An Operational Model for Multiprocessors with Caches

Salil Joshi and Sanjiva Prasad

Indian Institute of Technology Delhi
salil.ssj@gmail.com,
sanjiva@cse.iitd.ac.in

Abstract. Modern multiprocessors are equipped with local caches, to enhance program performance. However, the presence of caches can lead to the violation of *sequential consistency* [7] assumptions regarding program order and write atomicity. With respect to such *relaxed memory models* [1], we provide an operational description of program execution (in the style of [4]) that accounts for cache effects. In particular, we provide an operational characterization of cache invalidation and update policies and an abstract characterization of cache consistency. The programming model consists of a simple imperative language extended with common synchronization primitives such as locks or barrier instructions. The main results show that by precluding certain data races or by placing certain synchronization constraints, sequentially consistent behavior can be obtained for multiprocessor execution even in the presence of local caches.

1 Introduction

While shared memory multiprocessor systems are becoming increasingly common today, writing correct concurrent programs for such systems remains a challenge. Program behavior is determined by a *memory model*. Programmers commonly assume a model of memory that is *sequentially consistent* [7], i.e., all memory accesses appear to occur atomically in some total order, and those issued by any given processor occur as specified by the program order.

One feature for improving performance, found in all modern processors, is a *cache*. The presence of caches in a *multiprocessor* system can lead to violations of program order and write atomicity assumptions [1]. The goal of this work is to understand the effect of caches on concurrent program execution with respect to a weaker memory model where these assumptions are relaxed [2], and to discover constraints under which sequentially consistent behavior is guaranteed.

We follow the approach of Boudol and Petri [4] in presenting an *operational* model of memory and describe the execution of programs written in a simple imperative language with respect to a sequentially consistent model ("the specification"), and then with respect to a relaxed model ("the implementation"). We then consider some *synchronization primitives* (called "safety nets" in [1]) supported by the model; these are instructions used to temporarily force program order or write atomicity in order to make the program behavior more manageable. In particular, we show that for *locks* a *well synchronizedness condition*

C.S. Calude and V. Sassone (Eds.): TCS 2010, IFIP AICT 323, pp. 371–385, 2010.

(equivalent to *data-race-freedom*), and for *barrier instructions*, a *multiple-race-free barrier condition*, are sufficient to ensure sequentially consistent behavior in an otherwise non-sequentially consistent system.

Technically, this is done by establishing precise correspondences (e.g., "bisimilarity") between the specification and implementation behaviors. The consequence of these results is that for programs satisfying these constraints (which are stated at the *specification* level), the programmer need only consider the more intuitive set of sequentially consistent executions rather than all possible executions, when reasoning about program behavior.

A novel contribution in this paper is an *abstract operational characterization* of a memory model, general enough to express multiprocessor memory with local caches as a particular instance, in terms of a small set of operational properties. The theorems are thus proven for any memory model exhibiting these properties. We believe that our implementation semantics closely resemble actual processor architectures (with caches). At the same time, the semantics abstract over processor specific details like cache replacement policies, cache consistency protocols etc. Thus our models (and hence our theorems) should hold for a wide variety of multiprocessor systems. We give an example of a *cache-based system* which exhibits the required properties, and which allows the following relaxations with respect to the classification of relaxed memory models in [1]:

1. **W** $\rightarrow$ **R**: Reordering of a write with a following read to a different variable.
2. **W** $\rightarrow$ **W**: Reordering of a write with a following write to a different variable.
3. **Read other's write early**: This violates write atomicity.
4. **Read own write early**: This violates both write atomicity and program order.

Our approach differs from that of Boudol and Petri [4] in several ways: (i) while they present a higher-order language with ML-style imperative features, dynamic thread creation and scoped locks, we prefer a simple imperative language that we believe has greater applicability; (ii) in addition to locks, we consider synchronization primitives such as barrier or fence instructions; (iii) while Boudol *et al* consider write buffers, we believe we consider a more general multiprocessor model with caches, while being able to deal with a variety of cache management policies (update, invalidation). Furthermore our operational characterization is presented in terms of abstract properties.

The rest of this paper is organised as follows. §2 presents the language and the specification semantics. The abstract characterization of the implementation semantics is given in §3 as a collection of properties on the operational relation. §4 introduces scoped locks as a synchronization primitive and shows sequential consistency can be ensured by data race freedom; similarly a barrier condition is shown to achieve this when using barrier instructions (§5). In §6, we present an intuitive model of caches which satisfies the abstract properties of §3.2. §7 concludes the paper with some directions for future work. Proofs of lemmas and theorems are omitted in this paper but can be found in [6].

2 The Language

We employ a simple imperative language, which will later be extended with two different synchronization primitives.

$$\begin{array}{lll} \langle e\rangle ::= \langle int\rangle & \langle b\rangle ::= \textbf{true} \mid \textbf{false} & \langle C\rangle ::= \langle var\rangle := \langle e\rangle \\ \quad \mid \langle e\rangle \oplus \langle e\rangle & \quad \mid \neg \langle b\rangle \mid \langle b\rangle \wedge \langle b\rangle & \quad \mid \langle C\rangle; \langle C\rangle \\ \quad \mid \langle var\rangle & \quad \mid \langle e\rangle \leq \langle e\rangle & \quad \mid \textbf{if } \langle b\rangle \textbf{ then } \langle C\rangle \textbf{ else } \langle C\rangle \\ & & \quad \mid \textbf{while } \langle b\rangle \textbf{ do } \langle C\rangle \end{array}$$

All variables are integer valued, boolean values are used only for tests and the $\oplus$ operator is any one of $+, *, \ldots$ We also have a runtime marker, (), which is 'returned' when a command is executed. It is used only to make the operational semantics easier to formulate.

2.1 Specification Semantics

A *configuration* $\mathbb{C}$ is a pair (S, P), where S is the *store* (main memory) that is shared across the system, and P is a list of *processes*. Each process runs on its own processor. The store is common to all, and is a simple mapping from variable names to integers. For convenience, we will use $\mathbb{C}.S$ and $\mathbb{C}.P$ to refer to the store and program respectively of a specific configuration $\mathbb{C}$.

Transitions involve reducing a *redex* in an *evaluation context* [10]. An evaluation context consists of a number i (indicating that the process P_i is being executed), and a one-hole context. We use $P_i[\boldsymbol{E}[]]$ to denote the hole $\boldsymbol{E}[]$ in the i^{th} process in P.

The redexes and one-hole contexts ($\boldsymbol{E}[]$) are as follows:

$$\begin{array}{l} \langle bval\rangle ::= \textbf{true} \mid \textbf{false} \\ \langle redex\rangle ::= \langle int\rangle \oplus \langle int\rangle \\ \quad \mid \langle int\rangle \leq \langle int\rangle \\ \quad \mid \neg \langle bval\rangle \mid \langle bval\rangle \wedge \langle bval\rangle \\ \quad \mid \langle var\rangle \mid \langle var\rangle := \langle int\rangle \\ \quad \mid \textbf{if } \langle bval\rangle \textbf{ then } \langle C\rangle \textbf{ else } \langle C\rangle \\ \quad \mid \textbf{while } \langle b\rangle \textbf{ do } \langle C\rangle \mid ();\langle C\rangle \end{array} \qquad \begin{array}{l} \langle \boldsymbol{E}\rangle ::= [] \mid \boldsymbol{E} \oplus \langle e\rangle \mid \langle int\rangle \oplus \boldsymbol{E} \\ \quad \mid \boldsymbol{E} \leq \langle e\rangle \mid \langle int\rangle \leq \boldsymbol{E} \\ \quad \mid \neg \boldsymbol{E} \mid \boldsymbol{E} \wedge \langle b\rangle \mid \langle bval\rangle \wedge \boldsymbol{E} \\ \quad \mid \langle var\rangle := \boldsymbol{E} \mid \boldsymbol{E};\langle C\rangle \\ \quad \mid \textbf{if } \boldsymbol{E} \textbf{ then } \langle C\rangle \textbf{ else } \langle C\rangle \end{array}$$

The operational semantics are given in Figure 1. We have left out the obvious transitions such as those for the arithmetic and boolean operators.

Transitions are *decorated* as: $\xrightarrow{(a,i)}$. Here i is used to indicate that the transition is for process P_i, and a denotes the action being carried out. The possible actions are: τ (reduction which does not involve the store), $\mathtt{rd}_x^v$ (the value v is read from variable x) and $\mathtt{wr}_x^v$ (the value v is written to the variable x). For reads and writes we will use $\mathtt{rd}_x$ and $\mathtt{wr}_x$ when we do not care what value was read/written. Concurrent, conflicting transitions are said to form a race:

Definition 1. *In a sequence of transitions* $\mathbb{C}_0 \xrightarrow{(a_0,i_0)} \cdots \xrightarrow{(a_n,i_n)} \mathbb{C}_{n+1}$*, two transitions* $\xrightarrow{(a_j,i_j)}$ *and* $\xrightarrow{(a_k,i_k)}$ *are said to form a* race *if* $i_j \neq i_k$ *and* $a_j, a_k \in \{\mathtt{rd}_x, \mathtt{wr}_x\}$ *and at least one is* $\mathtt{wr}_x$*.*

$(S, P_i[\boldsymbol{E}[x]]) \xrightarrow{(\mathtt{rd}_x^v, i)} (S, P_i[\boldsymbol{E}[v]])$ *where* $S(x) = v$

$(S, P_i[\boldsymbol{E}[x{:=}v]]) \xrightarrow{(\mathtt{wr}_x^v, i)} (S[x \leftarrow v], P_i[\boldsymbol{E}[()]])$

$(S, P_i[\boldsymbol{E}[(); C]]) \xrightarrow{(\tau, i)} (S, P_i[\boldsymbol{E}[C]])$

$(S, P_i[\boldsymbol{E}[\textbf{if true then } C_t \textbf{ else } C_f]]) \xrightarrow{(\tau, i)} (S, P_i[\boldsymbol{E}[C_t]])$

$(S, P_i[\boldsymbol{E}[\textbf{if false then } C_t \textbf{ else } C_f]]) \xrightarrow{(\tau, i)} (S, P_i[\boldsymbol{E}[C_f]])$

$(S, P_i[\boldsymbol{E}[\textbf{while } b \textbf{ do } C]]) \xrightarrow{(\tau, i)} (S, P_i[\boldsymbol{E}[\textbf{if } b \textbf{ then } \{C;\ \textbf{while } b \textbf{ do } C\} \textbf{ else } ()]])$

Fig. 1. Specification Semantics

$(M, P_i[\boldsymbol{E}[x]]) \xrightarrow{(\mathtt{rd}_x^v, i)} (M', P_i[\boldsymbol{E}[v]])$ *where* $M_i[x] = (M', v)$

$(M, P_i[\boldsymbol{E}[x{:=}v]]) \xrightarrow{(\mathtt{wr}_x^v, i)} (M', P_i[\boldsymbol{E}[()]])$ *where* $M_i[x \leftarrow v] = M'$

Fig. 2. Implementation Semantics

Specification semantics correspond to a programmer's intuitive view of interleaving execution (i.e. a sequentially consistent memory model). Here, processes execute one at a time (conceptually) albeit in a non-deterministic order, program order is respected and writes are atomic. Storage features such as caches and write buffers can violate these guarantees in a multiprocessor setting [1].

3 Implementation Semantics

In the implementation semantics, we replace the store S with a more general abstraction for memory, denoted as M. Each processor has a different view of the memory; processor i sees the value of x as $M_i[x]$ and in general $M_i[x]$ and $M_j[x]$ need not be equal. We will use M to model a memory hierarchy where each processor has a local cache. Our semantic account abstracts from the internal structure of M. We place purely *operational* constraints on M in order to prove sequential consistency theorems and later show that common cache based architectures satisfy these constraints. Thus while our focus is on the effects of caches, the framework presented here is more general.

In Figure 2 we present the significant changes to the operational rules. We omit the rules of Figure 1 that do not involve memory.

Both $\mathtt{wr}_x^v$ and $\mathtt{rd}_x^v$ transitions access the memory, and potentially alter it. Writing a variable (denoted $M_i[x \leftarrow v]$) returns the modified memory M'. Reading a variable from memory (denoted $M_i[x]$) returns a pair (M', v) where v is the value read, and M' is the possibly modified memory (e.g. an altered cache). For convenience, we write $M_i[x].val = v$ when $M_i[x] = (M', v)$. The next section imposes some restrictions on the permissible changes in M'.

In addition, we have some more transitions called the 'system' transitions, denoted by $\rightarrow$. These are used by the system to manage the internal structure of the memory. We will use $\rightarrow$ moves later e.g. to model cache consistency and cache replacement protocols. These transitions can fire non-deterministically at

any time. Moreover, the program does not constrain which system transitions can occur, or when. We abstractly characterize the system transitions in §3.2 by a series of properties which we use in the sequel. In particular the model described in §6 implements M as a cache based system satisfying these properties.

The transitions introduced in Figures 1 and 2 are now called 'program' transitions (since they fire as a direct result of some piece of code). We will use $\mathbb{C} \rightarrow_I^* \mathbb{C}'$ to denote that $\mathbb{C}'$ is reachable from $\mathbb{C}$ by the *implementation* semantics (program and system transitions), and similarly $\mathbb{C} \rightarrow_S^* \mathbb{C}''$ for the specification semantics. Note that we will use $\stackrel{*}{\rightarrow}$ to denote 0 or more *system* transitions, whereas $\rightarrow_I^*$ means 0 or more system and program transitions.

3.1 Coherence and Consistency

Let us call a configuration $\mathbb{C}$ "$\rightarrow$-normal" if it cannot make any $\rightarrow$ moves (i.e. system transitions). In order to relate implementation semantics to specification semantics, we need the following definition:

Definition 2. *An implementation configuration $\mathbb{C}_I$ is said to* reduce to *a specification configuration $\mathbb{C}_S$ (written $\mathbb{C}_I \Downarrow \mathbb{C}_S$) if $\exists \mathbb{C}'_I : \mathbb{C}_I \stackrel{*}{\rightarrow} \mathbb{C}'_I$, $\mathbb{C}'_I$ is $\rightarrow$-normal, and $\forall i \forall x\ \mathbb{C}'_I.M_i[x].val = \mathbb{C}_S.S[x]$. $\mathbb{C}_S$ is called a* reduct *of $\mathbb{C}_I$.*

In the next subsection we impose conditions that ensure the existence of reducts.

Definition 3. $\mathbb{C}$ *is* coherent for x *if* $\exists v : \forall \mathbb{C}_S : \mathbb{C} \Downarrow \mathbb{C}_S, \mathbb{C}_S.S(x) = v$.

A configuration is *coherent* if it is coherent for all x. It follows that a coherent configuration has a unique reduct. We use $\lceil\mathbb{C}\rceil$ to refer to the unique reduct of a coherent configuration $\mathbb{C}$.

Definition 4. $\mathbb{C}$ *is* consistent for x *if (a) it is coherent for x, (b) $\forall i, j, \mathbb{C}.M_i[x].val = \mathbb{C}.M_j[x].val$, (c) $\forall \mathbb{C}_S, \mathbb{C} \Downarrow \mathbb{C}_S \Rightarrow \forall i, \mathbb{C}.M_i[x].val = \mathbb{C}_S.S(x)$ and (d) $\forall i, w$ if $\mathbb{C}'$ is the same as $\mathbb{C}$ except that $\mathbb{C}'.M = \mathbb{C}.M_i[x \leftarrow w]$, then $\mathbb{C}'$ is coherent for x.*

Condition (a) ensures that a consistent configuration is coherent, and (d) that it remains coherent after any single write. Thus there are no pending writes to x in a configuration that is consistent for x. Condition (b) ensures that all views of x coincide and (c) that it agrees on x with its reducts. $\mathbb{C}$ is *consistent* if it is consistent for all x. A consistent configurations is in some sense identifiable with its reduct.

3.2 Constraints on the Memory Model

We now present the properties that the memory structure and its system transitions should satisfy. The theorems in the following sections hold for any system which has these properties.

Property 1. *If $\mathbb{C} \rightarrow \mathbb{C}'$, then $\mathbb{C}.P = \mathbb{C}'.P$*

System transitions have no effect on the program.

Property 2. *An $\rightarrow$-normal configuration is consistent.*

Property 2 ensures that system transitions are adequate to ensure consistency. For example, it prevents pathological cache architectures where the contents of only one designated cache are copied to the rest. In this pathological system, in the absence of a cached entry in the designated cache, inconsistent $\rightarrow$-normal configurations are possible.

Property 3. *Every configuration reachable from a consistent configuration has at least one $\rightarrow$-normal configuration under $\stackrel{*}{\rightarrow}$.*

We restrict this condition to configurations reachable from a consistent configuration because we *always* begin with a consistent configuration in practice. The above two properties together mean that such a configuration has at least one reduct. They also imply that any such configuration can always become consistent, which is important for the synchronization primitives of the sequel.

Property 4. *Consider $\mathbb{C}_I \stackrel{*}{\rightarrow} \xrightarrow{(a,i)} \mathbb{C}'_I$. For any $\mathbb{C}'_S$ such that $\mathbb{C}'_I \Downarrow \mathbb{C}'_S$, $\exists \mathbb{C}_S$: $\mathbb{C}_I \Downarrow \mathbb{C}_S$ which is identical to $\mathbb{C}'_S$ except in the position of the redex in P_i, unless $\mathbb{C}'_S.S(x) = v$ and $a = \boldsymbol{wr}^v_x$ in which case $\mathbb{C}_S$, $\mathbb{C}'_S$ may also differ on $S(x)$.*

Property 4 states that only writes may make a *fundamental* change in M, and only to a single variable. Reads are allowed to change M, but the change is superficial in this sense (and usually done solely for performance reasons).

Property 5. *If $\mathbb{C}$ is coherent (resp. consistent) for x and $\mathbb{C} \rightarrow \mathbb{C}'$, then $\mathbb{C}'$ is also coherent (resp. consistent) for x.*

Property 5 states that system transitions preserve coherence and consistency.

Property 6. *Let $\mathbb{C}$ be consistent for x and consider the sequence $\mathbb{C} \stackrel{*}{\rightarrow} \xrightarrow{(a_0,i_0)} \ldots \stackrel{*}{\rightarrow} \xrightarrow{(a_n,i_n)} \mathbb{C}'$. If there is at most* one *processor i such that $(a_k, i_k) = (\boldsymbol{wr}_x, i)$ in this sequence then $\mathbb{C}'$ is coherent for x.*

Property 6 says if there is no conflicting write then coherence is maintained.

Property 7. *If $\mathbb{C}$ is consistent, $\mathbb{C} \rightarrow^*_I \mathbb{C}'$ then for any i, if $\mathbb{C}'.M_i[x].val = v$ then either $\mathbb{C}.M_i[x].val = v$ or there is a $\boldsymbol{wr}^v_x$ on some processor in $\mathbb{C} \rightarrow^*_I \mathbb{C}'$.*

Property 7 states that a value is either set by some write or is preserved.

Property 8. *Let $\mathbb{C}$ be consistent for x, $\mathbb{C} \rightarrow^*_I \mathbb{C}_I \xrightarrow{(a,i)} \mathbb{C}'_I$, and $\mathbb{C}'_I \Downarrow \mathbb{C}'_S$. If $a \in \{\boldsymbol{wr}^v_x, \boldsymbol{rd}^v_x\}$ and $\mathbb{C}'_S.S(x) = w \neq v$ then there exists a transition $(\boldsymbol{wr}^w_x, j)$ with $j \neq i$ in $\mathbb{C} \rightarrow^*_I \mathbb{C}_I$.*

Property 8 means that the last write cannot be ignored and the last read cannot read the wrong value, *unless* they form a race with some earlier transition. System transitions must ensure that the effects of a write *can* propagate.

3.3 Derived Properties

The following lemmas can be derived from the above constraints.

Lemma 1. *If* $\mathbb{C}$ *is coherent for* x, $\mathbb{C} \xrightarrow{(a,i)} \mathbb{C}'$ *and* $a \neq \mathtt{wr}_x$ *then* $\mathbb{C}'$ *is coherent for* x.

Any transition not involving a write on x will maintain coherence for x.

Lemma 2. *Let* $\mathbb{C}$ *be reachable from a consistent configuration. If* $\mathbb{C}$ *is consistent for* x, $\mathbb{C} \rightarrow_I^* \mathbb{C}_I \xrightarrow{(a,i)} \mathbb{C}'_I$, $\mathbb{C}_I$ *is coherent and* $\mathbb{C}_I \Downarrow \mathbb{C}_S$, *then* $\exists \mathbb{C}'_s$ *such that following diagram commutes:*

$$\begin{array}{ccc} \mathbb{C}_I & \xrightarrow{(a,i)} & \mathbb{C}'_I \\ \Downarrow & & \Downarrow \\ \mathbb{C}_S & \xrightarrow{(a,i)} & \mathbb{C}'_S \end{array}$$

where either a *is not a memory access or if* a *accesses* x *then in* $\mathbb{C} \rightarrow_I^* \mathbb{C}_I$ *there is a* $\mathtt{wr}_x$ *transition only on* i.

By Properties 2, 3, 4 and 8. This lemma means that a write-free sequence of transitions exactly implements the specification semantics.

The following is a useful special case of this lemma:

Lemma 3. *If* $\mathbb{C}_I$ *is a consistent configuration then* $\exists \mathbb{C}'_S$ *which makes the following diagram commutes:*

$$\begin{array}{ccc} \mathbb{C}_I & \xrightarrow{(a,i)} & \mathbb{C}'_I \\ \Downarrow & & \Downarrow \\ \ulcorner \mathbb{C}_I \urcorner & \xrightarrow{(a,i)} & \mathbb{C}'_S \end{array}$$

We are now ready to consider two synchronization primitives in turn, and give sufficient conditions for sequential consistency for each.

4 Locks

We extend our language with a *locking* construct, **with** l **do** $\langle C \rangle$ following the approach of [4].

$\langle C \rangle ::= \ldots \mid$ **with** l **do** $\langle C \rangle$ $\qquad\qquad$ $\langle \boldsymbol{E} \rangle ::= \ldots \mid$ **holding** l **do** $\langle \boldsymbol{E} \rangle$

$\langle redex \rangle ::= \ldots \mid$ **with** l **do** $\langle C \rangle \mid$ **holding** l **do** ()

Additionally we have a construct **holding** l **do** $\langle C \rangle$ which is a *runtime* construct. It is used when a lock is held, and $\langle C \rangle$ is being executed. In the following subsection, we assume that the *initial* configuration we consider is written in the source language, and thus has no runtime constructs.

The configurations also change, becoming (S, L, P) (specification) and (M, L, P) (implementation) where L is the set of locks that are currently held. L remains

$(S, L, P_i[\boldsymbol{E}[\textbf{with } l \textbf{ do } C]]) \xrightarrow{(\hat{\mathtt{l}},i)} (S, L \cup \{l\}, P_i[\boldsymbol{E}[\textbf{holding } l \textbf{ do } C]]) \quad \textit{where } l \notin L$
$(S, L, P_i[\boldsymbol{E}[\textbf{holding } l \textbf{ do}()]]) \xrightarrow{(\check{\mathtt{l}},i)} (S, L - \{l\}, P_i[\boldsymbol{E}[()]])$

Fig. 3. Locks: Specification Semantics

$(M, L, P_i[\boldsymbol{E}[\textbf{with } l \textbf{ do } C]]) \xrightarrow{(\hat{\mathtt{l}},i)} (M, L \cup \{l\}, P_i[\boldsymbol{E}[\textbf{holding } l \textbf{ do } C]]) \quad \textit{where } l \notin L$
$(M, L, P_i[\boldsymbol{E}[\textbf{holding } l \textbf{ do } ()]]) \xrightarrow{(\check{\mathtt{l}},i)} (M, L - \{l\}, P_i[\boldsymbol{E}[()]]) \quad \textit{configuration is consistent}$

Fig. 4. Locks: Implementation Semantics

unaffected by all the transitions given so far, appearing unchanged on both sides. We introduce two new transitions for locking, with the decorations: $\hat{\mathtt{l}}$ (acquire lock l) and $\check{\mathtt{l}}$ (release lock l). The specification and implementation semantics for locks are given in Figures 3 and 4 respectively.

A lock l can only be acquired by a process if no other process holds l. Also, in the implementation semantics a lock can *only* be released when the configuration is consistent. Recall that Properties 2 and 3 ensure that this can happen.

4.1 Sequential Consistency

A sufficient condition to ensure sequential consistency even in the implementation semantics, is that the initial configuration must be Data Race Free (DRF):

Definition 5. *A consistent configuration* $\mathbb{C}$ *involves a* data race *if it has two redexes* $P_i[E[r]]$ *and* $P_j[E[r']]$, $i \neq j$, r *and* r' *are both accesses to the same variable and at least one is a write.* $\mathbb{C}$ *is* data race free *(DRF) iff no configurations specification reachable from* $\lceil\mathbb{C}\rceil$ *involve a data race.*

The following definition for *well synchronizedness* is often taken to be synonymous with DRF, and we treat it as such. Boudol and Petri's proof [4] of their equivalence applies to our model nearly unchanged since the proof is at the specification level, and our specification semantics are essentially the same as their 'strong' semantics. The full proof can be found in [6].

Definition 6. *A consistent configuration* $\mathbb{C}$ *is said to be* well-synchronized *(WS), iff in any valid sequence of specification transitions* $\lceil\mathbb{C}\rceil = \mathbb{C}_0 \xrightarrow{(a_0,i_0)} \ldots \xrightarrow{(a_{n-1},i_{n-1})} \mathbb{C}_n$ *if there exists* n_1 *and* n_2 *(with* $n_1 < n_2$*) such that* (a_{n_1}, i_{n_1}) *and* (a_{n_2}, i_{n_2}) *form a race, then* $\exists n_3 : n_1 < n_3 < n_2 \wedge i_{n_3} = i_{n_1} \wedge a_{n_3} = \check{\mathtt{l}}$.

For proofs, we will use this characterization rather than Definition 5. Informally, this property means that there must exist an unlocking operation between every pair of transitions forming a race (in every sequence of specification transitions). Note that we need only analyze sequentially consistent executions of a program in order to determine whether it is WS.

There is one last definition that is required in order to prove that the implementation and specification semantics coincide for WS programs.

Definition 7. *For any given consistent configuration* $\mathbb{C}$*, define* $R(\mathbb{C})$ *as:* $(\mathbb{C}_I, \mathbb{C}_S) \in R(\mathbb{C})$ *if and only if there exists a sequence of implementation transitions*

$$\mathbb{C} = \mathbb{C}_0 \xrightarrow{*} \xrightarrow{(a_0,i_0)} \mathbb{C}_1 \cdots \xrightarrow{*} \xrightarrow{(a_n,i_n)} \mathbb{C}_n = \mathbb{C}_I$$

such that

$$\overline{\mathbb{C}} = \mathbb{C}'_0 \xrightarrow{(a_0,i_0)} \mathbb{C}'_1 \cdots \xrightarrow{(a_n,i_n)} \mathbb{C}'_n = \mathbb{C}_S$$

is a valid sequence of specification transitions, with $\mathbb{C}_j \Downarrow \mathbb{C}'_j$ *for all* j*.*

We show that if $\mathbb{C}$ is WS then the relation $R(\mathbb{C})$ is a bisimulation (and thus the specification and implementation semantics are essentially the same).

In one direction, the simulation holds whether or not $\mathbb{C}$ is WS.

Theorem 1. *If* $(\mathbb{C}_I, \mathbb{C}_S) \in R(\mathbb{C})$ *and* $\mathbb{C}_S \xrightarrow{(a,i)} \overline{\mathbb{C}}_S$ *then there exists* $\overline{\mathbb{C}}_I$ *with* $\mathbb{C}_I \xrightarrow{*} \xrightarrow{(a,i)} \overline{\mathbb{C}}_I$ *such that* $(\overline{\mathbb{C}}_I, \overline{\mathbb{C}}_S) \in R(\mathbb{C})$*.*

The other direction also holds when the configuration is WS. Further, coherence is maintained.

Theorem 2. *If* $(\mathbb{C}_I, \mathbb{C}_S) \in R(\mathbb{C})$ *(with WS* $\mathbb{C}$*),* $\mathbb{C}_I$ *is coherent and* $\mathbb{C}_I \xrightarrow{*} \xrightarrow{(a,i)} \overline{\mathbb{C}}_I$ *then* $\overline{\mathbb{C}}_I$ *is coherent and there exists* $\overline{\mathbb{C}}_S$ *such that* $\mathbb{C}_S \xrightarrow{(a,i)} \overline{\mathbb{C}}_S$ *with* $(\overline{\mathbb{C}}_I, \overline{\mathbb{C}}_S) \in R(\mathbb{C})$*.*

5 Barriers

Barriers or fences are a common safety net in various processors with a relaxed memory model [1] and have also been used in other contexts [3]. Their role is to prevent instruction re-ordering across the barrier (hence the name). Unlike locks, they cannot entirely prevent data races but they can still guarantee sequential consistency (at least with the semantics that we present below).

We extend our language with a **bar** command, and a new expression for barred reads:

$$\langle e \rangle ::= \ldots \mid !\langle var \rangle \qquad \langle C \rangle ::= \ldots \mid \textbf{bar} \qquad \langle redex \rangle ::= \ldots \mid \textbf{bar} \mid !\langle var \rangle$$

The one-hole contexts remain unchanged. We introduce a barrier transition, decorated with `bar`. In the specification semantics, this is a no-op. In the implementation, it is a way of waiting for pending writes to complete. Figures 5 and 6 give the specification and implementation semantics respectively. The new read is only a way of introducing a barrier in the middle of an expression. The actual read is still handled by the usual $\mathtt{rd}_x^v$ actions.

Note that our barrier semantics enforces a *global* constraint. Hardware implementations today often provide barriers whose effects are in some way local to the current processor (e.g. x86 `mFence`) but as noted in [9], this is insufficient to ensure sequential consistency.

$(S, P_i[\boldsymbol{E}[\mathbf{bar}]]) \xrightarrow{(\mathtt{bar},i)} (S, P_i[\boldsymbol{E}[()]])$ *i.e. do nothing*
$(S, P_i[\boldsymbol{E}[!x]]) \xrightarrow{(\mathtt{bar},i)} (S, P_i[\boldsymbol{E}[x]])$ *i.e. do nothing*

Fig. 5. Barriers: Specification Semantics

$(M, P_i[\boldsymbol{E}[\mathbf{bar}]]) \xrightarrow{(\mathtt{bar},i)} (M, P_i[\boldsymbol{E}[()]])$ *configuration is consistent*
$(M, P_i[\boldsymbol{E}[!x]]) \xrightarrow{(\mathtt{bar},i)} (M, P_i[\boldsymbol{E}[x]])$ *configuration is consistent*

Fig. 6. Barriers: Implementation Semantics

5.1 Sequential Consistency

An important difference between this and the lock model is that in general, it is not possible to place `bar`s in the program in a way that ensures that there is a `bar` between every pair of actions forming a race in every sequentially consistent execution. Instead, we achieve sequential consistency by preventing multiple races involving the *same* processor from appearing between a pair of `bar`s.

Definition 8. *A sequence of transitions* $\mathbb{C} \rightarrow_I^* \mathbb{C}'$ *(resp.* $\mathbb{C} \rightarrow_S^* \mathbb{C}'$*) is called* multiple race free *iff for every* k *such that* (a_k, i_k) *forms a race with some transition in the sequence, if* $\exists j : j \neq k \wedge i_j = i_k$ *then* (a_j, i_j) *does not form a race with* any *transition in the sequence.*

Definition 9. *A consistent configuration* $\mathbb{C}$ *is said to satisfy the* barrier condition *iff in any valid sequence of specification transitions* $\lceil\mathbb{C}\rceil = \mathbb{C}_0 \xrightarrow{(a_0,i_0)} \mathbb{C}_1 \ldots \mathbb{C}_n \xrightarrow{(a_n,i_n)} \mathbb{C}_{n+1}$*, if there are* `bar` *transitions at* $\{k_1, k_2, \ldots, k_m\}$ *then taking* $k_0 = 0$ *and* $k_{m+1} = n+1$*, each subsequence in* $\{\mathbb{C}_{k_i} \rightarrow_S^* \mathbb{C}_{k_{i+1}} | 0 \leq i < n+1\}$ *is multiple race free.*

As in the case for WS, we need only analyze sequentially consistent executions of a program to verify that it satisfies the barrier condition.

To prove the equivalence of the two semantics under this condition, we show that for every specification-reachable configuration there is an implementation-reachable configuration, and vice-versa. One direction is quite trivial, and the barrier condition is not required:

Theorem 3. *For any consistent configuration* $\mathbb{C}$*, for every specification configuration* $\lceil\mathbb{C}\rceil \rightarrow_S^* \mathbb{C}_S$*, there exists an implementation configuration* $\mathbb{C}_I$ *such that* $\mathbb{C} \rightarrow_I^* \mathbb{C}_I$ *and* $\mathbb{C}_I \Downarrow \mathbb{C}_S$*.*

Theorem 4. *For any consistent configuration* $\mathbb{C}$ *which satisfies the barrier condition, for every implementation configuration* $\mathbb{C}_I$ *such that* $\mathbb{C} \rightarrow_I^* \mathbb{C}_I$ *and every specification configuration* $\mathbb{C}_S$ *such that* $\mathbb{C}_I \Downarrow \mathbb{C}_S$*,* $\lceil\mathbb{C}\rceil \rightarrow_S^* \mathbb{C}_S$*.*

In order to prove this theorem, we need the following lemmas:

$$(S, C, P_i[\boldsymbol{E}[x]]) \xrightarrow{(\mathtt{rd1}^v_x, i)} (S, C, P_i[\boldsymbol{E}[v]]) \qquad \text{where } x \in dom(C_i) \wedge C_i[x].val = v$$

$$(S, C, P_i[\boldsymbol{E}[x]]) \xrightarrow{(\mathtt{rd2}^v_x, i)} (S, C, P_i[\boldsymbol{E}[v]]) \qquad \text{where } x \notin dom(C_i) \wedge S(x) = v$$

$$(S, C, P_i[\boldsymbol{E}[x]]) \xrightarrow{(\mathtt{rd3}^v_x, i)} (S, C_i[x \leftarrow (v, \mathtt{clean})], P_i[\boldsymbol{E}[v]]) \quad \text{where } x \notin dom(C_i) \wedge S(x) = v$$

$$(S, C, P_i[\boldsymbol{E}[x{:=}v]]) \xrightarrow{(\mathtt{wr}^v_x, i)} (S, C_i[x \leftarrow (v, \mathtt{dirty})], P_i[\boldsymbol{E}[()]])$$

Fig. 7. Implementation Semantics

Lemma 4. *Consider a consistent configuration $\mathbb{C}$, and a configuration $\mathbb{C}_I$ such that $\mathbb{C} \xrightarrow{*} \xrightarrow{(a_0,i_0)} \cdots \xrightarrow{*} \xrightarrow{(a_n,i_n)} \mathbb{C}_I$. If this sequence is multiple race free, then for every $\mathbb{C}_S$ such that $\mathbb{C}_I \Downarrow \mathbb{C}_S$ there exists a sequence $\lceil\mathbb{C}\rceil \xrightarrow{(b_0,j_0)} \cdots \xrightarrow{(b_n,j_n)} \mathbb{C}_S$. Furthermore, the sequence of pairs $\{(b_k, j_k)\}$ is a permutation of the sequence $\{(a_k, i_k)\}$.*

Lemma 5. *For any consistent configuration $\mathbb{C}$ which satisfies the barrier condition, for every implementation configuration $\mathbb{C}_I$ such that $\mathbb{C} \rightarrow_I^* \mathbb{C}_I$ and every specification configuration $\mathbb{C}_S$ such that $\mathbb{C}_I \Downarrow \mathbb{C}_S$, the following hold:*

1. *The sequence $\mathbb{C} \rightarrow_I^* \mathbb{C}_I$ is multiple race free between* `bars`.
2. *$\lceil\mathbb{C}\rceil \rightarrow_S^* \mathbb{C}_S$.*
3. *The sequence $\lceil\mathbb{C}\rceil \rightarrow_S^* \mathbb{C}_S$ is a barrier-bounded permutation of (the program transitions in) $\mathbb{C} \rightarrow_I^* \mathbb{C}_I$.*

6 Modeling Multiprocessors with Caches

This section gives examples of how a multiprocessor system with local caches can be modeled in our framework. The memory M now becomes a tuple (S, C), where S is a store (as in the specification semantics) and C is a set of $|P|$ caches. The caches contain a local copy of a subset of the store. When a variable is written to, the write is to the cache. System transitions are used to update the store and the other caches asynchronously at some later time. A read may also pull a variable into the cache.

If $x \in dom(C_n)$ then this means the n^{th} processor has x in its cache. Its value $C_n[x]$ is given by a pair $(val, state)$, where val is the ordinary integer value of the variable and $state$ may be either `clean` or `dirty`. A variable is `clean` either if it has not been written to by *this* processor, or if its changed value has been written through to the store. Otherwise it is `dirty`. However note that in general, $C_n(x) = (v, \mathtt{clean}) \not\Rightarrow S(x) = v$. The system may allow the store to contain a different value if some other processor has updated the store but this cache has not yet been notified. As a notational convenience, we shall write $C_i[x].val = v$ and $C_i[x].state = s$ when $C_i[x] = (v, s)$.

Figure 7 gives the semantics for $M_i[x]$ and $M_i[x \leftarrow v]$. We express all the possibilities as separate transitions to make it easier to read. There are three transitions for reading a variable but they merely represent the different cases possible for the same label $\mathtt{rd}^v_x$. Note also that there are two transitions for

$$(S,C,P) \xrightarrow[i\uparrow]{x} (S, C_i \uparrow x, P) \qquad C_i[x] = (v, \texttt{clean})$$

$$(S,C,P) \xrightarrow[i \to j]{x} (S, C_j[x \leftarrow (v, \texttt{clean})], P)$$
$$x \in dom(C_j) \wedge C_j[x] \neq (v, \texttt{clean}) \wedge C_i[x] = (v, \texttt{dirty})$$

$$(S,C,P) \xrightarrow[i \to S]{x} (S[x \leftarrow v], C_i[x \leftarrow (v, \texttt{clean})], P)$$
$$\forall j : j \neq i \wedge x \in dom(C_j), C_j[x] = (v, \texttt{clean}) \wedge C_i[x] = (v, \texttt{dirty})$$

Fig. 8. System Transitions: Update

$\texttt{rd}_x^v$ when $x \notin dom(C_i)$, corresponding to whether or not x is pulled into the cache. This decision is made non-deterministically, which (along with another transition for *eviction* to be introduced later) makes the model independent of the *cache-replacement* policy used by the actual implementation.

The system transitions are used to propagate writes to other caches and the store. In practice this is usually done either with an *update-based* protocol (where cached copies are updated with the new value) or with an *invalidation-based* protocol (where cached copies are invalidated, effectively removing them from the cache)[5]. We give two sets of system transitions, one for each type of protocol.

This model allows for $\mathbf{W} \rightarrow \mathbf{R}$ reordering, since a read may be serviced while a previous write (to the cache) has not yet been propagated. $\mathbf{W} \rightarrow \mathbf{W}$ reorderings are possible because there is no guarantee that writes will be propagated in the order in which they appear. Thus the behaviors described in Fig 5 (a) and (b) in [1] will be exhibited by this model. Further, a processor may see its own writes before any other processor, simply because updates/invalidates haven't occurred yet. Similarly, other processors may see the write at different times, since the updates/invalidates on other caches need not happen all at once.

In [6] we show that this model with an update-based protocol satisfies the constraints in §3.2. This means that the abstract operational description presented earlier is general enough to at least allow these four relaxations.

6.1 Coherence and Consistency

For both protocols, we reformulate the definitions of coherence and consistency. We will show that these *structural* definitions are equivalent to the *operational* definitions given earlier.

Definition 10. *A configuration* $\mathbb{C}$ *is said to be* coherent for x *if* $\exists v : \forall i : x \in dom(C_i) \wedge C_i[x].state = \texttt{dirty} \Rightarrow C_i[x].val = v$.

Definition 11. *A configuration* (S,C,P) *is said to be* consistent for x *if and only if* $\forall i : x \in dom(C_i), C_i[x] = (S(x), \texttt{clean})$.

6.2 Modeling Update-Based Protocols

Figure 8 gives the system transitions used in the update-based model. The transitions are as follows:

1. **Eviction $\xrightarrow[i\uparrow]{x}$:** Evict x from C_i. This is only used for the cache replacement policy. We do not need it to achieve a consistent configuration in this model.
2. **Cache update $\xrightarrow[i\to j]{x}$:** Update x in C_j from C_i. This is used to update other caches when a variable is written to in a cache. It can be applied anytime there is some cache whose entry for the variable differs from the "correct" one.
3. **Store update $\xrightarrow[i\to S]{x}$:** Update x in S from C_i. The condition for its application ensures that a store update only happens *after* all caches have been updated and agree on the value of the variable.

As an example of how these transitions work, suppose a write occurs on a processor, and this is the only processor where that variable is `dirty`. An update-based system would execute the cache update transition multiple times to update the value in the other caches, and then a store update to put that value in the store. The $\to$-normal configurations are those that have empty caches.

If $\mathbb{C}$ is coherent for x by the structural definition, then we can see that any $\xrightarrow[i\to S]{x}$ will set $S(x)$ to the same value. Conversely, if $\exists i, j : C_i[x].state = C_j[x].state = \texttt{dirty} \wedge C_i[x].val \neq C_j[x].val$ then depending on whether $\xrightarrow[i\to j]{x}$ or $\xrightarrow[j\to i]{x}$ occurs (one of the two must occur), two distinct store updates are possible. Thus the structural and operational definitions for coherence are equivalent.

Similarly, it is easy to see that if $\mathbb{C}$ is consistent by the structural definition, it is consistent by the operational definition. Conversely, for $\mathbb{C}$ to be consistent for x, $\forall i : x \in dom(C_i), C_i[x].val = S(x)$ due to (b) and (c) of Definition 4. Further, if $\exists i : C_i[x] = (v, \texttt{dirty})$ then for any $w \neq v \wedge j \neq i$, $\mathbb{C}'$ is not coherent for x where $\mathbb{C}'.M = \mathbb{C}.M_j[x \leftarrow w]$, thus violating condition (d) of Definition 4.

This model satisfies all the constraints in §3.2 [6]. We additionally prove that it is possible to achieve a consistent configuration *without* the use of the eviction transition. This models the intended semantics of an update based protocol.

Lemma 6. *For any configuration $\mathbb{C}$ reachable from a consistent configuration, there exists a sequence of system transitions $\xrightarrow{*}$ not involving evictions such that $\mathbb{C} \xrightarrow{*} \mathbb{C}'$ and $\mathbb{C}'$ is a consistent configuration.*

6.3 Relaxing the Unlocking condition

The following lemma holds in the update based model:

Lemma 7. *If $\mathbb{C}$ is consistent for x and $\mathbb{C} \to_I^* \mathbb{C}_I$ where $\mathbb{C}_I$ is such that $\forall i$, $C_i[x].state = \texttt{clean}$, then $\mathbb{C}_I$ is also consistent for x (i.e. $\forall i, \mathbb{C}_I.C_i[x].val = S(x)$)*

Using this lemma, we can relax the condition for an `ĭ` operation and still ensure that the implementation and specification semantics coincide for WS configurations. Currently, an unlock can only happen if the configuration is consistent, but we can replace the "*configuration is consistent*" condition with the following:

$$\forall x \in dom(C_i), C_i[x].state = \texttt{clean}$$

$$(S, C, P) \xrightarrow[i\uparrow]{x} (S, C_i \uparrow x, P) \qquad C_i[x] = (v, \texttt{clean})$$
$$(S, C, P) \xrightarrow[i \to S]{x} (S[x \leftarrow v], C_i[x \leftarrow (v, \texttt{clean})], P) \qquad C_i[x] = (v, \texttt{dirty})$$

Fig. 9. System Transitions: Invalidation

i.e. there are no `dirty` entries in the current cache. This is a purely *local* condition (i.e. local to the current processor) on the unlock similar to the semantics of the unlock instruction on x86 processors [8].

In a system where Lemma 7 holds, this condition ensures that the configuration is consistent for x after the last unlock $\check{\texttt{l}}$ in a WS sequence, as used in the proof of Theorem 2. We can modify the abstract semantics to allow local semantics for unlocking $\check{\texttt{l}}$ by introducing an abstract predicate on M, $\texttt{safe}_i^x$ with the conditions that only a $(\texttt{wr}_x, i)$ destroys $\texttt{safe}_i^x$, and $\forall i\ \texttt{safe}_i^x \Rightarrow$ *consistent for x.* An $\check{\texttt{l}}$ can then happen only when $\forall x\ \texttt{safe}_i^x$. Then in the cache model $\texttt{safe}_i^x$ is definable as $C_i[x].state = \texttt{clean}$.

Note that with the *global* semantics for locks it may be possible to simulate barriers with locks, but with these local semantics that is no longer true.

6.4 Modeling Invalidation-Based Protocols

Figure 9 gives the system transitions required to model an invalidation based cache consistency protocol. The transitions are as follows:

1. **Eviction $\xrightarrow[i\uparrow]{x}$:** This is now used for both cache replacement and cache consistency.
2. **Store update $\xrightarrow[i \to S]{x}$:** This can now happen *before* other caches have been notified about a write.

As an example of how these work, suppose a write occurs on a processor, and this is the only processor where that variable is `dirty`. An invalidation-based system would execute $\xrightarrow[j\uparrow]{x}$ on all other caches, and $\xrightarrow[i \to S]{x}$ on this cache (with no restrictions on the order in which these are carried out).

The proof that this model satisfies the constraints in §3.2 is nearly the same as that for the update model. Lemma 7 does *not* hold in this model, so the relaxed unlock condition cannot be used. The reason this lemma does not hold is that the last store update can happen before all caches have been invalidated. Thus some cache may hold a `clean` entry which nevertheless has a wrong value, simply because it has not yet been evicted.

7 Related and Future Work

Our work is complementary to the seminal work of Adve *et al* [1] wherein they present a classification of memory models (from a systems perspective rather

than a programming perspective) that examine permissible reorderings of instructions. The notions of Data Race Freedom and weak orderings are also extensively explored by Adve *et al* [2].

Owens *et al* [8] have considered the x86 memory model and shown the correspondence between an axiomatic characterization of the model and an operational one. It would be interesting to relate our work to such concrete instances.

There are several strands of work that we identify for the future. First, the conditions we give are certainly *sufficient* to ensure sequential consistency, but it is not clear whether they are *necessary*. We also plan to investigate other synchronization primitives in a similar manner, in particular atomic *compare-and-swap* instructions for synchronization which are preferred over locks in many processors.

An interesting direction is the development of program analysis tools that will help analyze whether a program satisfies the condition that guarantees sequentially consistent behavior. Finally, we are formalizing the results presented here using the proof assistant Coq.

References

1. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. Computer 29(12), 66–76 (1996)
2. Adve, S.V., Hill, M.D.: Weak Ordering—A new definition. SIGARCH Comput. Archit. News 18(3a), 2–14 (1990)
3. Arvind, N.N., Maessen, J.-W., Nikhil, R.S., Stoy, J.E.: A Lambda Calculus with Letrecs and Barriers. In: Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science, London, UK, pp. 19–36. Springer, Heidelberg (1996)
4. 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, pp. 392–403. ACM, New York (2009)
5. Handy, J.: The Cache Memory Book. Academic Press Professional, Inc., San Diego (1993)
6. Joshi, S., Prasad, S.: An Operational Model for Multiprocessors with Caches. Technical report, Indian Institute of Technology Delhi (2010), `http://cse.iitd.ac.in/~sanjiva/OpCache.pdf`
7. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers 100(28), 690–691 (1979)
8. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOL 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009)
9. 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. ACM SIGPLAN Notices 44(1), 379–391 (2009)
10. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Information and Computation 115, 38–94 (1992)