WOMM: A Weak Operational Memory Model

Arnab De[1], Abhik Roychoudhury[2], and Deepak D'Souza[1]

[1] Department of Computer Science and Automation,
Indian Institute of Science, Bangalore, India
{arnabde,deepakd}@csa.iisc.ernet.in
[2] School of Computing, National University of Singapore, Singapore.
abhik@comp.nus.edu.sg

Abstract. Memory models of shared memory concurrent programs define the values a read of a shared memory location is allowed to see. Such memory models are typically weaker than the intuitive sequential consistency semantics to allow efficient execution. In this paper, we present WOMM (abbreviation for Weak Operational Memory Model) that formally unifies two sources of weak behavior in hardware memory models: reordering of instructions and weakly consistent memory. We show that a large number of optimizations are allowed by WOMM. We also show that WOMM is weaker than a number of hardware memory models. Consequently, if a program behaves correctly under WOMM, it will be correct with respect to those hardware memory models. Hence, WOMM can be used as a formally specified abstraction of the hardware memory models. Moreover, unlike most weak memory models, WOMM is described using operational semantics, making it easy to integrate into a model checker for concurrent programs. We further show that WOMM has an important property - it has sequential consistency semantics for datarace-free programs.

1 Introduction

With the push towards multicore platforms in the coming years, the importance of concurrent programming is steadily growing. Increasingly, it is felt that computer architects will be moving towards processors with many cores and concurrent/parallel programming will become much more mainstream.

In general, programmers tend to find parallel/concurrent programming harder than sequential programming, owing to the many possible executions of a program for any given input. Shared-memory concurrent programming languages describe the semantics of concurrency via a *Memory Model* — it describes which writes can be visible to a program read operation. Programmers intuitively consider the memory model of a concurrent program as Sequential Consistency (SC) [1], where each process proceeds in program order and the operations from each process are interleaved and a read sees the last write to the same memory location in that interleaving. Although the SC semantics is easy for the programmers to understand, it does not allow many compiler and hardware optimizations.

As a simple example, one may consider the program fragment in Figure 1. Throughout the paper, we use `x`, `y`, ... to denote shared memory locations and `r1`, `r2`, ... to denote local variables/ registers.

Initially, `x == y == 0`

Proc 1:	Proc 2:
`x = 1;`	`r1 = y;`
`y = 1;`	`r2 = x;`

`r1 == 1, r2 == 0` not allowed by the SC

Fig. 1. Not allowed by SC - I

Initially, `x == 0`

Proc 1:	Proc 2:	Proc 3:
`x = 1;`	`r1 = x;`	`r3 = x;`
`x = 2;`	`x = 3;`	`x = 4;`
	`r2 = x;`	`r4 = x;`

`r1 == r4 == 1, r2 == r3 == 2` not allowed by SC

Fig. 2. Not allowed by SC - II

In this example, we would normally assume `r1` $\leq$ `r2` at the end of the program. In other words, SC does not allow `r1 == 1` $\wedge$ `r2 == 0` since `y` is set after `x` in Proc 1. However, Proc 1 may reorder the writes at runtime (as evidenced by hardware multiprocessor memory models such as Sparc Partial Store Order (PSO) [2]), thereby making the result `r1 == 1` $\wedge$ `r2 == 0` possible at the end of the program.

Similarly, the behavior demonstrated in the example from Figure 2 cannot be produced by an SC execution. Nevertheless, the interconnection network may cause Proc 3 to see the writes from Proc 1 out of order, but Proc 2 may see them in the program order, causing this behavior.

To accommodate such optimizations, weaker or more relaxed memory models have been proposed. Such memory models typically allow more behaviors than the SC. Such intricacies in the memory model makes understanding concurrent program behavior particularly difficult.

Conventionally *hardware multiprocessor memory models* — such as Total Store Order (TSO), Partial Store Order (PSO) and Relaxed Memory Order (RMO) of Sun Sparc [3, 2] have been specified using a *reordering table*, denoting whether two instructions accessing two different memory locations can be reordered at runtime. Similarly, the official memory models for IA-32 and AMD-64 architectures are defined as a set of informal rules [4, 5]. Recently, a rigorous, axiomatic memory model [6] has been proposed for x86 multiprocessors that matches the informal documentations. These memory models differ from each other in subtle but complex ways. Moreover, the specifications are either informal or axiomatically defined, making them difficult to follow. As a result, a compiler writer or a low-level programmer may not get an abstract view of the underlying hardware memory models.

We have observed that weak behavior in multiprocessors is mostly caused by two features. A processor can reorder instructions accessing different memory locations (as in Figure 1) or writes to the same memory location may be visible to different processors in different orders (as in Figure 2). In this paper, we present formal semantics of a Weak Operational Memory Model, referred as WOMM, that combines these two features. We use a language that contains some of the essential features of multiprocessor programs. We use a simple operational

semantics to specify our memory model, making it easy to follow for the compiler writers and low-level programmers. The operational style of our specification allows easy integration to program analysis tools such as model checkers. We demonstrate that a number of optimizations are allowed by WOMM. We show that WOMM is weaker than the Location Consistency (LC) [7] memory model, which, in turn, makes it weaker than a number of memory models such as TSO, PSO, RMO etc. Consequently, any execution allowed by these hardware memory models will be allowed by WOMM. As a result, if a compiler assumes WOMM as the underlying hardware memory model, the compilation will be correct for all these hardware memory models. Finally, we show that WOMM has an important property — it guarantees sequential consistency for datarace-free programs.

2 Program Model

We consider a simple and abstract program model for the sake of presentation. A multiprocessor system consists of a finite number of processors, each with an unique id. All the processors have a finite number of local registers. Memory is shared among all the processors.

The program consists of one executable per processor. Each executable has a finite number of static instructions, with designated first and last instructions. In the rest of the paper, we identify the executables by the processors they run on. Instructions can be categorized into following types. We use the following conventions: `r`, `r1`, ... represent registers, `[r]` a shared memory location whose address is equal to the contents of `r`, *exp* a local expression and `L` a program label.

Read (Rd) (`r1 = [r2]`): Reads the content of the memory location `[r2]` to the register `r1`.

Write (Wr) (`[r1] = r2`): Writes the value of the register `r2` to the memory location `[r1]`.

Local computation (Loc) (`r =` *exp*): Assigns the value of the expression *exp*, which does not have any memory access, to the register `r`.

Conditional goto (Cond) (`if r goto L`): If the register `r` has a non-zero value, program counter is set to the `L`, otherwise to the next program location.

Unconditional goto (Jmp) (`goto L`): Sets program counter to the `L`.

Acquire (Acq) (`acquire [r]`): Acquires the memory location `[r]`. Once a processor executes acquire on a memory location, no other processor can acquire that location till the former processor *releases* it. Moreover, it works as a one-way barrier — instructions after an acquire cannot be reordered before it at runtime.

Release (Rel) (`release [r]`): Releases the memory location `[r]` enabling other processors to acquire it. It also works as a one-way barrier — instructions before a release cannot be reordered after it at runtime.

A *synchronization instruction* is either an acquire or a release instruction. Note that a large number of synchronization primitives found in real hardwares can

be closely modeled with these two types of instructions. The acquire and release instructions are close to the load-acquire/ store-release instructions of the Itanium architecture [8]. A two-way memory barrier instruction can be modeled by an acquire instruction immediately followed by a release instruction on the same memory location, where there is one memory location per processor designated for this purpose. An atomic operation involving access to a single memory location (such as test-and-set and compare-and-exchange) can be modeled as acquire instruction on that memory location, followed by instructions implementing the atomic instruction, followed by a release instruction on the same memory location (assuming that memory location is not accessed in any other way).

In the rest of the paper, short names of the instructions in the above list represent the set of all possible instructions of the corresponding type and the representative form given along is used to represent an arbitrary instruction of that type. The word *instruction* may refer to a static instruction or a dynamic, runtime instance of a static instruction. depending on the context. We also use the term *action* to refer to dynamic instructions.

3 Operational Semantics

In this section we formally describe the operational semantics of *WOMM*, a weak abstract memory model for shared memory multiprocessors. Informally, instructions are *issued* in program order and put into an *instruction queue*. An instruction can *commit* any time after all the instructions it depends on commit. Each memory location, instead of containing a single value, contains the set of all accesses to that location along with some *causal order* among them. A read can return any value from this list under certain consistency restrictions.

3.1 Structure of States

Given a program P, a program state Ω consists of a *global state* Θ and one *local state* Φ_p for each processor p.

A global state Θ consists of the following components:

- A map Π from memory locations to *memory states*. Informally, a memory state in our semantics contains all the accesses to that memory location along with the *causal orders* among them. Formal description of a memory state is given below.
- A map L from memory locations to *lock states*. A lock state may contain either a processor id, denoting the location is acquired by the processor with id, or 0, denoting the location is free.

A *memory state* is a *partially ordered multiset* (*pomset*), denoted by $(S, \preceq)$, where S is the multiset and $\preceq$ is the partial order. A multiset S is a collection of *pomset items*. Being a multiset, it allows duplication of elements. Each item is added by some instruction. The pomset items can be of following types. Here

val denotes a value, *pid* denotes a the processor id performing the corresponding write/ release/ acquire action and *loc* denotes a memory location.

1. A *write item* (WI) which is a $\langle val, pid \rangle$ pair.
2. A *read item* (DI) which is simply $\langle pid \rangle$.
3. A *release item* (RI) which is a $\langle loc, pid \rangle$ pair.
4. An *acquire item* (AI) which is again a $\langle loc, pid \rangle$ pair.

The relation $\preceq$ is a partial order, i.e. a reflexive, antisymmetric and transitive binary relation on S. The corresponding irreflexive relation is denoted by $\prec$. Whenever this relation is updated (e.g. in Figure 3 and Figure 6), we assume that the new relation is reflexively and transitively closed to maintain the partial order property.

As the same local register name can be used in different unrelated instructions in a processor, it may create false dependencies among those instructions. For example, if two consecutive instructions assign to the same local register, there may not be any actual data flow between them, but they still cannot be reordered naïvely as a future read of that register may get affected by the reordering. To avoid such false dependency and facilitate reordering of instructions, local registers are renamed at runtime. A single local register can be associated with different dynamic registers at different points of the execution and each dynamic register can have different values. We assume that there are infinite number of dynamic registers available.

A local state Φ_p consists of the following components:

- A map μ_p that maps static register names of to their current dynamic names.
- A map σ_p that maps dynamic registers and the program counter to the values they contain. The domain of values includes a special value `undef`.
- An instruction queue Q_p, which contains dynamic instructions (actions) that have been issued but not yet committed (see Section 3.2). Actions are ordered by the order of their issue in the queue. Registers are renamed with dynamic names for the actions residing in the queue (see Section 3.4).
- A map δ_p that maps an issued action to the pomset item corresponding to the action if the instruction is a read or a write or a synchronization instruction.

Initial State: In the initial state, all the maps in the local states of the processors and the instruction queue are empty (i.e. returns `undef` for any input) except the program counter which points to the first instructions of every executable. All memory locations are free. All memory states contain a special write element, containing an arbitrary initial value of the location, denoted by $WI\langle val, \top \rangle$.

3.2 Execution

An execution of a program is a total order of *operations* where each operation is an *issue* or a *commit* of an action. An operation can occur if all its premises are

met and causes some state transition. The premises and effects of operations are described in Section 3.4 and Section 3.5. . The total order among operations in an execution is referred as the *occurs-before* relation (denoted by $\xrightarrow{ob}$).

If i is an action, $issue(i)$ and $commit(i)$ denote the corresponding issue and commit operations. Similarly, if op is an operation, $inst(op)$ denotes the corresponding dynamic instruction. The state transition from Ω to Ω' caused by an operation op from processor p is denoted by $\langle p : op, \Omega\rangle \rightarrow \Omega'$.

Execution of a program starts with issue of the first instruction from the any arbitrary processor. For each processor, the execution starts with the issue of the first instruction from that processor. A processor *finishes execution* when the last instruction, along with all previously issued instructions, of that processor is committed. The program *finishes execution* when all processors finish execution.

A *trace* is a sequence of operations, denoted as $\langle op_1, op_2, ...\rangle$. As occurs-before is a total order, any execution in WOMM can be expressed as a trace in a natural way. An execution represented as a trace is referred to as a *trace execution*.

3.3 Complete Execution and Observable State

We call an execution *complete* if all issued instructions are committed. Note that a complete execution might not be a finished execution.

The ultimate goal of our semantics is to define, given a closed program, the set of *observable states* reachable from an initial state, via a complete execution. An *observable state* is the value of the local registers and program counters of each processor. If p is a processor and $\mathtt{r}$ is register of p, the *value* of $\mathtt{r}$ is given by $\sigma_p(\mu_p(\mathtt{r}))$.

Note that an implementation obeying WOMM need not execute the instructions from different processors in a total order, but the observable state after executing a set of instructions by the implementation must be reachable via a complete trace execution of exactly the same set of instructions.

3.4 Semantics of Issue

Issues of instructions must be done in *program order*. As a consequence, the issue operations update the program counters. For the sake of simplicity, we do not show this requirement and effect in the formal rules. Issue of any instruction by a processor p also has the following effects:

- Renames the registers. If a register $\mathtt{r}$ is read in the instruction, it is renamed to $\mu_p(\mathtt{r})$. If it is assigned to, it is renamed with a new dynamic name and μ_p is updated to reflect that and $\sigma_p(\mu_p(r))$ is set to `undef`. It is important to follow the order as the same static register can be both read and assigned in a single instruction. This operation is denoted as *Rename*.
- Appends the renamed instruction to the instruction queue Q_p.

If there is an uncommitted *Cond* instruction issued from a processor, the next instruction to be issued is undefined. Hence an instruction can be issued by

a processor only if there is no *Cond* instruction in the corresponding instruction queue. Similarly, if the value of a dynamic register is `undef`, a read or write instruction dereferencing that register cannot be issued.

If the instruction is a read/write instruction, then the corresponding read/ write item is added to the pomset corresponding to the memory location involved in that instruction. The write items do not have the value defined.If the instruction is a synchronization instruction, then the corresponding release/ acquire item is added to all the pomsets. To mark that the synchronization instruction has not committed, the memory location in the corresponding release/ acquire item is kept undefined. The newly added items are ordered after the existing items from the same processor. The operation of adding an item I to a pomset P during issue is denoted by *IssueUpdate* and is formally defined in Figure 3, where *Pid* maps an item to the corresponding processor. The map δ_p is updated accordingly.

$$
\begin{aligned}
&\mathit{IssueUpdate} \equiv \lambda I.\lambda P.(S', \preceq') \\
&\text{where} \\
&\qquad P \;= (S, \preceq) \\
&\qquad S' \;= S \cup \{I\} \\
&\qquad \preceq' \;= \preceq \cup \{(e, I) \mid e \in S \\
&\qquad\qquad \wedge(\mathit{Pid}(e) = \mathit{Pid}(I) \vee \mathit{Pid}(e) = \top)\}
\end{aligned}
$$

Fig. 3. Issue Update

Figure 4 shows the formal rules for issue by an arbitrary processor p. *Issue-Rd*, *Issue-Wr*, *Issue-Acq*, *Issue-Rel* and *Issue-Gen* are the rules for issue of read, write, acquire, release and other types of instructions, respectively. $\Omega[C_1; \ldots; C_n]$ denotes a state which is equal to Ω except the changes $C_1, \ldots, C_n$, in that order. Any change C_i can have two forms: $M|M'$ denotes the state component M is modified to M' and if M is a map, $M\{x \rightarrow v\}$ denotes that $M(x)$ is updated with the value v. In the quantifiers, x ranges over the set of memory locations. If l is the address of a memory location, $[l]$ denotes the corresponding memory location. The list append operation is denoted by ".".

3.5 Semantics of Commit

Actions in the queue can be committed out-of-order, as long as it follows the following ordering restrictions:

- Synchronization actions must be committed in program order.
- Any action must be committed after any local acquire action issued before it.
- Any action must be committed before any local release action issued after it.
- Any read action can commit only if there is no local uncommitted write to the same memory location issued before it.
- Any action reading a register can be committed only if the value of all registers read in the action are not `undef`.

(Issue-Rd)

$$\frac{\begin{array}{c} inst \equiv \texttt{r1} = [\texttt{r2}] \quad Cond \cap Q_p = \emptyset \quad I = DI\langle p \rangle \\ l = \sigma_p(\mu_p(\texttt{r2})) \neq \texttt{undef} \quad P = IssueUpdate(I, \Pi([l])) \end{array}}{\langle p : issue(inst), \Omega \rangle \rightarrow \Omega[Q_p | Q_p.Rename(inst);\ \Pi\{[l] \rightarrow P\};\ \delta_p\{inst \rightarrow I\}]}$$

(Issue-Wr)

$$\frac{\begin{array}{c} inst \equiv [\texttt{r1}] = \texttt{r2} \quad Cond \cap Q_p = \emptyset \quad I = WI\langle *, p \rangle \\ l = \sigma_p(\mu_p(\texttt{r2})) \neq \texttt{undef} \quad P = IssueUpdate(I, \Pi([l])) \end{array}}{\langle p : issue(inst), \Omega \rangle \rightarrow \Omega[Q_p | Q_p.Rename(inst);\ \Pi\{[l] \rightarrow P\};\ \delta_p\{inst \rightarrow I\}]}$$

(Issue-Acq)

$$\frac{\begin{array}{c} inst \equiv \texttt{acquire}[\texttt{r}] \quad Cond \cap Q_p = \emptyset \quad I = AI\langle *, p \rangle \\ \Pi' = \forall x : \Pi\{x \rightarrow IssueUpdate(I, \Pi(x))\} \end{array}}{\langle p : issue(inst), \Omega \rangle \rightarrow \Omega[Q_p | Q_p.Rename(inst);\ \Pi | \Pi';\ \delta_p\{inst \rightarrow I\}]}$$

(Issue-Rel)

$$\frac{\begin{array}{c} inst \equiv \texttt{release}[\texttt{r}] \quad Cond \cap Q_p = \emptyset \quad I = RI\langle *, p \rangle \\ \Pi' = \forall x : \Pi\{x \rightarrow IssueUpdate(I, \Pi(x))\} \end{array}}{\langle p : issue(inst), \Omega \rangle \rightarrow \Omega[Q_p | Q_p.Rename(inst);\ \Pi | \Pi';\ \delta_p\{inst \rightarrow I\}]}$$

(Issue-Gen)

$$\frac{inst \notin Rd \cup Wr \cup Sync \quad Cond \cap Q_p = \emptyset}{\langle t : issue(inst), \Omega \rangle \rightarrow \Omega[Q_p | Q_p.Rename(inst)]}$$

Fig. 4. Rules for Issue of an Instruction

– An acquire action can commit only if the corresponding memory location is not acquired or acquired by the same processor. A release action can commit if the memory location is already acquired by the same processor.

Informally, the effects of a commit on the program state are as follows. Here p refers to the processor performing the operation and *inst* refers to the dynamic instruction corresponding to the operation. Note that the registers mentioned here are the dynamic ones, already renamed during issue.

Read (`r1 = [r2]`): Updates $\sigma_p(\texttt{r1})$ with the value read. The value read can be any value from the set of write items returned by the function *ConsistentRead*, defined in Figure 5, where x is the memory location read, I is the corresponding read item (given by $\delta_p(inst)$) and *Val* returns the value associated with a write item. This function returns the set of complete write items such that there is no intervening write item between any returned write item and the read item in the pomset order and any returned write item is not ordered after the read item. Note that the relation $\preceq$ is closed transitively after each update (see Section 3.1). The read cannot commit if the returned set is empty.

Write (`[r1] = r2`): Completes the corresponding write item (given by $\delta_p(inst)$) with the value of the local register (given by $\sigma_p(\texttt{r2})$).

$$
\begin{aligned}
&\mathit{ConsistentRead} \equiv \\
&\lambda x.\lambda I.\{e \mid \Pi(x) = (S, \preceq) \\
&\qquad \wedge\, e \in S \wedge\ e \in WI \\
&\qquad \wedge\, \nexists e' \in WI\colon e \prec e' \prec I \\
&\qquad \wedge\, \neg(I \prec e) \\
&\qquad \wedge\ \mathit{Val}(e) \neq *\}
\end{aligned}
$$

Fig. 5. Consistent Read

$$
\begin{aligned}
&\mathit{CommitUpdate} \equiv \\
&\lambda I.\lambda P.(S', \preceq') \text{ where} \\
&\quad P\ = (S, \preceq) \\
&\quad S' = S \\
&\quad \preceq' = \preceq \cup \{(e, I) \mid e \in S \\
&\qquad\qquad\qquad \wedge e \in RI \\
&\qquad\qquad\qquad \wedge \mathit{MemLoc}(e) = \mathit{MemLoc}(I)\}
\end{aligned}
$$

Fig. 6. Commit Update

Local computation (`r = exp`): Evaluates the local expression on RHS using the values given by σ_p and updates $\sigma_p(\mathtt{r})$. The evaluation of the expression is denoted by *Evaluate* function. If any of the local registers on RHS has value `undef`, *Evaluate* returns `undef`. Otherwise it follows the natural semantics of expression evaluation.

Acquire (`acquire [r]`): Completes the corresponding acquire item (given by $\delta_p(inst)$) with $[\sigma_p(\mathtt{r})]$.[3] Also modifies the lock state of the memory location $[\sigma_p(\mathtt{r})]$.

Release (`release [r]`): Completes the corresponding release item (given by $\delta_p(inst)$) with $[\sigma_p(\mathtt{r})]$. Also modifies the lock state of the memory location $[\sigma_p(\mathtt{r})]$.

Commit of any acquire action also updates the partial order of memory states. It makes the corresponding acquire item ordered after all previously committed release items on the same memory location. This operation, referred as *CommitUpdate*, is defined in Figure 6, where I is the acquire item, P is the pomset and *MemLoc* maps an acquire/release item to the memory location associated with it.

Cond and *Jmp* actions do not change the state, except removing themselves from the instruction queue.

Figures 7 show formal rules for committing an action. Note that the registers in these instructions are already renamed to dynamic names. We do not show the obvious requirement that the committing instruction must be present in the instruction queue and after it is committed, it is removed from the queue. $\mathit{Pre}(q, i)$ denotes the set of actions in instruction queue q which were issued before the action i and $\mathit{Post}(q, i)$ denotes the set of actions in instruction queue q issued after i, where $i \in q$. $\mathit{Wr}(\mathtt{x})$ denotes the set of write actions to the memory location `x`. We follow all the conventions defined in Section 3.4.

3.6 Abstract Execution

Although the trace execution semantics (Section 3.2) helps in generating and traversing executions of a program, it is sometimes useful to view the executions

[3] Note that the update of pomset item by commit of a synchronization instruction affects all pomsets as the synchronization item is added to all the pomsets during issue.

(Commit-Rd)

$$\frac{\begin{array}{c} inst \equiv \texttt{r1} = [\texttt{r2}] \quad l = \sigma_p(\texttt{r2}) \quad WI\langle v, p'\rangle \in ConsistentRead([l], \delta_p(inst))) \\ Pre(Q_p, inst) \cap Acq = \emptyset \quad Pre(Q_p, inst) \cap Wr([l]) = \emptyset \end{array}}{\langle p\colon commit(inst), \Omega\rangle \rightarrow \Omega[\sigma_p\{\texttt{r1} \rightarrow v\}]}$$

(Commit-Wr)

$$\frac{inst \equiv [\texttt{r1}] = \texttt{r2} \quad l = \sigma_p(\texttt{r1}) \quad v = \sigma_p(\texttt{r2}) \neq \texttt{undef} \quad Pre(Q_p, inst) \cap Acq = \emptyset}{\langle p\colon commit(inst), \Omega\rangle \rightarrow \Omega[\delta_p\{[l] \rightarrow WI\langle v, p\rangle\}]}$$

(Commit-Loc)

$$\frac{inst \equiv \texttt{r} = exp \quad v = Evaluate(\sigma_p, exp) \neq \texttt{undef} \quad Pre(Q_p, inst) \cap Acq = \emptyset}{\langle p\colon commit(inst), \Omega\rangle \rightarrow \Omega[\sigma_p\{\texttt{r} \rightarrow v\}]}$$

(Commit-Acq)

$$\frac{\begin{array}{c} inst \equiv \texttt{acquire [r]} \quad l = \sigma_p(\texttt{r}) \quad (L([l]) = \langle 0\rangle \;\; or \;\; L([l]) = \langle p\rangle) \\ L' = L\{[l] \rightarrow \langle p\rangle\} \\ \delta'_p = \delta_p\{inst \rightarrow AI\langle [l], p\rangle\} \\ \Pi' = \forall x : \Pi\{x \rightarrow CommitUpdate(\delta'_p(inst), \Pi(x))\} \\ Pre(Q_p, inst) \cap Sync = \emptyset \end{array}}{\langle p\colon commit(inst), \Omega\rangle \rightarrow \Omega[L|L';\ \delta_p|\delta'_p;\ \Pi|\Pi']}$$

(Commit-Rel)

$$\frac{\begin{array}{c} inst \equiv \texttt{release [r]} \quad l = \sigma_p(\texttt{r}) \quad L([l]) = \langle p\rangle \\ L' = L\{[l] \rightarrow \langle 0\rangle\} \\ \delta'_p = \delta_p\{inst \rightarrow RI\langle [l], p\rangle\} \\ Pre(Q_p, inst) = \emptyset \end{array}}{\langle p\colon commit(inst), \Omega\rangle \rightarrow \Omega[L|L';\ \delta_p|\delta'_p]}$$

Fig. 7. Rules for Committing an Instruction

as *abstract executions*, in the style of [9], where the actions are related by few causal orders. In this section we show how a complete WOMM trace execution can be naturally translated to an abstract execution.

More specifically, an abstract execution is a tuple $\langle P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow}\rangle$, where P is the program, A is the set of actions executed, $\overset{po}{\rightarrow}$ is the program order, $\overset{so}{\rightarrow}$ is the synchronization order, W is the write seen function, V is the value written function, $\overset{sw}{\rightarrow}$ is the synchronizes-with relation and $\overset{hb}{\rightarrow}$ is the happens-before relation. We define each component of the tuple in context of a complete WOMM execution as below:

- P is the closed program.
- A is the set of actions issued and committed. Note that in a complete execution, all issued actions are committed.
- Given two actions i and i', $i \overset{po}{\rightarrow} i'$ iff $issue(i) \overset{ob}{\rightarrow} issue(i')$ and $Pid(i) = Pid(i')$.
- Given two synchronization actions i and i', $i \overset{so}{\rightarrow} i'$ iff $commit(i) \overset{ob}{\rightarrow} commit(i')$.

- $W(r) = w$, if the read action r reads the value written by the write action w.
- $V(w) = v$, if the write action w writes the value v.
- A release action i *synchronizes-with* an acquire action i' if $MemLoc(i) = MemLoc(i')$ and $i \xrightarrow{so} i'$.
- Happens-before relation is the transitive closure of synchronizes-with and program order relations.

It should be noted that there can be multiple complete trace executions corresponding to a single abstract execution. All such executions lead to the same final observable state.

4 Relaxed Behaviors Allowed by WOMM

Initially, `x == y == 0`

Proc 1:	Proc 2:
`r1 = x;`	`r2 = y;`
`y = 1;`	`x = 1;`

`r1 == r2 == 1` is allowed by WOMM

Fig. 8. Behavior Allowed by WOMM

A large class of hardware optimizations can be described as reordering of *locally independent actions*, simply referred as *independent actions*. In the code fragment from Figure 8, the processors can reorder the writes before the reads as they access different memory locations, resulting in the behavior described. WOMM allows this behavior in the trace execution $\langle issue(\texttt{r1} = \texttt{x}), issue(\texttt{r2} = \texttt{y}), issue(\texttt{y} = \texttt{1}), issue(\texttt{x} = \texttt{1}), commit(\texttt{y} = \texttt{1}), commit(\texttt{x} = \texttt{1}), commit(\texttt{r1} = \texttt{x}), commit(\texttt{r2} = \texttt{y})\rangle$. Note that the semantics allow this trace. Before the commit of read of x in Proc 1, the pomset of `x` contains three items — two write items $WI\langle 0, \top\rangle$, $WI\langle 1, 2\rangle$ and a read item $DI\langle 1\rangle$. The relations in the pomset are as follows: $\{WI\langle 0, \top\rangle \prec WI\langle 1, 2\rangle, WI\langle 0, \top\rangle \prec DI\langle 1\rangle\}$. From the definition of *ConsistentRead* (Figure 5), the read can see the value `1`. Similarly, the read of `y` can also see the value `1`, thus making this behavior possible. The behavior in Figure 1 is also allowed by WOMM in a similar way.

We present the formal definition of independent actions as follows.

Definition 1 (Independent Actions). *Let s_1 and s_2 be two actions from the same processor p in an execution such that $s_1 \xrightarrow{po} s_2$. They are* independent *if all of the following are true:*

1. *s_1 and s_2 are not conflicting accesses.*
2. *There is no data flow from s_1 to s_2 through a register.*
3. *s_1 is not an acquire.*
4. *s_2 is not a release.*
5. *Both s_1 and s_2 are not synchronization instructions.*
6. *s_1 is not a conditional branch.*

The following theorem states that a processor can reorder independent instructions at runtime under WOMM semantics.

Theorem 1. *Let at any point of an execution E, s be an action issued by a processor p but not yet committed. The action s can commit immediately if all the uncommitted actions by processor p are independent with s.*

Proof. Proof of this theorem follows directly from the definition of independent actions and the semantics of commit (Figure 7). □

The memory subsystems and the communication networks of relaxed multiprocessor platforms often do not guarantee strict ordering of "events". In the example of Figure 2, the writes by Proc 1 are seen in different orders by the reads of Proc 2 and Proc 3. WOMM allows this behavior by committing the writes from Proc 1 before any reads. As Proc 1 is not synchronized with Proc 2 or Proc 3, the reads can see any of the writes from Proc 1. Note that this behavior cannot be produced by simply reordering independent actions.

In Section 5 we show that WOMM is weaker than the traditional hardware memory models such as TSO, PSO, RMO etc. This implies that WOMM allows all runtime optimizations allowed by those models.

5 Relationship with Other Memory Models

In this section, we show that WOMM is strictly weaker than the Location Consistency (LC) memory model [7]. As the LC is weaker than many common hardware memory models (such as TSO, PSO [2], Release Consistency [10]), this in turn, makes WOMM weaker than those traditional memory models. Hence, any runtime optimizations allowed by the these hardware memory models are also allowed by WOMM.

Theorem 2. *WOMM is strictly weaker than the Location Consistency memory model.*

Proof. As the LC semantics do not allow explicit reordering of instructions, an LC execution [7] can be viewed as a trace execution, where an issue of an instruction is immediately followed by the commit of the same and the ordering of those instructions is consistent with the program and synchronization order. The write and the synchronization instructions update the pomset the same way as WOMM's *IssueUpdate* (Figure 3) and *CommitUpdate* (Figure 6) functions. A read instruction in LC gets its value from the pomset, restricted by the following constraints — the write item is either not ordered before the read, or there is no other intervening write item in the pomset relation. As the instructions are not reordered and the execution respects all uniprocessor control dependencies, it is not possible for a read to see a write item which is ordered after the read. Hence the constraints imposed on read by the LC are equivalent to the *ConsistentRead* function. Hence it is easy to see that this trace is allowed by the WOMM. As a result, given a program, any observable state reachable under the LC semantics is also reachable under WOMM.

The example in Figure 8 shows that WOMM is strictly weaker than the LC. In the LC, as at least one read must commit before the writes with value `1`, both

`r1` and `r2` cannot be `1` simultaneously. But this is possible in WOMM as the writes with value `1` can commit first followed by the reads and both the reads can see the value `1`. □

6 The DRF Guarantee

Designers of memory models face two conflicting goals — on one hand, the semantics should be strong enough so that it is easy for the programmers to understand it (e.g. the SC); on the other hand, it should be weak enough to allow different optimizations. A compromise between these two goals is reached via the *DRF guarantee*.

Definition 2 (Datarace-Free Program). *In an execution, a pair of instructions accessing the same shared-memory location are called* conflicting accesses *if at least one of them is a write. A (closed) program is called* datarace-free *(DRF) or* correctly synchronized *if all pairs of conflicting accesses in any sequentially consistent execution of that program are ordered by* happens-before.

Definition 3 (DRF Guarantee). *Relaxed memory models with the* DRF guarantee *ensure that any execution of a datarace-free program under the relaxed memory model is equivalent to some sequentially consistent execution of that program.*

This property is highly desirable for weak memory models as it ensures that programmers need not worry about the weak memory models as long as they write only correctly synchronized programs. In this section, we prove that WOMM gives the DRF guarantee. To show that WOMM has this property, we first state the following lemma in context of an abstract execution allowed by WOMM.

Lemma 1. *In a WOMM execution of a correctly synchronized program, if each read sees a write that happens-before the read, then the execution has sequentially consistent behavior.*

This lemma is proved in [9] for the Java Memory Model. The same proof can be used in the context of an abstract WOMM execution.

The following lemma states a connection between the relation of pomset items at any state of a WOMM trace execution and the happens-before relation of the abstract execution equivalent to the trace.

Lemma 2. *Consider a WOMM trace execution E. Let I and I' be two complete pomset items, corresponding to the instructions i and i' respectively, belonging to the same memory location at any state Ω during E. Let E_A be the abstract execution equivalent to E. $I \prec I'$ in Ω iff $i \overset{hb}{\rightarrow} i'$ in E_A.*

This lemma is proved in Appendix A.

Now we prove the following theorem stating that WOMM provides the DRF guarantee.

Theorem 3. *Any abstract execution of a correctly synchronized program allowed by WOMM is equivalent to a sequentially consistent execution.*

Proof. Let E_A be an abstract execution of a correctly synchronized program P allowed by WOMM and E be a trace execution of P equivalent to E_A. By Lemma 1, if E_A is not equivalent to any SC execution of P, there must be at least one read that sees a write that does not happen-before it. Using Lemma 2, when such a read commits in E, the corresponding read item I_D will not be related to the write item I_W it sees. Let r be the first such read, which sees a unrelated write w, to commit in E. Using Lemma 2, all reads committed before r see writes that happen-before them.

We construct a complete trace execution E' from E the following way. Let A' be the set of last committed instructions from each processor (in *ob* order) when r is committed in E. We then include in A' all instructions of E that precede those instructions in program order. Some of such instructions may not be committed when r is committed in E. In E', those instructions are committed in program order such that each read sees a write that happens-before it (that is, the corresponding read item is ordered after the seen write item in the pomset). Note that none of these incomplete instructions can be an acquire instruction, according to the semantics of commit. Hence, instructions from different processors can be committed in any order consistent with program order, without affecting the happens-before relations. At last, the read r is modified such that it also sees a write that happens-before it. Note that, this completion does not affect the already committed instructions, neither does it introduce any new *sw* edge.

Let us consider the execution E'_A corresponding to the complete execution E'. In E'_A, each read sees a write that happens-before it. Hence, by Lemma 1, it must be equivalent to an SC execution, but there is a write w and a read r to the same memory location which are not related by happens-before. This contradicts the fact that the program is correctly synchronized. Thus, there cannot exist any read in E_A which sees a write that does not happen-before the read. Hence, by Lemma 1, the execution E_A is equivalent to an SC execution. □

7 Related Work

Most of the early works in relaxed memory models for hardware multiprocessors has been summarized in [3]. Shen et al [11] shows that breaking an instruction into finer-grained operations and using an abstract semantic cache can lead to weak memory models. The notion of abstract shared memory state used in the Location Consistency memory model [7] is very similar to our memory model. A recent work [6] developed rigorous axiomatic semantics of multiprocessor machine code for x86 architectures.

Some recent works have stressed the need for precise, concrete, operational style specification of memory models [12, 13]. Boudol et al [14] shows how operational specification of memory models help in proving the DRF guarantee for such models.

Verifying concurrent programs under relaxed memory models requires precise specification of the memory model. Burckhardt et al [15] exhaustively check all executions of a concurrent program under a relaxed memory model that allows certain reordering and buffering. In our earlier works, we formalized the memory model for C# [16] and proposed an operational approximation of the Java Memory Model [17] and developed memory model sensitive model checkers to find bugs in concurrent C# and Java programs.

8 Conclusion and Future Work

In this paper, we have presented a weak operational multiprocessor memory model. Our work is useful for understanding such models at an abstract level. The compiler writers and low-level programmers can follow this semantics without going into the details for each hardware memory model separately. We have shown that our proposed semantics is weak enough to allow many hardware optimizations. We formally compared it with the Location Consistency model. The operational style of our specification enables traversal and enumeration of executions of a given program that are allowed by the model, helping in easy integration into model checkers.

In future, we plan to extend our memory model to also allow speculative execution of instructions. We also plan to integrate the current model to state space exploration tools (such as software model checkers) for memory model sensitive program reasoning.

References

1. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9) (1979) 690–691
2. David L. Weaver and Tom Germond, Prentice Hall Publishers: The SPARC Architecture Manual : Version 9. (1994)
3. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer **29**(12) (1996) 66–76
4. Intel: Intel 64 and ia-32 architectures software developer's manual volume 3a: System programming guide. `http://www.intel.com/Assets/PDF/manual/253668.pdf`
5. AMD: Amd64 architecture programmer's manual (2007)
6. Sarkar, S., Sewell, P., Nardelli, F.Z., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86-cc multiprocessor machine code. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2009) 379–391
7. Gao, G.R., Sarkar, V.: Location consistency — a new memory model and cache consistency protocol. IEEE Trans. Comput. **49**(8) (2000) 798–813
8. Intel: A formal specification of intel itanium processor family memory ordering. `http://www.intel.com/design/itanium/downloads/251429.htm` (October 2002)

9. Manson, J., Pugh, W., Adve, S.V.: The java memory model. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2005) 378–391
10. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors. In: Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on. (May 1990) 15–26
11. Shen, X., Arvind, Rudolph, L.: Commit-reconcile & fences (crf): a new memory model for architects and compiler writers. SIGARCH Comput. Archit. News **27**(2) (1999) 150–161
12. Nardelli, F.Z., Sewell, P., Sevcik, J., Sarkar, S., Owens, S., Maranget, L., Batty, M., Alglave, J.: Relaxed memory models must be rigorous. In: Exploiting Concurrency Efficiently and Correctly Workshop. (2009)
13. Mador-Haim, S., Alur, R., , Martin, M.M.: Specifying relaxed memory models for state exploration tools. In: Exploiting Concurrency Efficiently and Correctly Workshop. (2009)
14. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2009) 392–403
15. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2007) 12–21
16. Huynh, T.Q., Roychoudhury, A.: Memory model sensitive bytecode verification. Form. Methods Syst. Des. **31**(3) (2007) 281–305
17. De, A., Roychoudhury, A., D'Souza, D.: Java memory model aware software validation. In: PASTE '08: Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, New York, NY, USA, ACM (2008) 8–14

A Proof of Lemma 2

Proof. If $I \prec I'$ at some state Ω in E, then, from the definition of *IssueUpdate* (Figure 3) and *CommitUpdate* (Figure 6), either i and i' are from the same processor and $issue(i) \overset{ob}{\rightarrow} issue(i')$, or there is a sequence of synchronization instructions $s_1, s'_1, \ldots, s_n, s'_n$ such that each s_j and s'_j are release and acquire instructions on the same memory location respectively, s'_j and s_{j+1} are from the same processor, $issue(i) \overset{ob}{\rightarrow} issue(s_1) \overset{ob}{\rightarrow} commit(s_1) \overset{ob}{\rightarrow} commit(s'_1) \ldots commit(s'_n) \overset{ob}{\rightarrow} commit(i')$ and $issue(s'_n) \overset{ob}{\rightarrow} issue(i')$ (i might be same as s_1 and s'_n might be same as i'). From Section 3.6, it implies that in E_A, $i \overset{po}{\rightarrow} s_1 \overset{sw}{\rightarrow} s'_1 \ldots s'_n \overset{po}{\rightarrow} i'$. From the definition of happens-before relation, $i \overset{hb}{\rightarrow} i'$.

Conversely, if $i \overset{hb}{\rightarrow} i'$, either $i \overset{po}{\rightarrow} i'$, or there are instructions $i_1, i'_1, \ldots, i_n, i'_n$ such that $i \overset{po}{\rightarrow} i_1 \overset{sw}{\rightarrow} i'_1 \overset{po}{\rightarrow} i_2 \ldots i_n \overset{sw}{\rightarrow} i'_n \overset{po}{\rightarrow} i'$, where each i_j is a release and i'_j is an acquire action. In the first case, clearly $I \prec I'$. In the second case, from the semantics of commit, when i' commits, all these instruction must already be

committed. From *IssueUpdate* and *CommitUpdate*, the pomset items of each of these instructions are ordered after the pomset item of the previous instruction. Hence, $I \prec I'$. □