) platforms support *processor ordering*
- Processor ordering: stores become visible in program order
- Example PSO execution not allowed by IA32:

P0	P1
S(a)1	L(b)1
S(b)1	L(a)0

Relaxing the Unique Store Order: Example

P0	P1	P2	P3
S(a)1	S(b)1		
		L(a)1	L(b)1
		L(b)0	L(a)0

- Potentially allowed by IA32:
 - Example: P0, P2 threads on one processor; P1, P3 on another
 - Example: P0, P2 in one node of cluster; P1, P3 in another
- Not produced by any existing IA32 multiprocessor*

*To the knowledge of this presenter

Non-Unique Store Order: How to Define?

- General technique: multiple visibility orders
- Separate visibility order per processor
- Each orders all stores and that processor's loads
- Two kinds of constraints
 - intra-order constraints (how each order respects program order)
 - inter-order constraints (how orders are consistent with each other)

Non-Unique Store Order: “Weak TSO”

- Each processor p has a visibility order \Rightarrow_p
- Visibility order \Rightarrow_p categorizes p 's loads as local/non-local
- Intra-order: for each p and for each X and Y ordered by \Rightarrow_p , $X \Rightarrow_p Y$ whenever $X \rightarrow Y$ and
 - X is a non-local load;
 - X and Y are both stores; or
 - X and Y access a common location
- Inter-order: if X and Y store to a common location, then either $(\forall p)(X \Rightarrow_p Y)$ or $(\forall p)(Y \Rightarrow_p X)$

Non-Unique Store Order: Example Revisited

P0	P1	P2	P3
S(a)1	S(b)1		
		L(a)1	L(b)1
		L(b)0	L(a)0

- Allowed by "weak TSO":

$S(a)1 \Rightarrow_0 S(b)1$

$S(b)1 \Rightarrow_1 S(a)1$


$S(a)1 \Rightarrow_2 L(a)1 \Rightarrow_2 L(b)0 \Rightarrow_2 S(b)1$

$S(b)1 \Rightarrow_3 L(b)1 \Rightarrow_3 L(a)0 \Rightarrow_3 S(a)1$

- Recall: processor does not order others' loads

Non-Unique Store Order: Another Example

P0	P1	P2
S(a)1	L(a)1	
	S(b)1	L(b)1
		L(a)0



- Allowed by "weak TSO":

$S(a)1 \Rightarrow_0 S(b)1$

$S(a)1 \Rightarrow_1 L(a)1 \Rightarrow_1 S(b)1$

$S(b)1 \Rightarrow_2 L(b)1 \Rightarrow_2 L(a)0 \Rightarrow_2 S(a)1$

- Not allowed by IA32!
- Intra-order constraints on ordering of **causally related** stores

Implementation Issue: Large-Scale Systems

- Single visibility order cannot be implemented scalably
- Similar problems for respecting causally related stores
- Larger systems reorder memory accesses more freely
- Problem: potential reordering complicates programming
- Solution: mixed models with “strong” and “weak” operations

Mixed Model: Release Consistency

- First developed for DASH project at Stanford (1990)
- Most loads and stores are *weak*: few ordering guarantees
- Operations can be *labeled* as *synchronization operations*
- *Acquire* operations cannot be passed by later operations
- *Release* operations cannot pass earlier operations

Release Consistency (continued)

- Acquire/release used to bracket critical sections:
 acquire operation
 weak-op₁
 weak-op₂
 ...
 release operation
- DASH defined two forms of release consistency
 - RC_{SC}: strong operations are sequentially consistent
 - RC_{PC}: strong operations are *processor consistent*
- Processor consistency: slightly weaker than TSO, IA32

Itanium™ Release Consistency

- DASH RC inspired Itanium™ memory ordering (joint Intel/Hewlett-Packard product)
- Informally, "RC_{TSO}"
- Weak operations are largely unordered
 - WL = weak load
 - WS = weak store
- Strong operations have TSO-like semantics
 - LA = load acquire
 - SR = store release

Itanium™ Example: Weak Operations

P0	P1
WS(a)1	WL(b)1
WS(b)1	WL(a)0

- This execution is allowed by Itanium™
- Operations can be arbitrarily reordered
- No guarantee that weak stores are seen in program order


Itanium™ Example: Basic Ordering

P0	P1
WS(a)1	LA(b)1
SR(b)1	WL(a)0

- This execution is not allowed by Itanium™
- SR(b)1 cannot pass WS(a)1 (release rule)
- WL(a)0 cannot pass LA(b)1 (acquire rule)
- This is how producer-consumer works with Itanium™

Itanium™ Example: Respect for Causality

P0	P1	P2
SR(a)1	LA(a)1 WS(b)1	LA(b)1 WL(a)0



- This execution is not allowed by Itanium™
- P3 must respect **causal** order from SR(a)1 to WS(b)1
- Any weaker labeling of this execution is allowed by Itanium™

Itanium™ Memory-Ordering: Strawman

- Intra-order: for each p and for each X and Y ordered by \Rightarrow_p ,
 $X \Rightarrow_p Y$ whenever $X \rightarrow Y$ and
 - X is a non-local LA;
 - Y is an SR; or
 - X and Y access a common location
- Inter-order:
 - If X and Y store to a common location,
then either $(\forall p)(X \Rightarrow_p Y)$ or $(\forall p)(Y \Rightarrow_p X)$
 - If X and Y are both SRs,
then either $(\forall p)(X \Rightarrow_p Y)$ or $(\forall p)(Y \Rightarrow_p X)$

Strawman Does Not Respect Causality

P0	P1	P2
SR(a)1	LA(a)1 WS(b)1	LA(b)1 WL(a)0

- Visibility orders allowed by strawman:
 - $SR(a)1 \Rightarrow_0 WS(b)1$
 - $SR(a)1 \Rightarrow_1 LA(a)1 \Rightarrow_1 WS(b)1$
 - $WS(b)1 \Rightarrow_2 LA(b)1 \Rightarrow_2 WL(a)0 \Rightarrow_2 SR(a)1$
- Continued ...

Strawman Definition and Causality

- Program orders:
 $SR(a)1$
 $LA(a)1 \rightarrow WS(b)1$
 $LA(b)1 \rightarrow WL(a)0$
- Visibility orders:
 $SR(a)1 \Rightarrow_0 WS(b)1$
 $SR(a)1 \Rightarrow_1 LA(a)1 \Rightarrow_1 WS(b)1$
 $WS(b)1 \Rightarrow_2 LA(b)1 \Rightarrow_2 WL(a)0 \Rightarrow_2 SR(a)1$
- Allowed by strawman
 - Intra-order constraints easily satisfied
 - Inter-order constraints vacuously satisfied (only one SR)

How Does Itanium™ Respect Causality?

- Additional element of Itanium™ memory ordering
- Store releases are *atomic* across all processors
- If SR(a)1 is visible to P1, then it is also visible to P2

P0	P1	P2
SR(a)1	LA(a)1 WS(b)1	LA(b)1 WL(a)0

Atomicity of Store Releases

- Single ordering of store releases does not capture atomicity
- Intuition behind atomicity suggests simultaneity
- How to specify without referring to timings of events?
- Solution: single visibility order
- Position of store release in order indicates global visibility

Weak Stores and Single Visibility Order

P0	P1	P2	P3
WS(a)1	WS(b)1	LA(a)1 WL(b)0	LA(b)1 WL(a)0

- This is allowed by Itanium™
- How to allow with a single visibility order?
- Decompose each weak store into n sub-operations:
 $RV_p(WS) \approx \text{visibility of } WS \text{ to } p$

Weak Stores and Single Visibility Order (continued)

P0	P1	P2	P3
WS(a)1	WS(b)1	LA(a)1 WL(b)0	LA(b)1 WL(a)0

Fragment of visibility order:

$$RV_2(WS(a)1) \Rightarrow RV_3(WS(b)1) \Rightarrow LA(a) \Rightarrow LA(b) \Rightarrow \\ WL(b) \Rightarrow WL(a) \Rightarrow RV_3(WS(a)1) \Rightarrow RV_2(WS(b)1)$$

Store Releases and Single Visibility Order (continued)

P0	P1
SR(a)1	SR(b)1
LA(a)1	LA(b)1
WL(b)0	WL(a)0

- This is allowed by Itanium™
- How to reconcile with atomicity of store-releases?
- Decompose each store release into 2 sub-operations
 - $LV(SR) \approx$ local visibility of SR to storing processor
 - $LV(SR) \approx$ global visibility of SR to all processors

Store Releases and Single Visibility Order (continued)

P0	P1
SR(a)1	SR(b)1
LA(a)1	LA(b)1
WL(b)0	WL(a)0

Visibility order:

$$\text{LV}(\text{SR}(\text{a})1) \Rightarrow \text{LV}(\text{SR}(\text{b})1) \Rightarrow \text{LA}(\text{a}) \Rightarrow \text{LA}(\text{b}) \Rightarrow \\ \text{WL}(\text{b}) \Rightarrow \text{WL}(\text{a}) \Rightarrow \text{GV}(\text{SR}(\text{a})1) \Rightarrow \text{GV}(\text{SR}(\text{b})1)$$

Formalizing Itanium™ Memory Ordering

- Single visibility order for all loads, store sub-operations
- Constraints on visibility as follows:
 - Sub-operations of store releases suitably ordered
 - Nothing passes a load acquire
 - Store releases do not pass anything (separate rules for LV, GV)
 - Operations by a processor on a common location are ordered*
 - Processors agree on order of stores to a single location
 - A load returns a locally written value if between LV(SR) and GV(SR)
- Special rules for memory fences, atomic read-modify-writes, data-flow dependence, non-cacheable memory

Itanium™ Respects Causality

P0	P1	P2
SR(a)1	LA(a)1 WS(b)1	LA(b)1 WL(a)0

- $GV(SR(a)1) \Rightarrow LA(a)1$ (visibility)
- $LA(a)1 \Rightarrow RV_2(WS(b)1)$ (acquire)
- $RV_2(WS(b)1) \Rightarrow LA(b)1$ (visibility)
- $LA(b)1 \Rightarrow WL(a)0$ (acquire)
- $GV(SR(a)1) \Rightarrow WL(a)0$ (transitivity)

Contradiction: violates visibility to WL(a)

Conclusions

- Most memory-ordering models can be specified in style of Lamport's original definition of sequential consistency
- Models designed for single-bus systems can use a single visibility order for all operations
- Weaker/mixed models can use per-processor visibility orders
- Mixed models with "atomic" operations may be more simply specified (!) with a single visibility order and sub-operations
- Formalization process facilitated by close interactions with system architects (thanks to Tom Willis and Rumi Zahir)