A Characterization of Scalable Shared Memories[*]

Prince Kohli
Gil Neiger
Mustaque Ahamad

GIT–CC–93/04

January 19, 1993

Abstract

The traditional consistency requirements of shared memory are expensive to provide both in large scale multiprocessor systems and also in distributed systems that implement a shared memory abstraction in software. As a result, several memory systems have been proposed that enhance performance and scalability of shared memories by providing weaker consistency guarantees. Often, different models are used to describe such memories which makes it difficult to relate and compare them. We develop a simple non-operational model and identify parameters that can be varied to describe not only the existing memories but also to identify new ones. We show how such a uniform framework makes it easy to compare and relate the various memories. We also use the model to show that a well known software solution to the critical section problem can be used to distinguish the RC_{sc} and RC_{pc} memories explored in the DASH architecture.

Keywords: Scalable shared memories, memory consistency, synchronization, formal models.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

[*]This work was supported in part by the National Science Foundation under grant CCR-9106627.

1 Introduction

The programming and performance of parallel and distributed applications depends on the mechanisms used for sharing state across processors in such systems. Shared memory is attractive because it simplifies programming since processors can access both local and remote state using the standard read and write operations. However, the strong consistency guarantees provided by traditional memories can have a significant impact on the performance of applications. Strong consistency also limits the scalability of shared memory systems. These problems have been recognized both in the study of multiprocessors, where the hardware provides support for shared memory, and in distributed systems, where the shared memory abstraction is implemented in software. As a result, a number of new memory models have been proposed [1,3,5,6,14,15,17] that seek to enhance performance and scalability by weakening the consistency guarantees provided by shared memory. In the first approach, taken in the systems described in [1,5,6], strong consistency is only provided for a subset of the operations (e.g. synchronization operations) and other operations can be executed more efficiently due to their weakened consistency. In these systems, programs that meet certain requirements (properly labeled or data-race-free) do not need to be aware of the weak consistency and can be programmed as if the system provides strong consistency. The second approach is taken in distributed systems [3,15] where the application programmer must directly program with the weakly consistent memory. It is argued that, for many applications, programming is simpler than when message passing is used and the weaker memory consistency leads to improved performance.

Many systems advocate high performance shared memories that provide weaker consistency guarantees and several such memory models have been proposed. These memories are defined using different models. For example, *Processor Consistency* (PC) as defined by Gharachorloo et al. provides an operational definition by stating how read and write operations are executed. On the other hand, the *Total Store Ordering* (TSO) memory model that is implemented by the SPARC architecture is defined using an axiomatic approach. The different models used for defining memories make it hard to relate and compare these memories. In this paper, we present a simple non-operational model and identify key parameters that can be varied to systematically define and relate the memories that have been proposed in the literature. The model can also be used to identify new memories. Memories in our model are characterized by the *execution histories* that are allowed by a given memory model. Informally, weaker consistency places fewer demands on the execution histories and, as a result, permits a larger set of histories. Thus, we are able to use a set-based approach to relate and compare the various memories.

We also use our model to demonstrate that Lamport's Bakery algorithm for solving the n-processor mutual exclusion problem executes correctly with the release consistency memory model of the DASH system when labeled operations are sequentially consistent (RC_{sc}) but does not when the labeled operations are only processor consistent (RC_{pc}). This demonstrates that the RC_{sc} and RC_{pc} models differ for applications that may use read and write operations to achieve mutual exclusion in a shared memory system.

The paper is organized as follows. Section 2 presents the model and identifies the parameters that can be varied to obtain the different memories. Several of the memories are defined using this model in Section 3. Section 4 relates and compares some of these memories. The Bakery algorithm and its executions on RC_{sc} and RC_{pc} memories are discussed in Section 5. We compare our results with other related work in Section 6. In particular, we compare our model to the axiomatic model used in defining TSO [17]. Finally we conclude the paper in Section 7.

2 The Model

This section describes the model that underlies our definitions and results. Our model is motivated by the ones used by Misra [16] and Herlihy and Wing [10]. We define the system to be a finite set of *processors* that interact via a shared memory consisting of a finite set of *locations*. Processors execute read and *write* operations. Each such operation acts on a named location and has an associated value. For example, a write operation executed by processor p, denoted by $w_p(x)v$, stores the value v in location x; a similarly denoted read operation, $r_p(x)v$, reports that v is stored in location x. The execution of a processor is defined by a *processor execution history*, which is a sequence of read and write operations. The execution history of processor p, denoted by H_p, is the sequence $o_{p,1}, o_{p,2}, \ldots, o_{p,i}, \ldots$, where $o_{p,i}$ is the ith operation issued by processor p. The set of processor execution histories is the *system execution history.* Thus, a system execution history $H = \{H_p \mid p \in \mathcal{P}\}$ where $\mathcal{P}$ is the set of processors in the system.

We characterize various memories by the set of system execution histories that can be produced when processors execute with a certain type of memory. In particular, we need to develop rules to determine if a certain system execution history H is possible with a given type of memory. Our general approach consists of showing that each processor can assume that the memory has performed some set of operations in a *sequential order.* This set of operations must include the processor's operations and can also include operations of other processors to shared locations. Since a processor view is sequential, there is a unique "most recent" write preceding each read operation and it is required that each read operation return the value written by that write.[1] As far as a processor is concerned, it can assume that the shared memory executed only the operations included in its view, one at a time, and in the order defined by the view. This exactly defines the state of the memory when an operation is executed by the processor.

More precisely, for each processor p, there exists an *ordered* or *sequential* execution history $S_{p+\delta_p}$ that includes all operations in H_p as well as a subset δ_p of operations from the execution histories of other processors. Furthermore, the value v returned by a read operation $r(x)v$ is written by a write operation $w(x)v$ that precedes $r(x)v$ in $S_{p+\delta_p}$ such that there are no other writes to x between these operations in $S_{p+\delta_p}$. We say that a sequential history that has this property is *legal.* We will use $S_{p+\delta_p}$ to denote the "view" of processor p. Since we allow each processor to define its own view of shared memory, our model permits us to specify weakly consistent memories that

[1]We assume that all locations have initial value 0.

permit processors to observe values written by different processors in different order; this can happen if different processors have different views.

The consistency guarantees of a memory model place restrictions on processor views or the sequential histories that can be created for processors. The following parameters characterize these restrictions:

1. *Set of Operations*: A history $S_{p+\delta_p}$ not only includes the operations executed by p but also operations of other processors. Thus, the membership of δ_p needs to be specified for a given memory model. Two natural choices for the set δ_p are, first, all operations of other processors, and second, all write operations of other processors. In the first case, each processor's view of memory must include all operations executed by the memory system. In this case, we use a to denote the value of δ_p for any processor p in the system and hence refer to $S_{p+\delta_p}$ simply as S_{p+a}. In the second case, a processor needs to include only the write operations of other processors; this is plausible since only these operations change the state of the shared memory. As we will see later, this allows processors to develop independent views of memory and is used in many of the weaker consistency memory models. When δ_p consists of write operations of processors other than p, we will denote it by w and hence we will use S_{p+w} instead of $S_{p+\delta_p}$.

   There exist models that further distinguish memory operations. Examples of these include labeled operations in *release consistency* [6] and strong and weak operations in *hybrid consistency* [4]. In such models, δ_p might include only a certain type of read or write operations of other processors.

2. *Mutual Consistency*: Although processors can define their own views of memory, there may need to be mutual consistency requirements as the views result from accessing a shared memory. For example, a memory model may require that all writes to a given location appear in the same order in the sequential histories for all processors even when this ordering is not required by (3) below. This particular form of consistency is equivalent to *coherence*, which is provided in several memory models [2,6]. Another example of mutual consistency could be the requirement that all strong (or labeled) operations appear in the same order in all processor views.

3. *Ordering*: Most memory models are such that the order of the operations in the processor views must reflect somehow the actual ordering of these operations in system execution history H. *Program order* is one such commonly used order which states that $o_{p,i}$ is ordered before $o_{p,j}$ when $i < j$ ($o_{p,i}$ and $o_{p,j}$ are operations of processor p). A memory model may require that this or some other order derived from H be preserved between operations when they are included in the sequential execution history or view of a processor. The following orders are used in defining many of the memories.

   - Program order: For operations $o_{p,i}$ and $o_{p,j}$, we say $o_{p,i} \underset{po}{\rightarrow} o_{p,j}$ when $o_{p,i}$ precedes $o_{p,j}$ in the program, i.e., $i < j$. In this case, we say $o_{p,i}$ *is ordered* *before* $o_{p,j}$ *by the program order.* This defines program order to be total on any processor execution history; it orders all operations of a given processor.

Some memory definitions consider *non-blocking operations* [7]; after invoking a non-blocking operation. When considering such operations, an operation $o_{p,i+1}$ that follows $o_{p,i}$ may *bypass* it. In other words, $o_{p,i+1}$ may complete before $o_{p,i}$ in some processor view. In this case, all orderings defined by $\underset{po}{\rightarrow}$ may not be maintained. Thus, in such systems operations of a processor are only *partially ordered.* We use $\underset{ppo}{\rightarrow}$ to represent this weaker program order and write $o_1 \underset{ppo}{\rightarrow} o_2$ if o_1 and o_2 are operations of the same processor, $o_1 \underset{po}{\rightarrow} o_2$, and one of the following holds:

- o_1 and o_2 are operations on the same location;
- o_1 and o_2 are both reads or both writes;
- o_1 is a read and o_2 is a write; or
- there is another operation o' at the same processors such that $o_1 \underset{ppo}{\rightarrow} o' \underset{ppo}{\rightarrow} o_2$.

Thus $\underset{ppo}{\rightarrow}$ only partially orders a processor's operations because the case where o_1 is a write and o_2 is a read is omitted above.

- Writes-before order: If o_1 is a write to some location and o_2 is a read of the same location by a processor[2] (which may be different from the writer), then $o_1 \underset{wb}{\rightarrow} o_2$ if o_2 reads the value written by o_1. We call this the *writes-before* order and it captures the natural requirement that, if a read operation returns the value written by a certain write operation, then the write operation must be ordered before the read.

- Causal order: The *happens-before* relation defined by Lamport [12] can also be adapted to a shared memory system and captures the causal relationship between the read and write operations. Two operations are ordered by this relation if they are executed by the same processor or they are related by the writes-before relation (this is similar to the order established when a sent message is received) or the transitive closure thereof. For any two operations o_1 and o_2 in H, $o_1 \underset{co}{\rightarrow} o_2$ if

  - $o_1 \underset{po}{\rightarrow} o_2$ or
  - $o_1 \underset{wb}{\rightarrow} o_2$ or
  - for some operation o', $o_1 \underset{co}{\rightarrow} o'$ and $o' \underset{co}{\rightarrow} o_2$,

  that is, $\underset{co}{\rightarrow} = (\underset{po}{\rightarrow} \cup \underset{wb}{\rightarrow})^+$.

[2]We will use single subscripts when the extra information is irrelevant.

3 Memories Definitions

The model developed in the previous section can be used to define a variety of memories. Basically, a memory can be characterized by specifying the three parameters we identified: set of operations, mutual consistency, and ordering.

3.1 Sequential Consistency

We first consider *sequential consistency* (SC) [13], which is a widely accepted correctness condition for shared memory. In SC, the results of a system execution history H should be equivalent to some sequential execution of the operations of all processors in which the program order between operations is maintained. This can be captured in our model by requiring that for each processor p, the δ_p in $S_{p+\delta_p}$ consist of all operations of processors other than p. Thus, the view of processor p is defined by a sequential history S_{p+a} (recall that a refers to all operations of other processors). The mutual consistency requirement is that processors agree on their views or that, for all processors p and q, $S_{p+a} = S_{q+a}$. The ordering requirement for operations is the one defined by the program order relation "$\underset{po}{\rightarrow}$." In other words, if $o \underset{po}{\rightarrow} o'$ in H then o must precede o' in each of the processor views (S_{p+a}). In the case of SC, the mutual consistency requirement is redundant. If a sequential history alone meets the other two requirements, then it includes all operations of all processors and preserves the relationships induced by program order. Thus, this single history can be the view of each processor and hence the mutual consistency requirement is trivially satisfied.

To see how memories with weaker consistency require all three kinds of requirements, we consider the *total store ordering* (TSO) [17] and the *processor consistency* (PC) [6] memory models that have been implemented in the DASH and SPARC architectures, respectively. We then consider other weak memories, including *pipelined RAM* (PRAM) [15] and *Causal Memory* [3].

3.2 TSO

In TSO, processors have local first-in-first-out (FIFO) buffers and a logically shared memory that is single-ported. A write operation[3] by a processor simply adds the newly written value to the buffer. A read operation must return either the most recently written value from the local buffer or a value must be fetched from shared memory when one does not exist in the buffer. The buffered writes are sent to the single memory in FIFO order, and a switch that controls access to the single-ported memory allows fair access to all processors.

TSO can be easily defined using our model. First, the operations in δ_p included in the view of processor p are simply the write operations of other processors. In other words, $S_{p+\delta_p}$ includes not only all operations of p but also all write operations of other processors. Thus, $\delta_p = w$. The mutual consistency condition requires that all write

[3]The SPARC architecture also includes *swap* instructions, which can be treated as similar to writes. For simplicity, we omit them in this discussion.

$$\begin{array}{lll} p\text{:} & w(x)1 & r(y)0 \\ q\text{:} & w(y)1 & r(x)0 \end{array}$$

Figure 1: TSO execution history

operations appear in the same order in all processor views. If we use $S_{p+w}|_w$ to denote the resulting sequence when all read operations are removed from S_{p+w}, then the mutual consistency requirement of TSO is such that, for all p and q, $S_{p+w}|_w = S_{q+w}|_w$. Thus, it is guaranteed that writes (or stores) are ordered the same way in all processor views. Unlike SC, the partial program order "$\underset{ppo}{\rightarrow}$" defines the ordering requirements for TSO. Thus, if $o \underset{ppo}{\rightarrow} o'$ and both of these operations appear in S_{p+w}, then o must precede o' in S_{p+w}.

The mutual consistency requirement enforces additional orderings that are not included in the "$\underset{ppo}{\rightarrow}$" order. For example, two writes by different processors are not ordered by $\underset{ppo}{\rightarrow}$ but must appear in the same order in all processor views because of the mutual consistency requirement.

To see that our definition does capture TSO, consider the simple example shown in Figure 1. Both p and q first write to locations x and y, respectively, and then read the location written by the other processor. This execution is not possible with SC because there exists no legal sequential execution that includes all four operations and maintains the program order dependencies between the operations. However, this execution is possible with TSO. The values produced by the write operations are buffered and the memory is not updated with these values when the read operations are executed. Since no values exist in the buffer for the locations being read, these reads return the initial values of locations x and y which are 0. Our model can also be used to show that the execution is possible with TSO. The processor views which include all operations of a given processor and writes of others can be constructed as follows:

$$\begin{array}{llll} S_{p+w}: & r_p(y)0 & w_p(x)1 & w_q(y)1 \\ S_{q+w}: & r_q(x)0 & w_p(x)1 & w_q(y)1 \end{array}$$

The order of write operations is the same in the two processor views and ordering specified by the partial program order "$\underset{ppo}{\rightarrow}$"is maintained. In particular, the order of q's operations is different from program order; this is allowed because the locations of the write and read operations are different and the read follows the write.

It is also easy to see that our characterization of TSO is equivalent to the axiomatic definition given in [17]. The first axiom requires that the memory execute write (store) operations in a total order. This is captured by the mutual consistency requirement on processor views in our model. The program orderings between write operations, as well as those between read and write operations, are precisely the orderings captured by "$\underset{ppo}{\rightarrow}$." The fact that reads return the value written by the most recent write (defined by the partial program order and the order imposed by memory on write operation) is trivially satisfied because processor views are legal. We do not need to explicitly state termination which is specified by an axiom in TSO. This is because an operation must appear in some sequential history and hence it must complete (unless the operation is the last one in a view). In a later section, we provide further comparison between our definition of TSO and the axiomatic specification given in [17].

3.3 Processor Consistency

We consider processor consistency as defined by Gharachorloo et al. [6]. They give an operational definition that describes the implementation of PC in the DASH architecture. This definition explicitly requires coherence; that is, for each memory location, there is a unique ordering of the writes to that location. Similar to TSO, if a processor executes a write followed by a read of a different location, PC also allows the read to "bypass" the write. In other words, the partial program order "$\underset{ppo}{\longrightarrow}$" defines the ordering requirements for the operations executed by a given processor. However, PC does not require that all writes appear in the same order in processor views. The orderings on writes of different processors arise from the following conditions for processor consistency that are given by Gharachorloo et al.:

1. before a read (LOAD) is allowed to perform with respect to any other processor, all previous read accesses must be performed, and

2. before a write (STORE) is allowed to perform with respect to any other processor, all previous accesses (reads and writes) must be performed.

For this definition, one operation is "previous" to another if both operations are by the same processor and the first precedes the other in program order. The notion of "perform" with respect to a processor is defined as follows. A read operation o_r is performed with respect to processor p when the value to be returned by o_r has been decided (a write by p to the location being read will not change the value returned by o_r). Similarly, a write operation o_w is performed with respect to p when a read of the same location by p returns the value written by o_w or a subsequent write operation (the notion of "subsequent" is well-defined because the memory is coherent).

These conditions lead to an ordering of writes that is quite different than that given by TSO. We define PC as follows. Similar to TSO, the view of a processor p includes not only the operations of p but also the write operations of all other processors. Thus, S_{p+w} denotes p's view. The mutual consistency condition is the following. If we denote by $S_{p+w}|_{w,x}$ the sequential history obtained from S_{p+w} after all read operations (to any location) and write operations to locations other than x have been deleted, then mutual consistency for PC requires that, for all processors p and q, $S_{p+w}|_{w,x} = S_{q+w}|_{w,x}$.

It is easy to see that this mutual consistency condition implies coherence. For any location x, construct a sequential history S_x that includes all read and write operations to x as follows. First, include all writes to x in the (unique) order ensured by the mutual consistency requirement of PC. Then, include each read of x (by any processor) following the write of the value it reads (there may be more than one read of the same value and the order of such reads by different processors is unimportant). It is easy to

$$\begin{array}{ll} p: & w(x)1 \\ q: & r(x)1 \;\; w(y)1 \\ r: & r(y)1 \;\; r(x)0 \end{array}$$

Figure 2: A PC execution history that is not TSO

see that this results in a sequential execution and that this can be done in a way that preserves the order of accesses to x. Thus, all operations to a given location are totally ordered.

For ordering operations within a processor's view, PC uses a "semi-causality" relation which is similar to the causal relation $\underset{co}{\rightarrow}$ defined earlier. The "$\underset{co}{\rightarrow}$" relation is a combination of program order $\underset{po}{\rightarrow}$ with the "writes-before" order $\underset{wb}{\rightarrow}$ that relates a write operation to any read of the value written. Because PC is coherent (given by the mutual consistency condition above), it makes sense to also consider a "reads-before" order that relates a read operation of an old value to a write of a new value. The semi-causality relation is defined by augmenting the weaker program order $(\underset{ppo}{\rightarrow})$ with weakened forms of the "writes-before" and "reads-before" orders.

The first of these weakened forms is called the *remote writes-before* order and is denoted by $\underset{rwb}{\rightarrow}$. We write $o_1 \underset{rwb}{\rightarrow} o_2$ if and only if $o_1 = w(x)v$, $o_2 = r(y)u$ and there is another operation $o' = w(y)u$ such that $o_1 \underset{ppo}{\rightarrow} o'$. That is, o_2 must be a read from a write that followed o_1 in program order. Note that a normal "writes-before" relation would have related o' to o_2; the remote relation relates an earlier write to o_2. The second relation is the *remote reads-before* order and is denoted by $\underset{rrb}{\rightarrow}$. We write $o_1 \underset{rrb}{\rightarrow} o_2$ if and only if $o_1 = r(x)v$, $o_2 = w(y)u$, and there is another operation $o' = w(x)v'$ such that o_1 precedes o' in coherence order (i.e., in S_x) and $o' \underset{ppo}{\rightarrow} o_2$. Again, there is a two-step chain from o_1 to o_2 but, in this case, the first link is from a read of an old value (from x) to the write of a new value (to x). The semi-causality relation, denoted $\underset{sem}{\rightarrow}$ is the transitive closure of the union of the partial program order relation $\underset{ppo}{\rightarrow}$, the remote writes-before relation $\underset{rwb}{\rightarrow}$, and the remote reads-before relation $\underset{rrb}{\rightarrow}$.

We can now specify the orderings that must be preserved between operations in processor views. If o_1 and o_2 are two operations in S_{p+w} such that $o_1 \underset{sem}{\rightarrow} o_2$, then o_1 must precede o_2 in S_{p+w}.

The correctness requirements of TSO are strictly stronger than PC (see next section). Thus, all executions that can be obtained with TSO are also executions with PC. However, there exist PC executions that are not allowed by TSO. Figure 2 shows an execution that is allowed by PC because the following processor views can be created:

$$\begin{array}{ll} S_{p+w}: & w_p(x)1 \;\; w_q(y)1 \\ S_{q+w}: & w_p(x)1 \;\; r_q(x)1 \;\; w_q(y)1 \\ S_{r+w}: & w_q(y)1 \;\; r_r(y)1 \;\; r_r(x)0 \;\; w_p(x)1 \end{array}$$

These views obviously satisfy the mutual consistency as well as the ordering requirements of PC. However, it is not possible to create processor views that satisfy TSO requirements. Since TSO requires writes to be ordered in the same way in all views, $w_p(x)1$ must precede $w_q(y)1$ according to q's view. However, r cannot then return 1 when it reads y and 0 (the initial value) for x in the read operation that follows $r_r(y)1$.

Before PC was implemented in the DASH architecture, it was first defined by Goodman [9]. It was later shown [2] that the two definitions were distinct and incomparable: neither is stronger than the other.

3.4 Release Consistency

In the DASH architecture, processor consistency is provided only for memory operations that implement "synchronization" between processors (other memories that provide selective synchronization were defined weak ordering [1] and hybrid consistency [4]). Such operations are called *labeled* and others are *ordinary*. *Release consistency* (RC) is designed to be used with programs that are *properly labeled*. Loosely speaking, these are programs in which all ordinary operations are "bracketed" between labeled operations that correspond to *acquire* (read) and *release* (write) operations on a synchronization variable. RC ensures that an ordinary operation completes before the following release operation is performed. Thus, if two ordinary write operations are executed to locations x and y, they could be propagated independently and their values may arrive in different order at different caches (coherence is required even for ordinary operations and hence writes to the same location must arrive in the same order). This provides added efficiency for ordinary operations. For a program to execute correctly, stronger consistency needs to be provided for synchronization operations. In [6], two consistency requirements are identified: RC_{sc} guarantees that labeled operations are sequentially consistent, and RC_{pc} guarantees that they are processor consistent.

Our goal is to characterize the behavior of shared memory when it provides RC. We do not consider here the properties of programs that ensure their correct execution on such a memory. We need to specify the three conditions identified earlier. In a system execution with a memory that provides release consistency, there are labeled operations as well as ordinary read and write operations. As in the case of most of the memories considered above, processor p's view must consist of all of its own operations and all write operations of other processors.[4] The mutual consistency requirement is that of coherence (see PC above).

The ordering requirements for RC are somewhat complex. For RC_{sc}, the labeled operations are SC, while for RC_{pc}, they are PC. This means, in the case of RC_{sc}, that, if $S_p|_\ell$ is the subsequence of S_p containing only labeled operations, then the sequences $S_p|_\ell$ meet the requirements of SC (notice that this implies an additional mutual consistency requirement on the labeled operations). A similar condition holds for RC_{pc}. All local

[4]We omit explicit consideration of read-modify-write operations, such as *test-and-set*. Within our formalism, these would be treated as write operations in the sense that they would be included in all processor views.

operations (including ordinary ones) obey the partial program order $\xrightarrow[ppo]{}$. That is, if o_1 and o_2 are two operations of p such that $o_1 \xrightarrow[ppo]{} o_2$, then o_1 precedes o_2 in S_p. Finally, the following conditions control the ordering of ordinary operations with respect to labeled operations (in all histories):

- Let o be an ordinary operation of p that follows a labeled read operation (acquire) operation o_r of p. Let o_w be the write operation (possibly by another processor) that is read by operation o_r. Then , o follows o_w in all histories in which they both appear.

- If o is an ordinary operation of p that precedes a labeled write operation (release) operation o_w of p, then o follows o_w in all histories in which they both appear.

(The first condition is slightly more complex because the acquire operation, which is a labeled read, does not appear in the views of all processors.) These two conditions ensure that ordinary operations are ordered, in all views, between the labeled operations that "bracket" them.

3.5 Other Memories

We can also use our model to define other memories that have been presented in the literature. In particular, we consider PRAM and causal memories.

PRAM [15] "pipelines" writes to memory, allowing the effects of writes (as perceived by other processors) to be arbitrarily delayed. An operational definition of PRAM is as follows: Assume each processor on a reliable network has a complete copy of a global shared memory. Reads and writes are performed on local memory: reads return the local value, writes update the local copy and broadcast the update. Processors receive and process these updates asynchronously and atomically (with respect to local operations). Broadcasts of update values are reliable and point-to-point ordered (all updates from a processor are received in order, but the relative order of updates from distinct processors may vary).

The copy of memory at each processor implements its view of the shared memory. Since the processor only executes its operations and received writes of others, δ_p consists of the write operations of other processors. PRAM has no mutual consistency requirement. Since writes from other processors arrive in FIFO channels, they are applied by a processor to its copy in program order. Thus, the ordering requirement can be specified by "$\xrightarrow[po]{}$." PRAM thus allows the execution shown in Figure 3, which is not allowed by TSO. Notice that, after each processor reads the value written by itself, p then reads the value written by q and q reads the value written by p. This is allowed by PRAM because each processor could receive the value written by the other after its first read operation and then update its copy with the value received in the message. The second read operation would then return the value written by the other processor. In fact, $S_{p+w} = w_p(x)1\ r_p(x)1\ w_q(x)2\ r_p(x)2$ and $S_{q+w} = w_q(x)2\ r_q(x)2\ w_p(x)1\ r_q(x)1$ are processor views that satisfy the PRAM requirements. The execution is not TSO

$$\begin{array}{lllll} p\text{:} & w(x)1 & r(x)1 & r(x)2 \\ q\text{:} & w(x)2 & r(x)2 & r(x)1 \end{array}$$

Figure 3: PRAM history that is not allowed by TSO

$$\begin{array}{lllll} p: & w(x)1 & w(y)1 & & \\ q: & r(y)1 & w(z)1 & r(x)2 & \\ r: & w(x)2 & r(x)1 & r(z)1 & r(y)1 \end{array}$$

Figure 4: Causal history that is not allowed by TSO

because no processor views exist in which each processor orders the writes the same way and their read operations return the values shown in the execution.

Causal memory [3] is similar to PRAM in the sense that processor views include local operations and write operations of other processors. Also similar to PRAM, causal memory has no mutual consistency requirement. However, the ordering requirement is stronger than that of PRAM. In particular, causal memory requires that the causal order "$\xrightarrow[co]{}$" be preserved between operations in processor views, whereas PRAM requires only the "$\xrightarrow[po]{}$" order be maintained. Because the causal order is stronger than program order, there exist execution histories that are allowed by PRAM but not allowed by causal memory. Figure 4 shows an execution that is allowed by causal but not by TSO. It is allowed by causal because the following processor views exist that maintain causal order between operations.

$$\begin{array}{llllllll} S_{p+w}: & w_p(x)1 & w_p(y)1 & w_q(z)1 & w_r(x)2 & & & \\ S_{q+w}: & w_p(x)1 & w_p(y)1 & r_q(y)1 & w_q(z)1 & w_r(x)2 & r_q(x)2 & \\ S_{r+w}: & w_r(x)2 & w_p(x)1 & r_r(x)1 & w_p(y)1 & w_q(z)1 & r_r(z)1 & r_r(y)1 \end{array}$$

The execution is not TSO because q's view implies that $w_r(x)2$ must have been performed after $w_p(x)1$ by the memory but, in that case, r's read of x cannot return 1. Notice that, in causal memory, once r returns the value 1 for z, it must also return the value 1 for y because it establishes a causal order between $w(y)1$ and r's read of y and there are no other writes to y. In PRAM, r need not return 1 when it executes the read operation for location y (it could return the initial value) because the message containing y's value might not have arrived at r.

4 Relating Memories

We have shown how several memories can be defined precisely using our model. The model also allows us to relate various memories. For example, if we can demonstrate

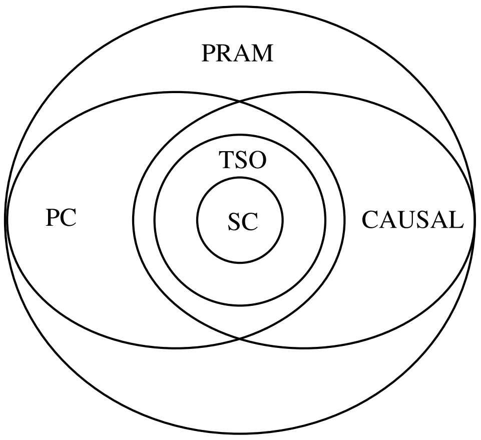

Figure 5: Relationship Between Memories

that TSO is strictly stronger than PC, it immediately follows that any application programmed to execute with PC can also execute correctly with TSO. A comparison can also provide intuition for the cost of implementing various memory models. A model that provides weaker consistency guarantees can usually be implemented less expensively than one with stronger consistency.

Our model provides a natural framework for relating and comparing memories. A memory model is characterized by the set of system execution histories that are allowed by the model (it must be possible to create processor views that include the specified set of operations and meet the ordering and mutual consistency requirements of the model for each system history included in the set). Thus, to show that one memory model A is strictly stronger than B, we need to show that each system execution history allowed by A is also a system execution history of B. Furthermore, there must exist histories allowed by B that are not histories of A. We can also use a Venn-diagram representation (see below) for the set of histories allowed by various memories. In this representation, if A is strictly stronger than B then A is contained in B.

Figure 5 shows how the memories we discussed in the previous section are related. SC is the strongest and hence the set representing it is contained in the sets of all other memories. PRAM is the weakest memory: we have already argued that it is weaker than causal memory; it is not hard to show [2] that it is also weaker than PC. TSO is weaker than SC but is strictly stronger than both PC and causal memory. Causal memory and PC are not comparable. This is because, while causal memory's ordering condition requires the stronger causal order, it lacks PC's mutual consistency requirement.

The relationships above are not hard to prove. For example, we prove that TSO is strictly stronger than PC. In the previous section, we demonstrated an execution history that is allowed by PC but not by TSO. Here, we show that every TSO system execution history is also allowed by PC. Let H be a TSO execution history. Since H is TSO, there must exist processor views S_{p+w} for each processor p such that

1. for all processors p and q, $S_{p+w}|_w = S_{q+w}|_w$, and

2. partial program order "$\xrightarrow[ppo]{}$" relationships between operations are preserved in the S_{p+w}'s.

PC also requires processor views to contain write operations of all other processors. We show that the S_{p+w} given by TSO can also be used to demonstrate that H is PC. Since $S_{p+w}|_w = S_{q+w}|_w$ for all p and q, it also follows that $S_{p+w}|_{w,x} = S_{q+w}|_{w,x}$ and hence the mutual consistency condition of PC is satisfied by the TSO processor views. It only remains to be seen that operations ordered by the semi-causality relation "$\xrightarrow[sem]{}$" appear in the same order in the S_{p+w}'s. Recall that the same partial program order is used by TSO and PC. Thus, the only order that PC requires but which might not be maintained by the S_{p+w}'s is the semi-causality order established when a processor p executes a read operation o_r that returns a value written by processor q and $p \neq q$. In this case, the write operation of q must precede o_r in S_{p+w} since the views are legal and any writes of p following o_r must come after this read operation. Thus, the write of q will precede the writes following o_r in S_{p+w} and hence in all processor views due to (1) above. We can extend this argument to cover cases where "$\xrightarrow[sem]{}$" orderings are introduced via several processors. Thus, TSO executions are also executions allowed by PC.

5 An Algorithm that Distinguishes RC_{sc} and RC_{pc}

It has been claimed that the RC_{sc} and RC_{pc} memory models should be equivalent for most practical applications. Gharachorloo et al. [6] state that "[f]or all applications that we have encountered, sequential consistency and processor consistency (for special [labeled] accesses) give the same results." In other words, programs that execute correctly on RC_{sc} memory should also execute correctly when they are run on a system that provides RC_{pc} memory. In this section, we show that a well-known algorithm that uses read and write operations to implement access to a critical section distinguishes RC_{sc} and RC_{pc}. The Bakery algorithm that was proposed by Lamport [11] executes correctly with RC_{sc} but fails when it is run on RC_{pc} memory.

Figure 6 shows the code for the Bakery algorithm, which controls access to a critical section that is shared by n processors numbered 1 to n (the code shown is for processor p_i. (We make the assumption that variables used for synchronization (*choosing* and *number*) are not accessed in the critical or remainder sections. We also assume that other shared variables are accessed only in the critical section.) The correct execution of the algorithm assumes that the data used by it is stored in sequentially consistent shared memory. When this is true, the algorithm ensures that at most one processor can be in the critical section at any given time and that the solution is free from deadlocks and starvation. To execute the algorithm on an RC_{sc} memory, we label all read and write operations of the code shown in Figure 6 except the ones in the critical and the remainder sections. Thus, all memory operations that are executed to gain access to the critical section or to release it are labeled. It is not hard to see that the program will now be *properly labeled* (see above). Gibbons, Merritt, and Gharachorloo [8] showed that any program proved correct with SC will remain correct if properly

```
shared $choosing[n]$ : boolean        /* Initially $false$ */
       $number[n]$ : integer          /* Initially 0 */
local  $j$ : integer

while $true$ do
    $w(choosing[i])true$
    $mine = 1 + \max\{number[j] \mid j \neq i\}$          /* reads the array $number$ */
    $w(number[i])mine$
    $w(choosing[i])false$
    for $j = 1$ to $n$ do
        if $j \neq i$ then
            repeat
                $test = r(choosing[j])$
            until not $test$
            repeat
                $other = r(number[j])$
            until $other = 0$ or $(mine, i) < (other, j)$ /* do lexicographic comparison */

    Critical section

    $w(number[i])0$

    Remainder section
```

Figure 6: Lamport's Bakery algorithm as executed by processor p_i

labeled and run with RC_{sc}. Thus, the Bakery algorithm can correctly control access to a critical section when run with RC_{sc}.

The Bakery algorithm fails to execute correctly when the memory model is changed to RC_{pc}. The labeled operations are now guaranteed only to be PC. Consider the same labeling as before and the case in which $n = 2$. It is not hard to construct an execution that admits the following local subhistories for p_1 and p_2:

$\cdots w_1(choosing[0])true$, $r_1(number[1])0$, $w_1(number[0])1$,
$w_1(choosing[0])false$, $r_1(choosing[1])false$, $r_1(number[1])0$, *Critical section,* $\cdots$; and

$\cdots w_2(choosing[1])true$, $r_2(number[0])0$, $w_2(number[1])1$,
$w_2(choosing[1])false$, $r_2(choosing[0])false$, $r_2(number[0])0$, *Critical section,* $\cdots$.

These two executions could occur because each processor can order the writes of the other after all of its own operations (that is, after it determines that it is safe to enter the critical section). In this case, both processors will enter the critical section simultaneously, a violation of correctness.

Intuitively, the failure of the algorithm is in part because it does not take advantage of the coherence provided by the memory. This is because no shared location is written by more than one processor. Thus, the Bakery algorithm shows that RC_{sc} and RC_{pc} differ in power for applications that implement processor coordination with read and write operations.

6 Comparison with Related Work

In this paper, we have focused on developing a uniform framework that can be used to define a variety of memories. We do not define new memories and thus our goal differs from many of the papers that introduce new memories. Other papers that have had similar goals are [16] and [17]. In [16], only atomic memory is considered, and this is stronger than sequential consistency. In [17], the TSO memory model is introduced using an axiom-based specification which can be used to capture several memories. The axiom-based model is also formal and precise and can be used to define and relate the memories as we have done in this paper.

We prefer the model used in this paper for several reasons. First, it is implementation independent but captures the essence of how memories can be implemented. For example, the per processor view can be thought of as the behavior of a local cache; the set of operations and the ordering requirements for the view specify how the cache should be accessed and updated. Also, processor views are a natural extension of how sequential consistency is defined and understood. SC requires all processors to agree on their views, whereas weaker memories allow them to differ in the set of operations they include and also in the ordering between the operations. Finally, we do not need to state termination explicitly; the fact that processor views are sequential histories and an operation needs to be included in some view implicitly requires termination of the operation. Also, we believe that the parameters of the model identified by us make it easy to relate and compare the various memories.

7 Concluding Remarks

It is difficult to relate and compare the numerous scalable or high performance memories that have been proposed because they are often defined using different models. We developed a framework that allows the characterization of many of the existing memories. It identifies the parameters that can be varied to get the various memories and provides us a simple technique to relate the memories. We used our model to characterize memories that include sequential consistency, processor consistency, release consistency, total store ordering, PRAM and causal. The model makes it easy to understand the relationships between the memories. We also used it to show that the Bakery algorithm, which can be used to implement n-processor mutual exclusion, fails to execute correctly when memory provides release consistency and the labeled (or strong) operations are processor consistent. This shows that the the RC_{sc} and RC_{pc} models are not identical when read and write operations are used to implement processor coordination.

We have focused on the characterization of and the relationships between different existing memories in this paper. However, the model also helps us in identifying new memories. For example, a mutual consistency condition that requires coherence can be added to causal memory or perhaps such coherence can only be required for labeled operations. The model can also help us to precisely characterize the program model for a given memory which can be used to identify the programs that will execute correctly on a certain weak memory when they are correct on sequentially consistent memory. We will address these issues in our future work.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering — a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

[2] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. Technical Report 92/34, College of Computing, Georgia Institute of Technology, 1992.

[3] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In S. Toueg, P. G. Spirakis, and L. Kirousis, editors, *Proceedings of the Fifth International Workshop on Distributed Algorithms*, volume 579 of *Lecture Notes on Computer Science*, pages 9–30. Springer-Verlag, October 1991.

[4] Hagit Attiya and Roy Friedman. A correctness condition for high performance multiprocessors. In *Proceedings of the Twenty-Fourth ACM Symposium on Theory of Computing*, pages 679–690. ACM Press, May 1992.

[5] Michel Dubois, Christoph Scheurich, and Faye Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–22, February 1988.

[6] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 15–26, May 1990.

[7] Phillip B. Gibbons and Michael Merritt. Specifying nonblocking shared memories (extended abstract). In *Proceedings of the Fourth Symposium on Parallel Algorithms and Architectures*, pages 306–315. ACM Press, June 1992.

[8] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories (extended abstract). In *Proceedings of the Third Symposium on Parallel Algorithms and Architectures*, pages 292–303. ACM Press, July 1991.

[9] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[10] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[11] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] Leslie Lamport. How to make a multiprocessor computer that correct executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[14] J. Lee and U. Ramachandran. Synchronization with multiprocessor caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 27–39, 1990.

[15] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.

[16] Jayadev Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[17] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Corporation, Palo Alto Research Center, December 1991.