Memory Consistency Models[1]

David Mosberger

TR 93/11

Abstract

This paper discusses memory consistency models and their influence on software in the context of parallel machines. In the first part we review previous work on memory consistency models. The second part discusses the issues that arise due to weakening memory consistency. We are especially interested in the influence that weakened consistency models have on language, compiler, and runtime system design. We conclude that tighter interaction between those parts and the memory system might improve performance considerably.

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This is an updated version of [Mos93]

1 Introduction

Traditionally, memory consistency models were of interest only to computer architects designing parallel machines. The goal was to present a model as close as possible to the model exhibited by sequential machines. The model of choice was sequential consistency (SC). Sequential consistency guarantees that the result of any execution of n processors is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program. However, this model severely restricts the set of possible optimizations. For example, in an architecture with a high-latency memory, it would be beneficial to pipeline write accesses and to use write buffers. None of these optimizations is possible with the strict SC model. Simulations have shown that weaker models allowing such optimizations could improve performance on the order of 10 to 40 percent over a strictly sequential model [GGH91, ZB92]. However, weakening the memory consistency model goes hand in hand with a change in the programming model. In general, the programming model becomes more restricted (and complicated) as the consistency model becomes weaker. That is, an architecture can employ a weaker memory model only if the software using it is prepared to deal with the new programming model. Consequently, memory consistency models are now of concern to operating system and language designers too.

We can also turn the coin around. A compiler normally considers memory accesses to be expensive and therefore tries to replace them by accesses to registers. In terms of a memory consistency model, this means that certain accesses suddenly are not observable any more. In effect, compilers implicitly generate weak memory consistency. This is possible because a compiler knows exactly (or estimates conservatively) the points where memory has to be consistent. For example, compilers typically write back register values before a function call, thus ensuring consistency. It is only natural to attempt to make this implicit weakening explicit in order to let the memory system take advantage too. In fact, it is anticipated that software could gain from a weak model to a much higher degree than hardware [GGH91] by enabling optimizations such as code scheduling or delaying updates that are not legal under SC.

In short, weaker memory consistency models can have a positive effect on the performance of parallel shared memory machines. The benefit increases as memory latency increases. In recent years, processor performance has increased significantly faster than memory system performance. In addition to that, memory latency increases as the number of processors in a system increases.

Shared memory can be implemented at the hardware or software level. In the latter case it is usually called Distributed Shared Memory (DSM). At both levels work has been done to reap the benefits of weaker models. We conjecture that in the near future most parallel machines will be based on consistency models significantly weaker than SC [LLG^{+}92, Sit92, BZ91, CBZ91, KCZ92].

The rest of this paper is organized as follows. In section 2 we discuss issues characteristic to memory consistency models. In the following section we present several consistency models and their implications on the programming model. We then take a look at implementation options in section 4. Finally, section 5 discusses the influence of weakened memory consistency models on software. In particular, we discuss the interactions between a weakened memory system and the software using it.

2 Consistency Model Issues

Choosing an appropriate memory consistency model (MCM) is a tradeoff between minimizing memory access order constraints and the complexity of the programming model as well as of the complexity of the memory model itself. The weakest possible memory is one returning for a read access some previously written value or a value that will be written in the future. Thus, the memory system could choose to return *any* of those values. While minimizing ordering constraints perfectly, it is not useful as a programming model. Implementing weaker models is also often more complex as it is necessary to keep track of outstanding accesses and restrict the order of execution of two accesses when necessary. It is therefore not surprising that many different MCM's have been proposed and new models are to be expected in the future. Unfortunately, there is no single hierarchy that could be used to classify the strictness of a consistency model. Below, we define the design space for consistency models. This allows us to classify the various models more easily.

Memory consistency models impose ordering restrictions on accesses depending on a number of attributes. In general, the more attributes a model distinguishes, the weaker the model is. Some attributes a model could distinguish are listed below:

- location of access
- direction of access (read, write, or both)
- value transmitted in access
- causality of access

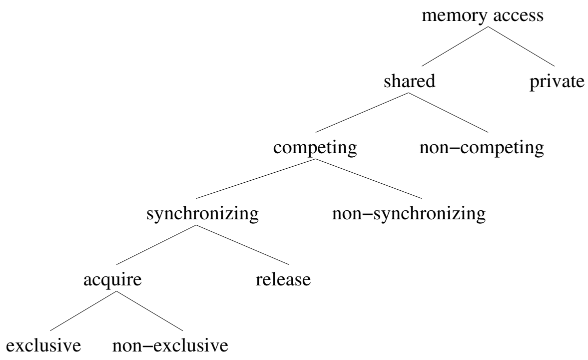

Figure 1: Access Categories

- category of access

The *causality* attribute is a relation that tells if two accesses a_1 and a_2 are (potentially) causally related [Lam78] and if so, whether a_1 occurred before a_2 or vice versa.

The access *category* is a static property of accesses. A useful (but by no means the only possible) categorization is shown in Figure 1. It is an extension of the categorization used in [GLL+90]. A memory access is either *shared* or *private*. Private accesses are easy to deal with, so we don't discuss them further. Shared accesses can be divided into *competing* and *non-competing* accesses. A pair of accesses is competing if they access the same location, at least one of them is a write access, and they are not ordered. For example, accesses to shared variables within a critical section are non-competing because mutual exclusion guarantees ordering[1]. A competing access can be either *synchronizing* or *non-synchronizing*. Synchronizing accesses are used to enforce order, for example by delaying an access until all previous accesses are performed. However, not all competing access are synchronizing accesses. Chaotic relaxation algorithms, for example, use competing accesses without imposing ordering constraints. Such algorithms work even if some read accesses do not return the most recent value.

Synchronizing accesses can be divided further into *acquire* or *release* accesses. An acquire is always associated with a read synchronizing access while a release is always a write synchronizing access. Atomic fetch-and-Φ operations can usually be treated as an acquiring read access followed by a non-synchronizing write access. Finally, an acquire operation can be either *exclusive* or *non-exclusive*. Multiple non-exclusive acquire accesses can be granted, but an exclusive acquire access is delayed until all previous acquisitions have been released.

[1] Assuming accesses by a single processor appear sequentially consistent

For example, if a critical region has only read accesses to shared variables, then acquiring the lock can be done non-exclusively.

Consistency models that distinguish access categories employ different ordering constraints depending on the access category. We therefore call such models *hybrid*. In contrast, models that do not distinguish access categories are called *uniform*. The motivation for hybrid models is engendered in Adve and Hill's definition for weak ordering [AH90]:

> *Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model.*

That is, as long as the synchronization model is respected, the memory system appears to be sequentially consistent. This allows the definition of almost arbitrarily weak consistency models while still presenting a reasonable programming model. All that changes is the number or severity of constraints imposed by the synchronization model.

3 Proposed Models

We now proceed to define some of the consistency models that have previously been proposed. We do not give formal definitions for the presented models as they do not help much to understand a model's implications on the programming model. More formal descriptions can be found for example in Ahamad et al. [ABJ+92] and Gharachorloo et al. [GLL+90]. We first discuss uniform models and then hybrid models. Figure 2 gives an overview of the relationships among the uniform models. An arrow from model A to B indicates that A is more strict than B. Each model is labeled with the subsection it is described in. Hybrid models are described roughly in decreasing order of strictness.

We use triplets of the form $a(l)v$ to denote a memory access, where a is either R for read access or W for write access, l denotes the accessed location, and v the transmitted value. In our examples, a triplet identifies an access uniquely. Memory locations are assumed to have an initial value of zero.

Execution histories are presented as diagrams with one line per processor. Time corresponds to the horizontal axis and increases to the right. For write accesses, a triplet in a diagram marks the time when it was issued, while for read accesses it denotes the time of completion. This asymmetry exists because a program can only "know" about these events. It does not know at which point in time a write performed or at which point in time a read was issued (there may be prefetching hardware,

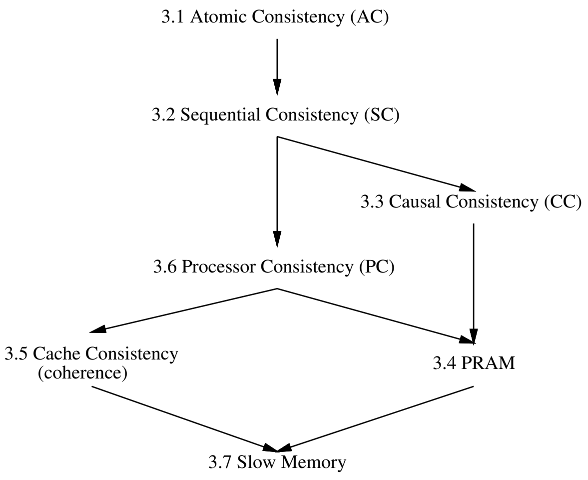

Figure 2: Structure of Uniform Models

for example). The following diagram is an example execution history:

$$\begin{array}{ll} P_1: & W(x)1 \\ \hline P_2: & \qquad\qquad R(x)1 \end{array}$$

Processor P_1 writes 1 to location x and processor P_2 subsequently observes this write by reading 1 from x. This implies that the write access completed (performed) some time between being issued by P_1 and being observed by P_2.

In the following discussion we use the word *processor* to refer to the entities performing memory accesses. In most cases it could be replaced by the word *process* as processes are simply a software abstraction of physical processors.

3.1 Atomic Consistency (AC)

This is the strictest of all consistency models. With atomic consistency, operations take effect at some point in an *operation interval*. It is easiest to think of operation intervals as dividing time into non-overlapping, consecutive slots. For example, the clock cycle of a memory bus could serve as an operation interval. Multiple accesses during the same operation interval are allowed, which causes a problem if reads and writes to the same location occur in the same operation interval. One solution is to define read operations to take effect at read-begin time and write operations to take effect at write-end time. This is called *static* atomic consistency[HA90]. With *dynamic* AC, operations can take effect at any point in the operation interval, as long as the resulting history is equivalent to some serial execution.

Atomic consistency is often used as a base model when evaluating the performance of an MCM.

3.2 Sequential Consistency (SC)

Sequential consistency was first defined by Lamport in 1979 [Lam79]. He defined a memory system to be sequentially consistent if

> *...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

This is equivalent to the *one-copy serializability* concept found in work on concurrency control for database systems [BHG87]. In a sequentially consistent system, all processors must agree on the order of observed effects. The following is a legal execution history for SC but not for AC:

$$\begin{array}{lllll} P_1: & W(x)1 & & & \\ \hline P_2: & & W(y)2 & & \\ \hline P_3: & & R(y)2 & R(x)0 & R(x)1 \end{array}$$

Note that $R(y)2$ by processor P_3 reads a value that has not been written yet! Of course, this is not possible in any real physical system. However, it shows a surprising flexibility of the SC model. Another reason why this is not a legal history for atomic consistency is that the write operations $W(x)1$ and $W(y)2$ appear commuted at processor P_3.

Sequential consistency has been the canonical memory consistency model for a long time. However, many multi-processor machines actually implement a slightly weaker model called processor consistency (see below).

3.3 Causal Consistency

Hutto and Ahamad [HA90] introduced causal consistency. Lamport [Lam78] defined the notion of *potential causality* to capture the flow of information in a distributed system. This notion can be applied to a memory system by interpreting a write as a message-send event and a read as a message-read event. A memory is causally consistent if all processors agree on the order of causally related events. Causally unrelated events (concurrent events) can be observed in different orders. For example, the following is a legal execution history under CC but not under SC:

P_1 :	$W(x)1$		$W(x)3$		
P_2 :	$R(x)1$	$W(x)2$			
P_3 :	$R(x)1$			$R(x)3$	$R(x)2$
P_4 :	$R(x)1$			$R(x)2$	$R(x)3$

Note that $W(x)1$ and $W(x)2$ are causally related as P_2 observed the first write by P_1. Furthermore, P_3 and P_4 observe the accesses $W(x)2$ and $W(x)3$ in different orders, which would not be legal under SC.

Among the uniform models, CC appears to be one of the more difficult to implement in hardware. This can probably be explained by the fact that most other models have been designed with a hardware implementation in mind. However, this does not imply that a CC implementation necessarily performs worse than an implementation of one of the simpler uniform models.

3.4 Pipelined RAM (PRAM)

Lipton and Sandberg [LS88] defined the Pipelined RAM (PRAM) consistency model. The reader should be aware that the acronym PRAM is often used as a shorthand for Parallel Random Access Machine which has nothing in common with the Pipelined RAM consistency model.

The reasoning that led to this model was as follows: consider a multi-processor where each processor has a local copy of the shared memory. For the memory to be scalable, an access should be independent of the time it takes to access the other processors' memories. They proposed that on a read, a PRAM would simply return the value stored in the local copy of the memory. On a write, it would update the local copy first and broadcast the new value to the other processors. Assuming a constant time for initiating a broadcast operation, the goal of making the cost for a read or write constant is thus achieved. In terms of ordering constraints, this is equivalent to requiring that all processors observe the writes from a single processor in the same order while they may disagree on the order of writes by different processors. The following execution history is legal under PRAM but not under CC:

P_1 :	$W(x)1$				
P_2 :		$R(x)1$	$W(x)2$		
P_3 :				$R(x)1$	$R(x)2$
P_4 :				$R(x)2$	$R(x)1$

P_3 and P_4 observe the writes by P_1 and P_2 in different orders, although $W(x)1$ and $W(x)2$ are potentially causally related. Thus, this would not be a legal history for CC.

3.5 Cache Consistency (Coherence)

Cache consistency [Goo89] and coherence [GLL+90] are synonymous and to avoid confusion with causal consistency, we will use the term coherence in this paper.

Coherence is a location-relative weakening of SC. Recall that under SC, all processors have to agree on some sequential order of execution for *all* accesses. Coherence only requires that accesses are sequentially consistent on a *per-location* basis. Clearly, SC implies coherence but not vice versa. Thus, coherence is strictly weaker than SC. The example below is a history that is coherent but not sequentially consistent:

P_1 :	$W(x)1$	$R(y)0$
P_2 :	$W(y)1$	$R(x)0$

Clearly, any serial execution that respects program order starts with writing 1 into either x or y. It is therefore impossible that both read accesses return 0. However, the accesses to x can be linearized into $R(x)0$, $W(x)1$ and so can the accesses to y: $R(y)0$, $W(y)1$. The history is therefore coherent, but not SC. In essence, coherence removes the ordering constraints that program order imposes on accesses to different memory locations.

3.6 Processor Consistency (PC)

Goodman proposed processor consistency in [Goo89]. Unfortunately, his definition is informal and caused a controversy as to what exactly PC refers to. Ahamad et al. [ABJ+92] give a formal definition of PC which removes all ambiguity and appears to be a faithful translation of Goodman's definition. They also show that PC as defined by the DASH group in [GLL+90] is not comparable to Goodman's definition (i.e., it is neither weaker nor stronger). We will not discuss the DASH version of PC except in the context of release consistency (RC) and hence will use PC to refer to Goodman's version and PCD to refer to the DASH version.

Goodman defined PC to be stronger than coherence but weaker than SC. PC can be interpreted as a combination of coherence and PRAM. Thus, every PC history is also coherent and PRAM. However, for a history to be PC it not only has to be coherent and PRAM but those two conditions also must be satisfiable in a mutually consistent way. It is easiest to think of PC as a consistency model that requires a history to be coherent and PRAM *simultaneously*, rather than individually. That is, processors must agree on the order of writes from each processor but can disagree on the order of writes by different processors, as long as those writes are to different locations. The example given for coherence is also PC so we give here a history that fails to be PC (this and the previous example are from [Goo89]):

$$\begin{array}{r|lll} P_1: & W(x)1 & W(c)1 & R(y)0 \\ \hline P_2: & W(y)1 & W(c)2 & R(x)0 \end{array}$$

Notice that P_1 observes accesses in the order:

$$W(x)1, W(c)1, R(y)0, W(y)1, W(c)2,$$

while P_2 observes accesses in the order:

$$W(y)1, W(c)2, R(x)0, W(x)1, W(c)1.$$

That is, P_1 and P_2 disagree on the order of writes to location c. As there is no consistent ordering that would remove this disagreement, the history fails to be PC.

The differences between PC and SC are subtle enough that Goodman claims most applications give the same results under these two models. He also says that many existing multiprocessors (e.g., VAX 8800) satisfy PC, but not sequential consistency [Goo89]. Ahamad et al. prove that the Tie-Breaker algorithm executes correctly under PC while the Bakery algorithm does not (see [And91] for a description of those algorithms). Bershad and Zekauskas [BZ91] mention that processor consistent machines are easier to build than sequentially consistent systems.

3.7 Slow Memory

Slow memory is a location relative weakening of PRAM [HA90]. It requires that all processors agree on the order of observed writes to *each location* by a *single* processor. Furthermore, local writes must be visible immediately (as in the PRAM model). The name for this model was chosen because writes propagate slowly through the system. Slow memory is probably one of the weakest uniform consistency models that can still be used for interprocess communication. Hutto and Ahamad present a mutual exclusion algorithm in [HA90]. However, this algorithm guarantees physical exclusion only. There is no guarantee of logical exclusion. For example, after two processes P_1 and P_2 were subsequently granted access to a critical section and both wrote two variables a and b, then a third process P_3 may enter the critical region and read the value of a as written by P_1 and the value of b as written by P_2. Thus, for P_3 it looks like P_1 and P_2 had had simultaneous access to the critical section. This problem is inherent to slow memory because the knowledge that an access to one location has performed cannot be used to infer that accesses to other locations have also performed. Slow memory does not appear to be of any practical significance.

3.8 Weak Consistency (WC)

Weak consistency is the first and most strict hybrid model we discuss. The model was originally proposed by Dubois et al. [DSB86]. A memory system is weakly consistent if it enforces the following restrictions:

1. accesses to synchronization variables are sequentially consistent and
2. no access to a synchronization variable is issued in a processor before all previous data accesses have been performed and
3. no access is issued by a processor before a previous access to a synchronization variable has been performed

Notice that the meaning of "previous" is well-defined because it refers to program order. That is, an access A precedes access B if and only if the processor that executed access B has previously executed access A. Synchronizing accesses work as *fences*. At the time a synchronizing access performs, all previous accesses by that processor are guaranteed to have performed and all future accesses by that processor are guaranteed not to have performed. The synchronization model corresponding to these access order constraints is relatively simple. A program executing on a weakly consistent system appears sequentially consistent if the following two constraints are observed [AH90, ZB92]:

1. there are no data races (i.e., no competing accesses)
2. synchronization is visible to the memory system

Note that WC does not allow for chaotic accesses as found in chaotic relaxation algorithms. Such algorithms would either have to be changed to avoid data races or it would be necessary to mask chaotic accesses as synchronizing accesses. The latter would be overly restrictive.

3.9 Release Consistency (RC)

Release consistency as defined by Gharachorloo et al. [GLL+90] is a refinement of WC in the sense that competing accesses are divided into acquire, release, and non-synchronizing accesses. Competing accesses are also called *special* to distinguish them from non-competing, ordinary accesses. Non-synchronizing accesses are competing accesses that do not serve a synchronization purpose. This type of access was introduced to be able to handle chaotic relaxation algorithms. An acquire access works like a synchronizing access under WC, except that the fence delays future accesses only. Similarly, a release works like a synchronizing access under WC, except that the fence delays until all previous accesses have been performed. This, for example,

```
worker[p : 1..N] :
  arrived[p] := true                [release]
  do not go[p] → skip od            [acquire]
  go[p] := false                    [ordinary]

coordinator :
  fa i := 1 to N →
    do not arrived[i] →             [nsync]
      skip
    od
    arrived[i] := false             [nsync]
  af
  fa i := 1 to N →
    go[i] := true                   [nsync]
  af
```

Figure 3: Barrier Under Release Consistency

allows (limited) overlap in executing critical sections, which is not possible under WC. Another, more subtle, change is that special accesses are executed under PCD only (not under SC, as in WC).

To make the model more concrete, we give an example of how a critical section and a coordinator barrier could be programmed under RC (see [And91], for example). Below we show how a critical section could be implemented under this model:

```
do test_and_set(locked) → [rd:acquire;wr:nsync]
  skip
od
...critical section...
locked := false              [release]
```

Note the labeling of the read-modify-write operation *test_and_set()*. The read is labeled *acquire*, while the write is labeled *nsync*, which stands for non-synchronizing access. The *acquire* label ensures that no future access is performed before the read has completed and the *nsync* label ensures that the write occurs under PCD. Note that it would be legal but unnecessarily restrictive to mark the write access *release*. The *release* label for the write access resetting the *locked* flag ensures that all accesses in the critical sections are performed by the time the flag is actually reset.

The coordinator barrier is considerably more complicated. The important thing however is that the heart of the barrier is realized by a *release* followed by an *acquire*, while the critical section does just the opposite. Pseudo-code for the barrier is shown in Figure 3. From these examples it should be clear that it is not at all straight forward to write synchronization primitives under RC. However, it is important to realize that such primitives are often written "once-and-forever." That is, the typical programmer doesn't need to worry about labeling accesses correctly as high-level synchronization primitives would be provided by a language or operating system. Also, it is always safe to label a program conservatively. For example, if a compiler has incomplete information available, it could always revert to label reads with *acquire* and writes with *release*.

3.10 Entry Consistency (EC)

The entry consistency model is even weaker than RC [BZ91]. However, it imposes more restrictions on the programming model. EC is like RC except that every shared variable needs to be associated with a synchronization variable. A synchronizing variable is either a lock or a barrier. The association between a variable and its synchronization variable can change dynamically under program control. Note that this, like slow memory, is a location relative weakening of a consistency model. This has the effect that accesses to different critical sections can proceed concurrently, which would not be possible under RC. Another feature of EC is that it refines acquire accesses into exclusive and non-exclusive acquisitions. This, again, increases potential concurrency as non-exclusive acquisitions to the same synchronization variable can be granted concurrently. However, unlike RC, entry consistency is not prepared to handle chaotic accesses. This model is the first that was specifically designed to be implemented as a software shared memory system.

4 Implementations of Memory Consistency Models

An implementation of a memory consistency model is often stricter than the model would allow. For example, SC allows the possibility of a read returning a value that hasn't been written yet (see example discussed under 3.2 Sequential Consistency). Clearly, no implementation will ever exhibit an execution with such a history. In general, it is often simpler to implement a slightly stricter model than its definition would require. This is especially true for hardware realizations of shared memories [AHJ91, GLL+90].

For each consistency model there are a number of implementation issues. Some of the more general questions are:

- What is the consistency unit?
- Enforce eager or lazy consistency?
- Use update or invalidation protocol to maintain consistency?

In hardware implementations the consistency unit is typically a word or a cache line. In software shared memories, the overhead per consistency unit is much higher in absolute terms, so that a memory page or a shared object (structured variable, segment) is often chosen as the consistency unit.

The notion of eager versus lazy maintenance of memory consistency appears to have been invented independently by Borrmann/Herdieckerhoff [BH90] and Bershad/Zekauskas [BZ91]. This notion is based on the observation that the consistency protocol can either be invoked each time an inconsistency *arises* or only when an inconsistency could be *detected*. Eager implementations do the former, lazy the latter. The expected benefit of lazy implementations is that if a process has a cached copy of a shared variable but doesn't access it anymore, then this process does not have to participate in maintaining consistency for this variable. Lazy release consistency [KCZ92] and Midway [BZ91] are two examples of lazy implementations. No performance data is yet available.

5 Influence of Consistency Model on Software

As mentioned earlier, choosing a memory consistency model is a tradeoff between increasing concurrency by decreasing ordering constraints and implementation and programming model complexity. With hybrid models, the memory system is sequentially consistent as long as its synchronization model is respected. That is, the software executing on such a memory system has to provide information about synchronization events to the memory system and its synchronization model must match the memory system's model. Synchronization information is provided by either a programmer in a explicitly concurrent language[2] or by a compiler or its runtime system in a high-level language. Thus, software running on a hybrid memory system has to provide information to execute correctly. However, it is possible and beneficial to go beyond that point. If the software can provide information on the expected *access pattern* to a shared variable, optimizations for each particular access pattern could be enabled resulting in substantially improved performance. Munin [CBZ91] does this by providing a fixed set of sharing annotations. Each annotation corresponds to a consistency protocol optimized for a particular access pattern. A similar approach was taken by Chiba et al. [CKM92] where they annotate Linda programs in order to select an optimized protocol for `in` operations if they are used with certain restrictions. For example, the weakest and most efficient protocol can be used only if, for a tuple with tag t, there is at most one process performing `in` operations and no process performing `read` operations. Unfortunately, so far no performance study of the advantage of such "guided" memory systems has been reported. Carter [CBZ91] indicates that Munin performs well for matrix multiplication and SOR when compared to a hand-coded message passing algorithm, but no comparison with a single-protocol DSM or a strict DSM was reported.

Also note that a change in the consistency model of a memory system can lead to quite subtle changes. For example, Zucker and Baer note that

> *the analysis of* Relax *[a benchmark program] made us realize that how the program is written or compiled for peak performance depends upon the memory model to be used.*

In their example, under SC it was more efficient to schedule a read access causing a cache-miss at the end of a sequence of eight read accesses hitting the cache, while under WC and RC the same access had to be scheduled at the beginning of the read-sequence.

5.1 Chaotic Accesses

Another issue raised by the introduction of weaker consistency models is chaotic accesses (i.e., non-synchronizing competing accesses). Current DSM systems do not handle them well. Neither Munin nor Midway have special provisions for chaotic accesses. Note that algorithms using such accesses often depend on having a "fairly recent" value available. That is, if accesses to variable x are unsynchronized, then reading x must not return *any* previously written value but a "recent" one. For example, the *LocusRoute* application of the SPLASH benchmark does not perform well if non-synchronizing competing accesses return very old values [Rin92, SWG91]. RC maintains such accesses under PCD (which is safe but conservative in many cases). Another type of algorithm using non-synchronizing competing accesses is of the kind where a process needs some of the neighbor's data, but instead of synchronizing with its neighbor, the process computes the value itself and stores it in the neighbors data field. In effect, this type of algorithm trades synchronization with (re-)computation. We would expect having specialized consistency protocols for chaotic accesses could improve the performance of such algorithms.

[2] By "explicitly concurrent language" we mean a language in which it is possible to program synchronization operations.

5.2 Annotating Compilers

Only very little work has been done on annotating parallel programs automatically. In the general case, determining the access patterns to a shared variable is undecidable. It is also unclear exactly what access patterns are useful to distinguish (some work in this direction was done for Munin). However, a language could be designed such that it becomes easier to infer certain aspects of an access pattern. A simple example is a constant object. As there are no write accesses, such objects can be replicated among processes without needing any consistency protocol. Another example is determining whether a critical region contains no write accesses to shared variables. Under EC, this information determines whether a lock can be acquired in exclusive or non-exclusive mode. As critical regions are typically short and do not contain any function calls or unbounded loops, this problem could be decided in most cases.

5.3 Explicitly Parallel Languages

As mentioned above, in an explicitly parallel language the MCM defines the allowable memory-access optimizations. Such a language depends very directly on the memory consistency model as it allows the implementation of synchronization operations. For AC, SC, and PC no special constructs must be available. For WC a *memory-barrier* (or *full fence*) operation would be sufficient. A memory-barrier would have to be inserted in a program wherever consistency of the memory has to be enforced. For RC things become even more complex. Every access would have to be labeled according to its category. With EC, synchronization operations can be implemented based on the locks and barriers provided by the system only. This shows clearly that it is not a good idea to allow a programmer to implement his or her own synchronization primitives based on individual memory accesses. Instead, a language should provide efficient and primitive operations which can then be used to implement higher-level synchronization operations. Maybe locks and barriers as provided under EC would be sufficient. However, for barriers it is not clear whether a single implementation would be sufficient for all possible applications. For example, sometimes it is useful to do some work at the time all processes have joined at a barrier but before releasing them. Under EC, such a construct would have to be implemented with two barriers or in terms of locks; both methods would likely be more inefficient than a direct implementation.

5.4 Implicitly Parallel Languages

Implicitly parallel languages do not have any notion of concurrent execution at the language level. Concurrency control is typically implemented by compiler-generated calls to a the runtime system. Therefore all that needs to be done to adapt to a new MCM is to change the runtime system. As mentioned above, it is still advantageous to integrate the consistency model with the compiler and runtime system more tightly. As the compiler already has information on synchronization and the concurrency structure of the program, it might as well make this information available to the memory system. Jade [RSL92] is a step in this direction. Its runtime system has for each process precise information on the accessed locations and whether a location is only read or also modified. The language also allows one to express that some data will not be accessed anymore in the future.

It is unclear at this point exactly which information can and should be provided to the memory system. It is equally open what information the memory system could provide to the runtime system. The latter, for example, could be useful to guide a runtime system's scheduler based on what data is cheaply available (cached) in the memory system.

6 Conclusions

The central theme of this work is that being memory-model conscious is a good thing. This applies to distributed shared memories, runtime systems, and compilers, as well as languages. We have argued that consistency models are important and that weaker models are beneficial to performance. While there are weakened models that are uniform, they appear to be less promising than hybrid models. Most current work seems to concentrate on the latter. While quite some work has been done in this area, the lack of meaningful performance data is surprising. Also, it appears that in the language, compiler, and runtime-system realms there are still a lot of open questions that could warrant further research. We expect that a tighter coupling between the memory system and the software using it could result in considerable performance improvements.

Acknowledgements

Several people provided useful comments on drafts of this paper: Gregory Andrews, David Lowenthal and Vincent Freeh. Several others provided helpful information on aspects of memory consistency models: Brian Bershad, John Carter, Kourosh Gharachorloo, James Goodman, Bob Janssens, Karen Pieper, Martin Rinard, and Andy Tanenbaum.

References

[ABJ+92] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. Technical Report GIT-CC-92/34, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA, 1992.

[AH90] Sarita Adve and Mark Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[AHJ91] M. Ahamad, P. W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 274–281, May 1991.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Menlo Park, 1991.

[BH90] Lothar Borrmann and Martin Herdieckerhoff. A coherency model for virtually shared memory. In *International Conference on Parallel Processing*, volume II, pages 252–257, 1990.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[BZ91] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, 1991.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Symposium on Operating System Principles*, pages 152–164, 1991.

[CKM92] Shigeru Chiba, Kazuhiko Kato, and Takashi Masuda. Exploiting a weak consistency to implement distributed tuple space. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 416–423, June 1992.

[DSB86] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared memory multiprocessors. *ACM SIGPLAN Notices*, 26(4):245–257, April 1991.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Computer Architecture News*, 18(2):15–26, June 1990.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, March 1989.

[HA90] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 302–311, May 1990.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *SIGARCH Computer Architecture News*, 20(2), May 1992.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[LS88] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[Mos93] David Mosberger. Memory consistency models. *Operating Systems Review*, 17(1):18–26, January 1993.

[Rin92] Martin Rinard, September 1992. Personal communication.

[RSL92] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. September 1992.

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, Burlington, MA, 1992.

[SWG91] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, 1991.

[ZB92] R. N. Zucker and J-L. Baer. A performance study of memory consistency models. *SIGARCH Computer Architecture News*, 20(2), May 1992.