Formally specifying memory consistency models and automatically generating executable specifications

Prosenjit Chatterjee and Ganesh Gopalakrishnan*
School of Computing, University of Utah
{prosen | ganesh}@cs.utah.edu
Technical Report UUCS-01-012

Abstract

Memory ordering properties of shared memory multiprocessors are more subtle and less well understood than cache coherence. These properties tend to be processor or platform specific and are not always formally specified. It is difficult to compare even those platforms whose memory ordering properties have been clearly specified as each such platform is usually specified in its own definitional framework. We present a generic and formal specification scheme to specify any realistic memory consistency model that gives an intuitive undertstanding to architects, and implementors of platforms whose memory model is being defined and also a common definitional framework to compare memory models. Another contribution of the paper is to generate an executable specification automatically, given the specification of any memory consistency model expressed in our newly defined framework. This alternative specification can be used to generate all possible outcomes of small assembly-language multiprocessor programs in a given memory model, which is very helpful for understanding the subtleties of the model. The executable specification can also check the correctness of assembly language programs including synchronization routines.

1 Introduction

Shared-memory multiprocessors are increasingly employed both as servers (for computations, databases, files, and the web) and as clients. To improve performance, multiprocessor system designers use a variety of complex and interacting optimizations . These optimizations include cache coherence via snooping or directory protocols, out-of-order processors, store buffers. These optimizations add considerable complexity at the architectural level and even more complexity at the implementation level. Directory protocols, for example, require the system to transition from many shared copies of a block to one exclusive one. Unfortunately, these transitions must be implemented with many non-atomic lower-level transitions that expose additional race conditions, buffering requirements, and forward-progress concerns. Due to this complexity, industrial product groups spend more time verifying their system than actually designing and optimizing it.

To verify a system, engineers should unambiguously define what "correct" means. For a shared-memory system, "correct" is defined by a memory consistency model. A memory consistency model defines for programmers the allowable behavior of hardware.

Sequential Consistency is the most intuitive model for writing shared memory programs[?] mainly because all the memory operations in an *execution* that obeys SC can be viewed as a total order 'as if' the program has been run in a uniprocessor. However, many commercial processors implement more relaxed memory consistency models in an effort to improve performance. An example is the insertion of FIFO or coalescing store buffers, non-blocking loads and so on. SPARC Total Store Order (TSO)[?], relax the SC requirement where in the total ordering of memory operations a store(st) can appear after a load(ld) that follows it in program order. More relaxed models, such as Compaq(DEC) Alpha[?], allow re-ordering between any two instructions . Cray3TD[?], PowerPc[?] implements

*This work was supported by National Science Foundation Grants CCR-9987516 and CCR-0081406

even more relaxed memory models where a processor can read a store intstruction written by another processor even before the remaining processors see it hence deviating from SC largely in all its memory operations now not forming any such total order . While modern high performance processors gain additional performance from relaxed consistency schemes, they lead to less intuitive programming models unlike SC simply because the total ordering of all memory operations(loads and stores) in an *execution* doesnt exist any more . This section of the work contributes in formally specifying any memory model that is either as strong as or is weaker than sequential consistency in a way such that, an *execution* obeys a given memory model if there exists at least one total order of all memory operations that satisfies all the memory ordering rules defining that memory model. Hence even if we consider a memory model that is very weak compared to sequential consistency and hence have hardly anything common between them, its still possible to constrain their differences to behavior internal to the processor.
In Section 2 we define some memory ordering rules and the concept of logical total ordering of events. In section 3 we categorize memory models into four classes and define memory ordering rules in light of every class. Section 5 describes the methodology to generate an operational model given the memory model specification expressed in our framework and finally, Section 5 concludes with directions towards future work.

2 Memory model specification

Before proceeding, we introduce some terminologies. A concurrent shared memory program is viewed as a set of sequences of instructions, one sequence per processor. Each sequence in the set captures the *program order* at that processor. An *execution* of this program is obtained by running the shared memory program on an MP system. Formally, an execution is the above set of sequences of instructions with each *load* labeled with its returned value. Every instruction in an execution can be decomposed into one or more *events* (in the former case, we shall use the words 'instruction' and 'event' interchangeably). Rule *Read Value* indicates what data value a load event in an *execution* should return. Rule *Per Processor Order* constrains how any two events from the same processor should be ordered in their "logical" total order relation. Finally, Rule *Write atomicity* requires all store events to appear to be visible to all processors instantaneously.
We define an *execution* to obey a memory model if all the memory events of the *execution* form at least one logical total order which obeys the *Per Processor Order* and is consistent with the *Read Value*. '$\rightarrow$' indicates the total order relation (often referred to as "logical total order" or "logical order"). The logical order of completion of events may not be the same as the temporal order of completion. To illustrate this distinction, consider a processor that allows *local bypassing, i.e*, early retirement of matching loads by directly reading from the store buffer. Let this processor also implement memory fences[1] aggressively by marking existing entries in the store buffer, and flushing the marked entries only just prior to subsequent coherent transactions. In such a processor, an instruction sequence `store(a); fence; load(a)` may be carried out in temporal order as *store(a).local; load(a); store(a).global.* In other words, the `load` appears to have finished "globally" before the `store` finishes globally. However, in a logical explanation, the `store(a).global` must precede the `load(a)` due to the presence of a memory fence between them. Other examples of this situation arise in speculative implementation schemes where, following a `store` that misses in the local cache, a *speculative* `load` that hits in the local cache may be entertained [?].
Each instruction t is defined as a tuple (p,l,o,a,d) where

- $p(t)$: processor in whose program t originates from.
- $l(t)$: label of instruction t in p's program .
- $o(t)$: operation type
- $a(t)$: memory address
- $d(t)$: data

If there are two instructions t_1, t_2 then $l(t_1) < l(t_2)$ means that t_1 appears before t_2 in program order.

[1] Any event issued after a fence should appear to complete only after any event that was issued before the fence.

3 Hierarchy of Memory Models

An executable specification (operational model) of any memory model can be visualized as a correct and simple 'implementation' whose every run satisfies the memory model definition and also every run that satisfies the memory model can be generated by the executable specification. The selection of this simplified 'implementation' depends on the memory model under examination. In this section we categorize memory models into four classes and show how common data stuctures to design the operational model can be derived for memory models belonging to a particular memory model class, thus providing a systematic approach to deriving the operational model. The four classes of memory models are as follows::

1. *Strong*: requires *Write atomicity* and does not allow local bypassing. (e.g. Sequential Consistency, IBM-370).
2. *Weak*: requires *Write atomicity* and allows local bypassing (e.g. Ultra Sparc TSO, PSO and RMO, Alpha)
3. *Weakest*: does not require *Write atomicity* and allows local bypassing (e.g.PC, PowerPC).
4. *Hybrid*: supports weak load and store instructions that come under memory model *Weakest* and also support strong load and store instructions that come under *Strong* or *Weak* memory model categories. (e.g. Itanium, DASH-RC, RC_{PC}).

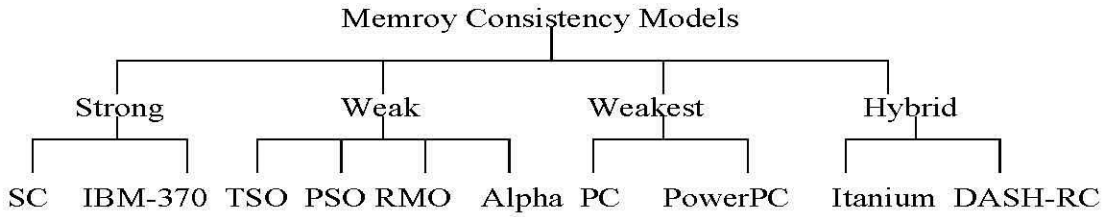

Figure 1. The four classes of memory models

Memory models that are weaker than the *Strong* category do not usually require that an *execution* have all its load and store instructions form a total order as a part of their specification.

In our work, we adopt a style of specifying even these memory models to adopt the approach of identifying a logical total order such as $\rightarrow$ described earlier. In other words, even in case of these weaker memory models, by splitting loads and stores to finer events, we define a logical total order of all the memory events.

Depending upon the category a memory model falls under, we split a store instruction into one or more events. Load instructions for any memory model can always be treated as a single event. Here are a few examples of splitting events. In case of Sequential Consistency, we do not split even the stores as sequential consistency demands a single global total order of the loads and stores. For a weak memory model such as the Ultra Sparc TSO, we split the store instruction into two events, a local store event (which means that the store is only visible to the processor who issued it) and a global event (which means that the store event is visible to all processors). Since the *Weakest* category of memory models lack write atomicity, we need to split stores into $p+1$ events, where p is number of processors, thus ending up with a local store event and p global events (global event i would mean that the store event is visible to processor i).

We now look at each of the four classes of memory models in details.

3.1 *Strong* memory models:

For specifying memory consistency models under this category, every load and store instruction is kept unsplit. So we can view every instruction to be an event itself.

3.1.1 Memory ordering rules

1. *Read Value:* Let t_1 be a load instruction . Then the data value of t_1 is the data value of the "most recent" store instruction t_2 to the same memory location as t_1 in the total order relation $\rightarrow$. i.e

   (a) $o(t_2) = st, a(t_1) = a(t_2), t_2 \rightarrow t_1$ and

   (b) there does not exist a store instruction t_3 s.t $a(t_1) = a(t_3)$ and $t_2 \rightarrow t_3 \rightarrow t_1$.

2. *Per processor order:* Let t_1 and t_2 be two instructions s.t $p(t_1) = p(t_2)$, $l(t_1) < l(t_2)$. *Per processor order* rules constitutes of four sub-rules.

   (a) $ld \rightarrow ld$: $o(t_1) = ld, o(t_2) = ld$.

   (b) $ld \rightarrow st$: $o(t_1) = ld, o(t_2) = st$.

   (c) $st \rightarrow ld$: $o(t_1) = st, o(t_2) = ld$.

   (d) $st \rightarrow st$: $o(t_1) = st, o(t_2) = st$.

   (e) fence: there exists a fence instruction t_f s.t $l(t_1) < l(t_f) < l(t_2)$.

   For any sub-rule that holds true for a memory model, $t_1 \rightarrow t_2$.
   Sometimes a sub-rule holds only per memory location, and in that case we explicitly specify that sub-rule to hold *per memory location* (e.g. if $ld \rightarrow st$ holds only when $a(t_1) = a(t_2)$ then we say that $ld \rightarrow st$ *per memory location* holds).
   We use concise representations to express the list of sub-rules that holds for a memory model(e.g if both $ld \rightarrow ld$ and $ld \rightarrow st$ sub-rules are satisfied then we can represent that memory model to satisfy $ld \rightarrow X$ where X means any memory instruction)

3. *Write Atomicity* All the store events form a single totat order in a way that every store instruction appear to be visible atomically to all processors.

3.1.2 Examples of *Strong* Memory Models :

An *execution* satisfies a *Strong* memory model if there exists at least one logical total order of all memory events in that *execution* that obeys *Read Value* and *Write Atomicity* rules and also obeys zero or more sub-rules of the Per processor rules depending upon that memory model.
The sub-rules of Per processor rules that are applicable to the following memory models are as follows :

1. Sequential Consistency: All subrules except *fence* holds(SC is defined in absence of fence instructions). Hence we can concisely say that $X \rightarrow X$ holds.

2. IBM-370: Sub-rules $ld \rightarrow X$, $st \rightarrow st$, $st \rightarrow ld$ *per memory location* and fence holds.

3.2 *Weak* memory models:

Every store instruction is split into two memory events . Hence a store instruction $t = (p, l, st, a, d)$ is split into $t^{local} = (p, l, st_{local}, a, d)$ and $t^{global} = (p, l, st_{global}, a, d)$, both of which have the same data value. Each load instruction may get its *Read Value* from a t^{local} for which the corresponding t^{global} has not yet occurred. The goal in this case is to model "local bypassing" i.e to model a store buffer bypassing where stores enters the store buffer as t^{local} when its visible to *only* the processor that issued it and exit with a t^{global} when its visible to all processors. $t^{local} \rightarrow t^{global}$ always.

3.2.1 Memory ordering rules:

1. *Read Value*: Let t_1 be a load instruction . Then the data value of t_1 is the data value of the "most recent" store event split from the store instruction t_2,to the same memory location as t_1 in the total order relation $\rightarrow$. i.e

   (a) if

i. $a(t_1) = a(t_2)$, $t_2^{local} \rightarrow t_1 \rightarrow t_2^{global}$ and

ii. there does not exist a store instruction t_3 s.t $p(t_1) = p(t_3)$, $a(t_1) = a(t_3)$, $t_2^{local} \rightarrow t_3^{local} \rightarrow t_1$.

(b) else if

i. $a(t_1) = a(t_2)$, $t_2^{global} \rightarrow t_1$ and

ii. there does not exist a store instruction t_3 s.t $a(t_1) = a(t_3)$, $t_2^{global} \rightarrow t_3^{global} \rightarrow t_1$.

(c) else, t_1 receives the initial value 0.

2. Per processor order Let t_1 and t_2 be two events s.t $p(t_1) = p(t_2)$,$l(t_1) < l(t_2)$. There are six possible sub-rules as follows:

(a) $ld \rightarrow ld$: $o(t_1) = ld$, $o(t_2) = ld$.

(b) $ld \rightarrow st$: $o(t_1) = ld$, $o(t_2) = st_{global}$.

(c) $st \rightarrow ld$: $o(t_1) = st_{global}$, $o(t_2) = ld$.

(d) $st \rightarrow st$: $o(t_1) = st_{global}$, $o(t_2) = st_{global}$.

(e) $fence$: there exists a fence instruction t_f s.t $l(t_1) < l(t_f) < l(t_2)$.

(f) Memory Data Dependence: This sub-rule pertains to instrcutions involving the same memory location. Hence, for $a(t_1) = a(t_2)$,

i. $ld \rightarrow ld$: $o(t_1) = ld$, $o(t_2) = ld$.

ii. $ld \rightarrow st$: $o(t_1) = ld$, $o(t_2) = st_{local}$.

iii. $st \rightarrow ld$: $o(t_1) = st_{local}$, $o(t_2) = ld$.

iv. $st \rightarrow st$: $o(t_1) = st_{local}$, $o(t_2) = st_{local}$.

For any sub-rule that holds true for a memory model, $t_1 \rightarrow t_2$. Whenever we use "X" we refer to it as ld or st instruction.

3. *Write Atomicity* All the store events t where $o(t) = st_{global}$, form a single totat order in a way that every store event appear to be visible atomically to all processors.

3.2.2 Examples of *Weak* Models :

An *execution* satisfies a memory model in this category if there exists at least one logical total order of all memory events in that *execution* that obeys *Read Value* and *Write Atomicity* rules and obeys zero or more sub-rules of the *Per processor order* rules depending upon that memory model.
The sub-rules of Per processor rules that are applicable to the following memory models are as follows :

1. TSO: Sub-rules $ld \rightarrow X$, $X \rightarrow st$, *Memory Data Dependence* and fence holds.
2. PSO: Sub-rules $ld \rightarrow X$, *Memory Data Dependence* and fence holds.
3. RMO: Sub-rules *Memory Data Dependence* and fence holds.
4. Alpha: Sub-rules *Memory Data Dependence* and fence holds. Note that Alpha has two kinds of fences, MB which is equivalent to our definition of fence and $MB - WW$ which is applicable between store instuctions only ($MB - WW$ can be defined in a simlar way as MB was defined).

3.3 *Weakest* memory models:

Every store instruction is split into $p + 1$ events where p is the number of processors. Thus t where $o(t) = st$ is split into t^{local} where $o(t^{local}) = st_{local}$ and $t^1, .t^k ... t^p$ where $o(t^k) = st^k_{global}$ and $t^{local} \rightarrow t^{p(t)} \rightarrow t^k, \forall k (1 \leq k \leq p$ and $k \neq p(t))$. Since *Weakest* memory models do not require *Write Atomicity*, a store instrcution may be visible to one processor earlier or later than when its visible to another processor. Hence the need to identify when a store event is visible to all processors at different times where t^k event corresponds to when the store event t is visible to processor k.

3.3.1 Memory Ordering Rules

1. *Read Value*: Let t_1 be a load instruction . Then the data value of t_1 is the data value of the "most recent" store event split from the store instruction t_2, to the same memory location as t_1 in the total order relation $\rightarrow$. i.e
   (a) if
      - $p(t_1) = p(t_2)$, $a(t_1) = a(t_2)$, $t_2^{local} \rightarrow t_1 \rightarrow t_2^{global}$ and
      - there does not exist a store instruction t_3 s.t $p(t_1) = p(t_3)$, $a(t_1) = a(t_3)$, $t_2^{local} \rightarrow t_3^{local} \rightarrow t_1$.

   (b) else if
      - $a(t_1) = a(t_2)$, $t_2^{p(t_1)} \rightarrow t_1$ and
      - there does not exist a store instruction t_3 s.t $a(t_1) = a(t_3)$, $t_2^{p(t_1)} \rightarrow t_3^{p(t_1)} \rightarrow t_1$.

   (c) else, t_1 receives the initial value 0.

2. *Per processor order*: Let t_1 and t_2 be two events(can be $ld.acq, st^{local}, st^k$ for any $k \in$ processors) s.t $p(t_1) = p(t_2)$, $l(t_1) < l(t_2)$. There are six possible sub-rules.
   (a) $ld \rightarrow ld$: $o(t_1) = ld$, $o(t_2) = ld$.
   (b) $ld \rightarrow st$: $o(t_1) = ld$, $o(t_2) = st^{p(t)}$.
   (c) $st \rightarrow ld$: $o(t_1) = st^{p(t)}$, $o(t_2) = ld$.
   (d) $st \rightarrow st$: $o(t_1) = st^k$, $o(t_2) = st^k$ for all processors k.
   (e) fence: there exists a fence instruction t_f s.t $l(t_1) < l(t_f) < l(t_2)$ and $o(t_1) = o(t_2) = st^{p(t)}$.
   (f) Memory Data Dependence: This sub-rule pertains to instructions involving the same memory location. Hence, for $a(t_1) = a(t_2)$,
      i. $ld \rightarrow ld$: $o(t_1) = ld$, $o(t_2) = ld$.
      ii. $ld \rightarrow st$: $o(t_1) = ld$, $o(t_2) = st_{local}$.
      iii. $st \rightarrow ld$: $o(t_1) = st_{local}$, $o(t_2) = ld$.
      iv. $st \rightarrow st$: $o(t_1) = st_{local}$, $o(t_2) = st_{local}$.

      If any one of the above sub-rules hold then $t_1 \rightarrow t_2$. Again, if we use "X" then we refer to it as ld or st instruction.

3. *Coherence*: All this time we never referred to *Coherence* as for *Strong* and *Weak* memory models *Coherence* is a subrule of *Write Atomicity*. However now we need to define *Coherence* separately simply because *Weakest* memory models does not support *Write Atomicity* but may or may not support *Coherence*.
   Coherence requires that if t_1 and t_2 be two store instructions s.t $a(t_1) = a(t_2)$ then
   (a) If $p(t_1) = p(t_2)$, $l(t_1) < l(t_2)$, then $t_1^k \rightarrow t_2^k \forall$ processors k.
   (b) If $t_1^p \rightarrow t_2^p$ for some processor p then $t_1^k \rightarrow t_2^k \forall$ processors k.

4. *Write Atomicity*: Although *Weakest* memory models do not require *Write Atomicity* we will still define it when store instructions are split to $p + 1$ instructions, its importance will be apparent in the next section where we will define *Hybrid* models.
   Here we have two cases as follows
   (a) case 1: If a store instruction t is split into $p + 1$ events i.e $t^{local}, t^1, ...t^p$ then they all appear to occur atomically i.e in the total order relation if $t^i \rightarrow t^j$ where $i, j \in$ {local,1,..,p} and if $t^i \rightarrow$t'$\rightarrow t^j$ then t' can only be $\in \{t^{local}, t^1, .., t^p\}$. Thus, all the events of t are ordered in a way s.t no instruction that does not belong to an event of t can come in between.
   (b) case 2: If a store instruction t is split into $p + 1$ instructions i.e $t^{local}, t^1, ...t^p$ then except t^{local} they all appeat to occur atomically i.e in the total order relation if $t^i \rightarrow t^j$ where $i, j \in$ {local,1,..,p} and if $t^i \rightarrow$t'$\rightarrow t^j$ then t' can only be $\in \{t^1, .., t^p\}$.

Note that the definition of case 1 is same as the definition of *Write Atomicity* for *Strong* memory models and the definition of case 2 is same as the definition of WA for *Weak* memory models where they only differ in how store instructions are split.

3.3.2 Examples of *Weakest* Memory Models

An *execution* satisfies a memory model in this category if there exists at least one logical total order of all memory events in that *execution* that obeys *Read Value* , may or may not satisfy *Coherence* and obeys zero or more sub-rules of the Per processor rules depending upon that memory model.
The sub-rules of Per processor rules that are applicable to the following memory models are as follows :

1. PC: Sub-rules $ld \rightarrow X$, $X \rightarrow st$, *Memory Data Dependence* and fence holds.
2. PowePC: Sub-rules *Memory Data Dependence* and fence holds.

3.4 *Hybrid* memory models:

All memory models that support more than one kind of load and store operations where all *executions* containing only the strong load and store operations obey a memory model under *Strong* or *Weak* memory model classes and all *executions* containing only the weak load and store operations obey a memory model under *Weakest* memory model classes.
In such models where both strong and weak store operations exist there can be two ways to view these store instructions as follows:

- split the strong store operations t into t^{local} and t^{global} and split the weak store operations into $p+1$ events i.e $t^{local}, t^1, .., t^p$.
- split both the weak and strong store operations into $p+1$ events i.e $t^{local}, t^1, .., t^p$.

Although breaking up the strong store operations into $p+1$ operations seems redundant in order to define their properties but to retain uniformity and to be able to explain the interaction between weak and strong store operations more clearly the latter method seems more logical. Hence every strong and weak store operations are split into $p+1$ events as explained before . The memory ordering rules like *Read Value*,*Per processor order*,*Coherence*,*Write Atomicity* are same as defined for *Weak* memory ordering rules. However, a reference to ld or st instruction corresponds to weak load and store instructions respectively, ld.acq and st.rel refers to strong load and store instructions respectively, unlike as for *Weakest* memory models where ld and st corresponds to one unique type of load ans store instruction. Consequently, for example, a sub-rule $ld \rightarrow st^{local}$ of per processor rules defined for *Weakest* memory models corresponds to $ld.acq \rightarrow st^{local}$, $ld.acq \rightarrow st.rel^{local}$, $ld \rightarrow st^{local}$, and $ld \rightarrow st.rel^{local}$ subrules for *Hybrid* memory models where each of these four sub-rules are defined similar to $ld \rightarrow st^{local}$ sub-rule for *Weakest* memory models .
If for a rule or sub-rule the type of load or store instruction is not mentioned then that rule or sub-rule is applicable to both types of load and store instructions respectively. "X" would refer to $ld, ld.acq, st^{local}$, or $st.rel^{local}$ event.

3.4.1 Examples of *Hybrid* Memory Models

1. $Itanium_{TM}$: An *execution* obeys Itanium memory model if there exists a logical total order of all memory events (ld,st,ld.acq,st.rel,fence) that obeys *Read Value*, *Coherence*, *Write Atomicity*(case 2) for all st.rel store events and *Per processor order* rules which include subrules $ld.acq \rightarrow X$, $X \rightarrow st.rel$, *Memory Data Dependence* except $ld \rightarrow ld$ and fence. Note that *Coherence* is defined irrespective of whether a store instruction is strong or weak.

4 Automatic generation of executable specification

4.1 Strong and Weak memory models

The operational semantics of the Generalized Weak memory model is described in terms of three data structures (see Figure 1), and how each instruction tuple t that is issued updates these data structures and/or returns the read value, as per Table 1. By 'buffer' we mean an unbounded structure in which the entries maintain their arrival order as in a FIFO, but entries may be removed from anywhere provided a removal condition is satisfied. The oldest entry is always at the head and the youngest at the tail. Initially, all buffers are empty. The data structure elements are:

Event	*Guard*	*Actions*
$ld(t)$ (hit)	$\exists$ youngest $t^{'} \in WOB_{p(t)} : a(t^{'}) = a(t) \wedge d(t^{'}) = d(t)$	
$ld(t)$ (miss)	$\neg\exists\ t^{'} \in WOB_{p(t)} : a(t^{'}) = a(t)$	Issue($RB_{p(t)}$, t)
$ld(t)$ (miss) *contd.*	t$\in RB_{p(t)} \wedge$ Allowed($RB_{p(t)}, t) \wedge\ M[a(t)] = d(t)$	Delete($RB_{p(t)}, t$)
$st_{local}(t)$	True	Issue($WOB_{p(t)}, t$)
$st_{global}(t)$	$t \in WOB_{p(t)} \wedge$ Allowed($WOB_{p(t)}, t$)	$M[a(t)] \leftarrow d(t)$; Delete($WOB_{p(t)}, t$);
$Fence(t)$	True	Flush(t)

Table 1. Transition System

1. a single port memory M that spans the entire address-space and holds word-sized data in each location. M is updated when the $MW(t)$ event of Table 1 fires, which removes an entry from $WOB_{p(t)}$ writes into M. Initially, each location of M carries data 0.

2. a write out re-order buffer WOB_i into which st instructions are enqueued. When the $st_{global}(t)$ event of Table 1 fires and $p(t) = i$, the entry t is removed from WOB_i and atomically copied into M.

3. a re-order load buffer RB_i into which ld instructions are enqueued when event ($ld(t)$) of Table 1 fires. Eventually, t is removed from RB_i, and data $d(t)$ corresponding to this tuple gets returned.

4.2 State Transition Rules

Table 1 defines the operational semantics. The first column shows *Event*s that happen if the *Guard* condition in the second column is true, performing the *Actions* shown in the last column. At any time, any one of the eligible events may be picked in a *fair* manner. Each event happens when the next instruction t is issued by processor $p(t)$ ($ld(t)$,$st(t)$,$Fence(t)$). Notice that in case of event $ld(t)$, tuple t carries the data $d(t)$ being returned (following the convention used in [?]). When these events fire, a constraint expressed in the Guard field shows what this data is. We use $=$ for equality testing, and $\leftarrow$ for assignment.
$ld(t)$: We seek an entry $t^{'}$ in $WOB_{p(t)}$ such that $a(t^{'}) = a(t)$, and $t^{'}$ is the youngest such entry, if multiple entries exist. If $t^{'}$ exists (hit), the returned data $d(t)$ is the same as $d(t^{'})$. If no such entry exists (miss), t is enqueued into $RB_{p(t)}$ via $Issue(RB_{p(t)}, t)$. Eventually, the $ld(t)$ event completes by being serviced by M which provides the data $d(t)$. Its guard 'Allowed' captures when tuple t, which is present in $RB_{p(t)}$, can be processed ahead of all the other tuples within $RB_{p(t)}$.
$st_{local}(t)$: results in t being enqueued into $WOB_{p(t)}$ via procedure $Issue$.
$st_{global}(t)$ updates the memory array M from $WOB_{p(t)}$. Its guard 'Allowed' captures when tuple t, which is present in $WOB_{p(t)}$, can be processed ahead of all the other tuples within $WOB_{p(t)}$.
$Fence(t)$ is carried out by procedure Flush, which flushes every pending $RB_{p(t)}$ entry, every $WOB_{p(t)}$ entry, where the entry comes from $p(t)$ and occurs earlier than t in program order. The functions used in the transition system are now described.
Allowed($WOB_{p(t)}$,t): The function evaluates to true if the following conditions are satisfied as follows:

1. $\neg\exists t^{'} \in WOB_{p(t)}$ s.t $l(t') < l(t)$ and
   (a) sub-rule $st \rightarrow st$[2] or,
   (b) $(a(t_1) = a(t_2) \wedge$ *Memory Data Dependence* sub-rule $st \rightarrow st$).
2. $\neg\exists t^{'} \in RB_{p(t)}$ s.t $l(t') < l(t)$ and
   (a) sub-rule $ld \rightarrow st$ or,
   (b) $(a(t_1) = a(t_2) \wedge$ *Memory Data Dependence* sub-rule $ld \rightarrow st$)

Allowed($RB_{p(t)}$,t): The function evaluates to true if the following conditions are satisfied as follows:

[2]indicates that the following condition is true if the memory model requires the sub-rule to hold

1. $\neg\exists t^{'} \in RB_{p(t)}$ s.t $l(t') < l(t)$ and
   (a) sub-rule $ld \rightarrow ld$ or,
   (b) $(a(t_1) = a(t_2) \wedge$ *Memory Data Dependence* sub-rule $ld \rightarrow ld)$
2. $\neg\exists t^{'} \in WOB_{p(t)}$ s.t $l(t') < l(t)$ and
   (a) sub-rule $st \rightarrow ld$ or,
   (b) $(a(t_1) = a(t_2) \wedge$ *Memory Data Dependence* sub-rule $ld \rightarrow st)$

Issue$(Buffer, t)$: Add t to the tail of Buffer queue.
Delete$(Buffer, t)$: This procedure deletes t wherever it may be in Buffer.
Although we designed a Generalized Weak memory model, Strong memory models can also be designed with the same data structures and operational semantics except that now a load instruction cannot hit the WOB buffer (local bypassing) due to Strong memory models not supporting local bypassing.

4.3 Weakest and Hybrid memory models

The operational semantics of the Generalized Weakest memory model is described in terms of five data structures (see Figure 1), and how each instruction tuple t that is issued updates these data structures and/or returns the read value, as per Table . Initially, all buffers are empty. The data structure elements are:

Event	*Guard*	*Actions*
$ld(t)$ (hit)	$\exists$ youngest $t^{'} \in WOB_{p(t)} : a(t^{'}) = a(t) \wedge d(t^{'}) = d(t)$	
$ld(t)$ (miss)	$\neg\exists\ t^{'} \in WOB_{p(t)} : a(t^{'}) = a(t)$	Issue($RB_{p(t)}$, t)
$ld(t)$ (miss) *contd.*	t$\in RB_{p(t)} \wedge$ Allowed$(RB_{p(t)}, t) \wedge\ M_{p(t)}[a(t)] = d(t)$	Delete$(RB_{p(t)}, t)$
$st_{local}(t)$	True	Issue$(WOB_{p(t)}, t)$
$st_{global}(t)$	$t \in WOB_{p(t)} \wedge$ Allowed$(WOB_{p(t)}, t)$	$\forall$ processors i Issue(WIB_i, t); Delete$(WOB_{p(t)}, t)$;
$st_{global}(t)$ *contd.*	$t \in WIB_{p(t)} \wedge$ Allowed(WIB_i, t)	$M_i[a(t)] \leftarrow d(t)$; Delete$(WIB_i, t)$;
$Fence(t)$	True	Flush(t)

Table 2. Transition System

1. memory M_i per processor i that spans the entire address-space and holds word-sized data in each location. M_i is updated when the st_{global} *contd.* event of Table 1 fires, which removes an entry from $WIB_{p(t)}$ and writes into M_i. Initially, each location of M_i carries data 0.
2. a write out re-order buffer WOB_i into which st instructions are enqueued. When the $st_{global}(t)$ event of Table 1 fires, the entry t is removed from WOB_i and atomically copied into WIB_i for every processor i.
3. a re-order load buffer RB_i into which ld instructions are enqueued when event $(ld(t))$ of Table 1 fires. Eventually, t is removed from RB_i, and data $d(t)$ corresponding to this tuple gets returned.
4. a write in re-order buffer WIB_i into which st instructions are enqueued. When the $st_{global}(t)$ *contd.* event of Table 1 fires, the entry t is removed from WIB_i and atomically copied into M_i.

4.4 State Transition Rules

Table 2 defines the operational semantics.
$ld(t)$: We seek an entry $t^{'}$ in $WOB_{p(t)}$ such that $a(t^{'}) = a(t)$, and $t^{'}$ is the youngest such entry, if multiple entries exist. If $t^{'}$ exists (hit), the returned data $d(t)$ is the same as $d(t^{'})$. If no such entry exists (miss), t is enqueued into $RB_{p(t)}$ via $Issue(RB_{p(t)}, t)$. Eventually, the $ld(t)$ event completes by being serviced by $M_p(t)$ which provides the data $d(t)$. Its guard 'Allowed' captures when tuple t, which is present in $RB_{p(t)}$, can be processed ahead of all the other tuples within $RB_{p(t)}$.

$\overline{st_{local}(t)}$: results in t being enqueued into $WOB_{p(t)}$ via procedure $Issue$.
$\overline{st_{global}(t)}$ updates the memory array M_i from $WIB_{p(t)}$. Its guard 'Allowed' captures when tuple t, which is present in $\overline{WIB_{p(t)}}$, can be processed ahead of all the other tuples within $WIB_{p(t)}$.
$\overline{Fence(t)}$ is carried out by procedure Flush, which flushes every pending $RB_{p(t)}$ entry, every $WOB_{p(t)}$ entry, where the entry comes from $p(t)$ and occurs earlier than t in program order. The functions **Allowed**($WOB_{p(t)}$), **Allowed**($RB_{p(t)}$), **Issue**($Buffer, t$) and **Delete**($Buffer, t$) used in the transition system are same as that of the Generalized Weak memory model. Function Allowed($WIB_{p(t)}$ is now described.
Allowed($WIB_{p(t)}$,t): The function evaluates to true if $\neg\exists t' \in WIB_{p(t)}$ s.t $l(t') < l(t)$ and sub-rule $st \rightarrow st$ or $a(t_1) = a(t_2)$ and *Data Dependence* sub-rule $st \rightarrow st$ is satisfied by the memory model.
Although we designed a Generalized Weakest memory model, Hybrid memory models can also be designed with the same data structures and operational semantics but now due to multiple load and store instruction types, there will be additional rules pertaining to these new instructions, for example, reordering between st and $st.rel$ in WOB and WIB. The operational model for Itanium has been designed [?] and can be referred to understand how our Generalized Weakest memory model can be easily extended to handle Hybrid memory models.

5 Conclusion