An Overview of Mermera: A System and Formalism for Non-coherent Distributed Parallel Memory*

Abdelsalam Heddaya
heddaya@cs.bu.edu

Himanshu Sinha†
hss@cs.bu.edu

30 September 1992

BU-CS-92-009

Computer Science Department
Boston University
Boston, MA 02215
Phone: 617 353-8919
Fax: 617 353-6457

To appear in: Proceedings of 26th Hawaii International Conference on System Sciences (HICSS-26), Maui, Hawaii, Jan. 5--8, 1993.

*This research was supported in part by NSF under grants IRI-8910195, IRI-9041581 and CDA-8920936. Thanks also to the Advanced Computing Research Facility, Mathematics and Computer Science Division of Argonne National Laboratory for providing access to their TC2000 computer.

†This author was additionally supported in part by NSF under grant CCR-8901647.

Abstract

Several non-coherent memories have been proposed to address the problem of poor scalability of traditional coherent shared memory systems. However, such memories make programming much more difficult than coherent memory.

We give an overview of MERMERA *a system that gives the programmer the choice of coherent* and *non-coherent behavior in the* same *program. We sketch a formal model that describes* MERMERA*'s non-coherent behavior. This model helps us identify a new non-coherent behavior that we call Local Consistency. We also report our measurements from a pilot implementation on a BBN Butterfly which show the response time of pipelined-RAM operations to be an order of magnitude faster than that of coherent operations. Thus, we begin to illuminate a new trade-off in which the programmer can significantly improve shared memory performance at the cost of tolerating varying degrees of non-coherence.*

1 Introduction

In recent years high performance workstations have become cheaper and increasingly interconnected. This has encouraged people to harness their power by using a number of workstations together as a parallel computer to solve problems that previously required powerful computers [6, 23]. Most work in this area uses the message passing method of inter-process communication. While message passing can squeeze the maximum performance from a network of workstations, the shared memory paradigm has the advantage that it hides the number and identities of processors from the programmer, making it easier to program such machines. Attempts have been made to provide the programmer with a shared memory interface [25] but the approach has not been scalable (in the sense of maintaining performance improvement as the number of processors increases) beyond a few tens of machines. Earlier results from work by Deminet [13] reveal the same phenomenon of the early breakdown of the speedup curve for many more applications on distributed memory machines.

We believe that the synchronization overhead needed to keep the shared memory coherent[1] is a major contributor to its poor scalability. Lipton and Sandberg [26] have shown that it is impossible to have coherent shared memory whose access time is less than linear in the worst case delay of the communication network. They then proposed a new non-coherent memory, PRAM[2]. Other non-coherent distributed shared memories have been proposed by Hutto and Ahamad [20], and Weihl and Wang [30]. We expect non-coherent shared memory to be much more scalable than coherent

[1] We will formally define coherence in section 2.1. For the time being, the reader may consider coherence to mean that all processes agree on the order in which they observe memory events.

[2] **Pipelined** Random Access Memory.

shared memory thereby allowing us to use more processors to solve a problem of a given size.

Despite the variety of non-coherent memory types that have been proposed, a uniform formal model within which one can describe and contrast those memories has been lacking. Most memories have been specified in terms of protocols that implement them, making it hard to compare them. A unifying model enables the design of alternative algorithms that demonstrably implement the same memory behavior. Moreover, such a model would permit application programs to be written and reasoned about without reference to the implementation of the underlying shared memory.

In section 2 we present an overview of a formal framework within which we describe the behavior of many of the non-coherent memories previously proposed. As a direct consequence of our development of this model, we identify a new non-coherent memory behavior that we call *local consistency*, that is inspired by the idea of *external consistency* in distributed systems [18]. We did not set out to propose a new memory behavior, and we feel that local consistency should be viewed as a necessary, but not sufficient, condition for a system to satisfy. We have used our formalism to prove that Slow memory, a highly non-coherent memory proposed in [20], is sufficient for a large class of asynchronous iterative methods to solve fixed-point problems [5, 8, 9]. Due to limitation on space we omit this proof and only summarize our formalism in section 2. We refer the interested reader to [16].

The architecture community has also recognized the need to relax consistency constraints for shared memory multiprocessors, resulting in the proposals given in [29, 1, 14]. Systems implementing these weaker constraints have been described in [24, 11]. We have not applied our formalism to describe their correctness conditions, although it would be profitable to do so.

We advocate a system in which the programmer has a choice of different types of memory behaviors, and the choice of mixing such behaviors in the same execution. Munin [12] is one such system. Hybrid consistency [4] is a correctness condition for a system that, roughly speaking, mixes Local Consistency, defined in section 2.2 below, and Linearizability [19].

We are building MERMERA, which gives this choice to the programmer. We describe the memory behavior provided by our system in section 3. In section 4 we describe a pilot implementation that we did on a Butterfly TC2000 to investigate the feasibility of a system like MERMERA. Some preliminary performance measurements are reported in section 5. We summarize our arguments for a system that provides both coherent and non-coherent behavior in section 6.

Notation	Meaning
$x.r_i{:}v$	A *read* event of location x, returning v and uniquely identified by i. We use i to refer to the event's process.
$x.w_i(v)$	A *write* event that writes the value v to location x.
$\alpha, \beta \ldots$	Event variables.
Δ_i	The set of all events in process i.
Δ	The set of all events in a computation. $\Delta = \bigcup_i \Delta_i$
$\mathbf{R}_{\ldots}$	A relation generally on Δ and generally not transitive.
$\mathbf{R}^+_{\ldots}$	irreflexive transitive closure of $\mathbf{R}_{\ldots}$
$\mathbf{R}_i$	*Process program ordering* imposed by i's program on its events, Δ_i. Must be a partial order.
$\mathbf{R}_{wr}$	*Writes-to ordering* between write events and the read events that read their values (def. 1).
$\mathbf{R}$	Global *program ordering*, the union of all process program orderings and the writes-to ordering. $\mathbf{R} = \mathbf{R}_{wr} \cup \bigcup_i \mathbf{R}_i$.

Table 1: Notation used in this paper.

2 Formalism

In this section we give an overview of the formalism that we use to describe different types of shared memory. We attempt to expose the reader to the flavor of our formalism and the interested reader is referred to [16] for its details.

We base our model on memory *events* (*i.e.*, executions of memory operations) rather than on memory states, since the latter are only observable through the former. We describe the correctness conditions of the various kinds of non-coherent memories in terms of the event orderings that they allow. The notation and the definitions we need are summarized in table 1.

An *event* is a particular operation execution on behalf of a particular process. Events are related to each other by various orderings, such as the ordering induced by each process' program, or the ordering induced by information flow between processes. These orderings are defined as needed throughout this section.

A *process* i is a partially ordered set (POSET) of events $\langle \Delta_i, \mathbf{R}_i \rangle$, where $\mathbf{R}_i$ is an irreflexive partial ordering induced by process i's program. The set of all events Δ is given by $\bigcup_i \Delta_i$.

To simplify our presentation, we assume, without loss of generality, that

Assumption 1 *Every value v can be written to each memory location at most once. That is,*

$$x.w_i(v_1) \neq x.w_j(v_2) \implies v_1 \neq v_2.$$

The *writes-to ordering*[3] relates a write event to every read event that returns its value. Formally,

Definition 1 $\mathbf{R}_{wr} \subseteq \Delta \times \Delta$ *is the minimal relation such that, for every pair of write and read events in* Δ *that share the same value* v, $x.w_i(v)\ \mathbf{R}_{wr}\ x.r_j{:}v$.

We can now define a *computation* as follows.

Definition 2 *A computation* E *is a pair* $\langle \Delta, \mathbf{R} \rangle$, *where* $\mathbf{R} = \mathbf{R}_{wr} \cup \bigcup_i \mathbf{R}_i$.

This means that the relations $\mathbf{R}_i$ and $\mathbf{R}_{wr}$ contain all the information needed about a computation in order for us to specify memory behaviors. Furthermore, we view the relation $\mathbf{R}$ as embodying the notion of *potential causality* [21] in the context of read/write shared memory.

Definition 3 *We say that a computation is* **well-formed** *if for every read event* $\beta \in \Delta$ *there exists a write event* $\alpha \in \Delta$ *such that the value read by* β *is written by* α, *i.e.,* $\alpha R_{wr} \beta$.

In this paper we deal with well-formed computations only, which are precisely those that are admissible by *weak* memory [20].

In the following subsections we use the notions of events, program ordering and writes-to ordering defined above, to construct other ordering relations. By constraining these ordering relations in various ways, we arrive at precise formal definitions of a variety of non-coherent shared memory behaviors.

2.1 Coherent memory

In this section we describe executions that we consider to be *coherent*. Intuitively, an execution is coherent if all processes agree on an ordering of all the events in the execution, that is consistent with potential causality.

We define a relation $\mathbf{R}_{ww}$ (figure 1) on Δ, that orders two distinct write events to the same location according to the order they are *observed*. In general, we consider an event α to be observed by another event γ, iff $\alpha\ \mathbf{R}^+\ \gamma$. Intuitively, an event γ observes another α if the value read (or written) by γ could be determined, even partially, by the value read (or written) by α. For example, if a process reads two different values a, b from the same location x in succession, then the corresponding writes would be ordered similarly:

$$x.w(a)\ \mathbf{R}_{ww}\ x.w(b).$$

Figure 1 gives the general definition, in which $\alpha = x.w(a)$, $\beta = x.w(b)$ and $\gamma = x.r{:}b$.

[3] Inspired by the reads-from ordering used in defining *view serializability* of database transactions [28].

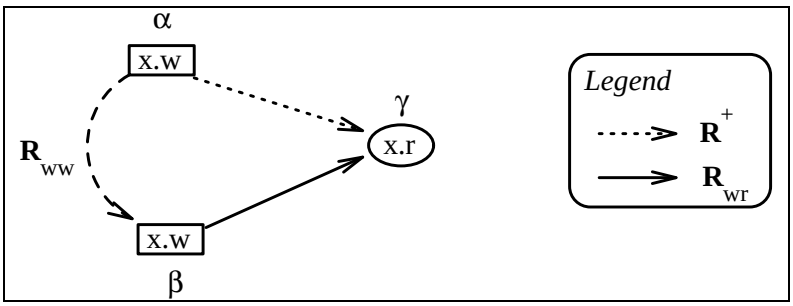

Figure 1: $\alpha\ \mathbf{R}_{ww}\ \beta$ iff $\exists\gamma$ s.t. $\alpha\mathbf{R}^{+}\gamma\ \wedge\ \beta\mathbf{R}_{wr}\gamma$.

Definition 4 *A computation* $E = \langle\Delta, \mathbf{R}\rangle$ *is* coherent (CO) *iff* $\langle\Delta, (\mathbf{R} \cup \mathbf{R}_{ww})^{+}\rangle$ *is a partial order.*

There are other notions of coherence prevalent in the literature. A hierarchy of coherent memories is presented in [20]. The most commonly defined notion is that of sequential consistency (SC). An execution is sequentially consistent if its results are 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 [22]. We define the *result* of an execution to be the values read by the read events. However, shared memory implementations like [25] and coherent caches in shared memory multiprocessors implement stronger conditions like static atomicity (SA) and dynamic atomicity (DA) [20]. In dynamic atomic executions, an event takes effect[4] for all observers at a point in the absolute (or real) time interval that lies between its invocation and its return. In static atomic executions, events are globally ordered according to points that are fixed with respect to the event intervals (*e.g.*, the beginnings or ends of the intervals). Linearizability(LIN) [19] requires the existence of a sequential order as in SC, but this order is restricted to preserve the order (in absolute time) of non-overlapping (again in absolute time) events. It differs from DA in that an event can take effect outside its interval, but not before the end of the latest previous interval, nor after the beginning of the earliest following interval. Linearizability guarantees progress of information flow between processes, at the cost of relying on absolute time in the definition. The condition that that the order of non-overlapping writes be preserved is termed External Consistency in [18]. Sequentially consistent computations are not necessarily externally consistent. For example, a computation in which every process reads only those values that it writes without ever reading values written by other processes satisfies sequential consistency but not external consistency. Therefore such a computation is not linearizable. Since

[4] A write event takes effect at the earliest time its value becomes readable by any process; a read event takes effect at the latest time at which a write event can be observed by it.

external consistency does not say anything about overlapping events, one may not have a sequential order as in SC or LIN. So it is possible to have an externally consistent computation that is not sequentially consistent nor linearizable.

Some of the notions of coherence mentioned above can be related to the different notions of serializability of transactions in Database theory[5]. We will now define sequential consistency in our notation.

Definition 5 *A computation* $E = \langle \Delta, \mathbf{R} \rangle$ *is* sequentially consistent *iff*

1. $\exists$ *a total order* $\mathbf{R}_s$ *over* Δ *such that* $\langle \Delta, \bigcup_i \mathbf{R}_i \cup \mathbf{R}_s \rangle$ *is a partial order, and*

2. $\forall (x.w(v_1)\, \mathbf{R}_{wr}\, x.r{:}v_1), \neg\exists x.w(v_2)$ *such that* $x.w(v_1)\, \mathbf{R}_s\, x.w(v_2)\, \mathbf{R}_s\, x.r{:}v_1$.

The first condition states that the sequence defined by $\mathbf{R}_s$ agrees with the ordering of events specified by each process' program. Since $\mathbf{R}_s$ and $\mathbf{R}$ are defined over the same set of events, the values read by corresponding read events in the computations $E = \langle \Delta, \mathbf{R} \rangle$ and $E' = \langle \Delta, \mathbf{R_s} \rangle$ are the same. The second condition states that a read operation returns the value written by the most recent (according to $\mathbf{R}_s$) write in E'.

We conjecture that sequential consistency is equivalent to our notion of coherence (CO, definition 4), in the sense that any computation permitted by SC is also permitted by CO and vice versa. In [16], we prove the first part, *i.e.*, that a computation that satisfies SC also satisfies CO.

2.2 Non-coherent memories

In this section we discuss computations that violate our definition of Coherence (CO). We consider computations allowed by Causal [20] memory, pipelined random access memory (PRAM) [26], Slow and Weak memories [20] and Multiversion memory (MVM) [30]. Our correctness conditions are defined in terms of relations that we construct on a given computation. None of these relations require information that is not already available in a computation denoted by $E = \langle \Delta, \mathbf{R} \rangle$.

[5] **Relation to Serializability:** Database theory gives us the notion of transaction *serializability* [7, 28]. Variants of serializability include final state serializability (FSR), view serializability (VSR) and conflict serializability (CSR). The two phase locking (2PL) protocol permits a strict subset of the executions allowed by CSR. If we treat each read or write operation on memory as a transaction, then 2PL becomes equivalent to dynamic atomic executions. Linearizability becomes the same as strict serializability, defined in [28]. Sequential consistency becomes equivalent to the conjunction of VSR and Local Consistency (LC).

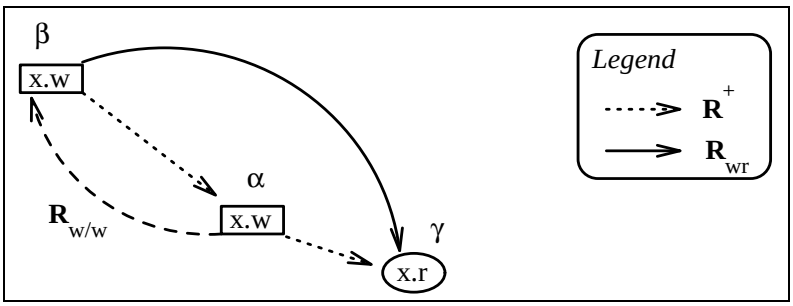

Figure 2: $\alpha \, \mathbf{R}_{w/w} \, \beta$

2.2.1 Causal memory

In [20] Hutto and Ahamad define *causal memory* to be a memory "...that requires all processors to agree on the order of causally related effects (writes) but allows events not related by potential causality (*concurrent* events) to be observed in differing orders."

The notion of potential causality among read and write events is derived from Lamport's definition [21] of the term for message passing systems. In [20] a write and read events are considered similar to message send and receive events, respectively A more precise definition of causal memory is presented by Ahamad et al. in [3].

To help define a causal computation and other interesting non-coherent computations, we introduce two relations $\mathbf{R}_{w/w}$ and $\mathbf{R}_{r/w}$ over Δ.

$\mathbf{R}_{w/w}$ (figure 2) relates two distinct writes α and β to the same location if and only if α follows β in $\mathbf{R}$ but the value written by β is read by a read event that follows α (in $\mathbf{R}^+$). It captures the notion of a write event overwriting the value written by previous (in $\mathbf{R}^+$) writes.

$\mathbf{R}_{r/w}$, illustrated in figure 3 captures a different kind of overwrite semantics than $\mathbf{R}_{w/w}$. While $\mathbf{R}_{w/w}$ relates writes when one of them overwrites the other because of potential causality[6], $\mathbf{R}_{r/w}$ relates a read to a preceding[7] write whose value it does not return, but a subsequent (in $\mathbf{R}^+$) read does.

Definition 6 $\mathbf{R}_{Causal} = \mathbf{R}^+ \cup \mathbf{R}_{r/w} \cup \mathbf{R}_{w/w}$. *A computation is* Causal *iff* $\langle \Delta, \mathbf{R}_{Causal} \rangle$ *is a partial order.*

In figure 4 we give an example of a computation that is Causal but not Coherent (definition 4). It also shows the pairs of events that cause $(\Delta, (\mathbf{R} \cup \mathbf{R}_{ww})^+)$ not to be a partial order. A memory

[6] We say that an event e potentially causes another event e' iff $e\mathbf{R}^+e'$.

[7] *Precede* and *subsequent* refer to the order imposed by potential causality.

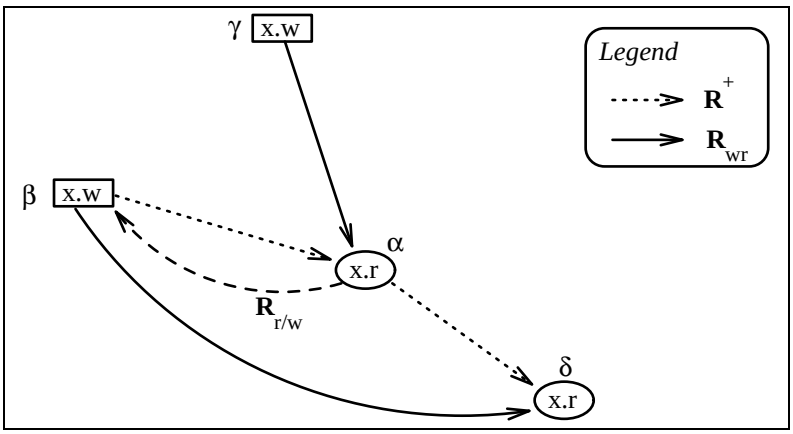

Figure 3: $\alpha \; \mathbf{R}_{r/w} \; \beta$

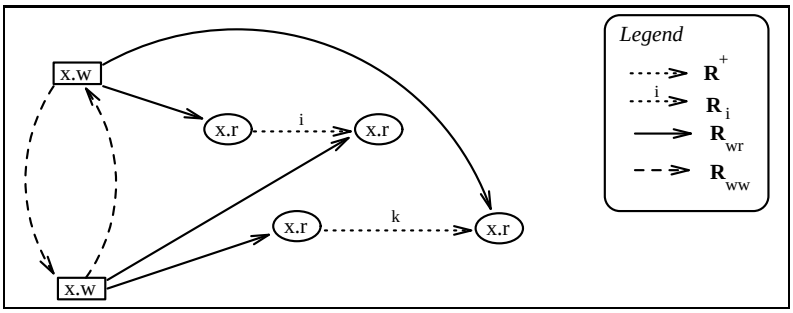

Figure 4: A computation that is Causal but not Coherent.

is Causal iff it permits only causal computations.

2.2.2 PRAM

Pipelined Random Access Memory (PRAM) is proposed in [26]. In a PRAM system each process has a copy of every shared location. A process always reads the local copy. It writes to the local copy and broadcasts the value written to all processes. The writing process does not wait for the broadcast to complete. When a process receives the value broadcast it updates its local copy.

To specify the correctness conditions for PRAM computations in terms of the ordering of events that are allowed, we use the following relations. Let $\Delta_{ij} = \Delta_i \cup \Delta_j$.

Definition 7 $\mathbf{R}_{PRAM}(\Delta_{ij}) = \mathbf{R}^{+} \cup \mathbf{R}_{w/w} \cup \mathbf{R}_{r/w}$, *where the relations are restricted*[8] *to events in* Δ_{ij}. *A computation is a* PRAM *computation iff* $\forall i, j \in P$, *the set of all processes,* $\langle \Delta_{ij}, \mathbf{R}_{PRAM}(\Delta_{ij}) \rangle$ *is a partial order.*

[8] The exact manner in which this restriction is done is described in [16].

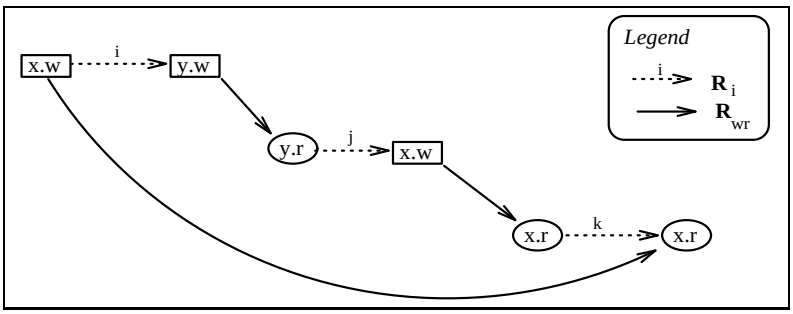

Figure 5: A computation that is allowed by PRAM but is not Causal.

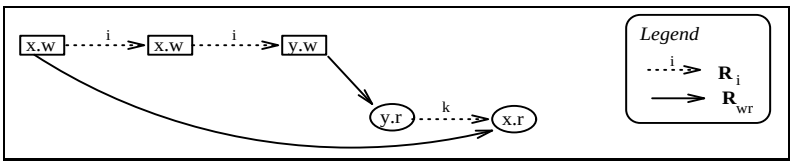

Figure 6: A computation that is Slow but is not allowed by PRAM.

Intuitively, a PRAM computation respects potential causality only between pairs of processes. A computation that is a PRAM computation but is not a causal computation is shown in figure 5. A memory is PRAM iff it permits PRAM computations.

2.2.3 Slow memory

Hutto and Ahamad [20] define Slow memory as follows: "...reads must return *some* value that has been previously written to the location being read. Once a value has been read, no earlier writes to that location (by the processor that wrote the value read earlier) can be returned. Local writes must be immediately visible."

To describe the computations allowed by Slow memory we define Δ_{ijx} as the set of events pertaining to location x and belonging to process i or process j, *i.e.*, $\Delta_{ijx} = \{\alpha \in \Delta_{ij} | \alpha = x.o\}$. Furthermore, Θ^w is the *write-closure* of Θ; it contains all the events in Θ and all other write events in Δ which are read by read operations in Θ.

Definition 8 $\mathbf{R}_{Slow}(\Delta_{ijx}) = (\mathbf{R}^+ \cup \mathbf{R}_{w/w} \cup \mathbf{R}_{r/w})$. *These relations are restricted to events in* Δ_{ijx}. *A computation is a* Slow *computation iff* $\forall i, j \in P, \forall x \in L, \langle \Delta^w_{ijx}, \mathbf{R}_{Slow}(\Delta_{ijx}) \rangle$ *is a partial order.*

In a slow computation the order of all writes to a location by a process is preserved by all processes that share that location. A computation that is Slow but not PRAM is shown in figure 6. A memory is Slow iff it allows Slow computations.

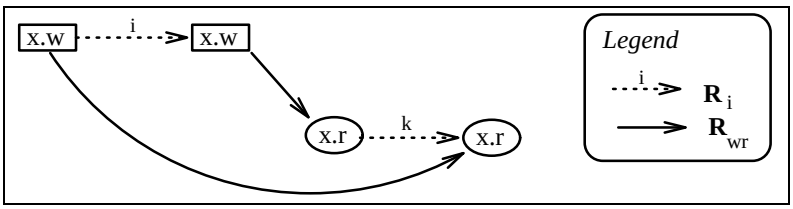

Figure 7: An execution that is Locally Consistent but not Slow.

2.2.4 Locally consistent memory

In this section we define a new condition called *local consistency* (LC). This condition requires that each process observe events as if they were executed on a single processor. This is different from sequential consistency in that SC requires *all* events to occur as if they were executed on one processor while LC requires that only events observed by a process appear to *it* as if they were executed on a single processor. Different processes may observe the same events in different orders.

In the following definition, $\mathbf{R}^{w}_{\Theta}$ is a restriction on $\mathbf{R}$ that contains all pairs in $\mathbf{R}$ that are also in $\Theta \times \Theta$, together with the *writes-to* ($\mathbf{R}_{wr}$) pairs whose reads are in Θ.

Definition 9 $\mathbf{R}_{LC}(\Delta_{ix}) = (\mathbf{R}^{w}_{\Theta})^{+} \cup \mathbf{R}_{w/w}(\Theta^{w}) \cup \mathbf{R}_{r/w}(\Theta)$, *where* $\Theta = \Delta_{ix}$. *A computation is* Locally Consistent *iff* $\forall i, x, \langle \Delta^{w}_{ix}, \mathbf{R}_{LC}(\Delta_{ix}) \rangle$ *is a partial order.*

A computation that is Locally Consistent but that is not slow is shown in figure 7. A memory is Locally Consistent iff it permits Locally Consistent computations.

2.2.5 Weak memory

The last type of computation we describe is that supported by *Weak* memory [20].

Definition 10 *A computation is* Weak *iff* $\forall x.r{:}v \in \Delta$, $\exists x.w(v) \in \Delta$.

Consequently, for every read event, there exists a pair in the writes-to relation ($\mathbf{R}_{wr}$) whose second element is this read event. We adopt this condition as our well-formedness criterion (definition 3), thus, we rule out from consideration any memory behavior that is not Weak. An execution that is Weak but not Locally Consistent is shown in figure 8. A memory is Weak iff it permits only Weak computations.

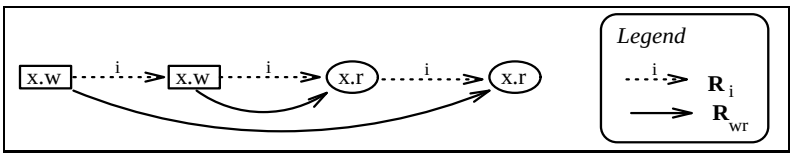

Figure 8: An execution that is Weak but not Locally Consistent.

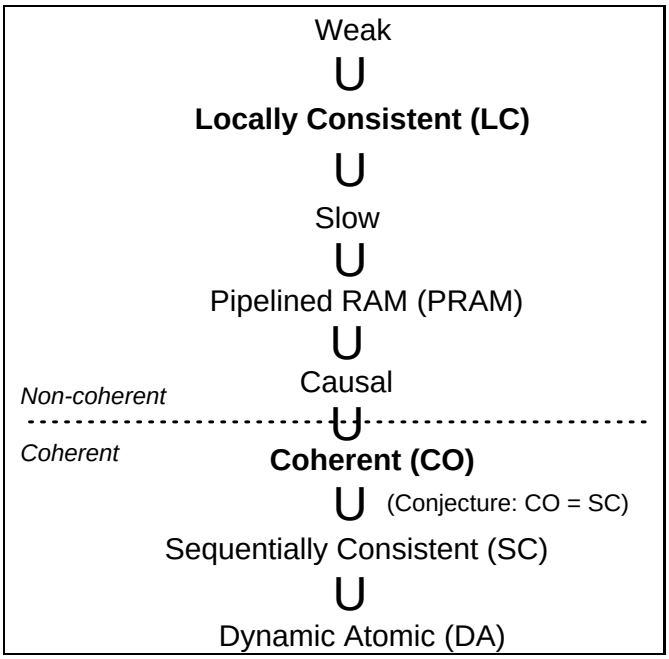

Figure 9: Hierarchy of Distributed Shared Memory.

3 Combining memories

3.1 Why combine memories

In the last section we described a number of types of memories. In [16] we formally prove the hierarchy shown in figure 9. The proofs follow directly from our definitions of the different kinds of memories.

Memories higher (Weak being the highest) in the hierarchy can be implemented with lower latency for read/write operations because the ordering constraints that they have to obey are less global than the constraints that memories lower in the hierarchy have to satisfy. As a result, they are more amenable to buffering and pipelining of operations.

Weak memory should be the most efficient to implement, but we are not aware of any applications that can run on Weak memory alone. Multi-version Memory (MVM) [30] is a memory in which the programmer has access to Weak memory as well as dynamic atomic memory. MVM was used to implement B-trees.

There are classes of programs that can run directly on Slow memory so that one can exploit its potential performance advantages. We show in [16] that certain asynchronous iterative algorithms to find fix points can be directly implemented on Slow memory. We believe that asynchronous iterative graph algorithms (*e.g.*, Bellman-Ford shortest path algorithm) can also be implemented efficiently on Slow memory.

Lipton and Sandberg show in [26] that PRAM can be used to solve a large number of applications like FFT, matrix-vector product, matrix-matrix product, dynamic programming and other computations that are in the large class of *oblivious computations*[9].

The use of Causal memory to solve the traveling salesman problem, the dictionary problem and for solving a system of linear equations is described in [2].

Attiya and Friedman [4] show that any mutual exclusion protocol implemented on some non-coherent memories, must be cooperative in the sense that certain processes must participate regardless of whether they attempt to enter their critical sections or not. They note that traditional mutual exclusion protocols that rely on sequentially consistent memory do not have this requirement. While we believe that the notion of mutual exclusion has little practical relevance to non-coherent memory systems[10], we find this result instructive in that it illustrates what may be a fundamental distinction between coherent and non-coherent memories, with respect to supporting protocols that attempt to achieve global agreement.

The above discussion suggests that different problems are amenable to efficient solutions on different kinds of memory. So, a system should provide both coherent and non-coherent memory behavior to the programmer and this has been the approach in MVM [30] and Munin [12]. MVM offers non-coherent (specifically, Weak) memory behavior, and performance, to applications that can detect harmful memory anomalies and request coherent operations explicitly. Munin supports *release consistency*, which guarantees sequential consistency, but only at synchronization points declared in the program text.

We are building MERMERA, a system which gives the programmer a memory in which a process can treat different memory locations to be of different levels of coherence, and even allow a given location's level to change during program execution.

[9] According to [26] "A computation is oblivious if its data motion and the operations it executes at a given step are *independent* of the actual values of data."

[10] It does not help the computation to know that at most one critical section will be executed at a time, since the values read during this critical section are not guaranteed to be globally current at the beginning of the section, nor are the values written assured of propagation to critical sections that execute subsequently.

3.2 Mermera's behavior

MERMERA provides three types of write operations and one type of read operation. The write operations take a memory *location* and a *value* as arguments and write the value into the location in shared memory. The read operations take a *location* as argument and return a value. We now give an informal description of these operations that does not completely specify the hybrid behavior. A complete formal treatment is deferred to [17]. The write operations are:

CO_Write(Loc, Val): The set of all CO_Writes and their corresponding reads must form a Coherent computation (definition 4).

PRAM_Write(Loc,Val): All PRAM_Writes, together with CO_Writes, and the reads that return the values they write, satisfy the conditions of a PRAM computation (definition 7).

Slow_Write(Loc,Val): All writes and reads constitute a Slow computation (definition 8).

In our system, programs with only CO_Writes execute as if they were executing on Coherent memory. Similarly, programs with only PRAM_Writes (Slow_Writes) execute as if they were executing on PRAM (Slow Memory).

In executions that mix all three kinds of write events, a CO_Write event $x.w(v)$ and a corresponding read event $x.r{:}v$ restrict the entire computation so that every read that observes $x.r{:}v$ must read a value written by a write that is not observed by $x.w(v)$ and vice versa. In other words, one can think of a coherent write event as *cleaving* the computation by forcing all processes to agree on whether other (possibly weaker) writes appear to precede or to follow the CO_Write. PRAM_Write events have a similar effect, except only on the subset of the computation that contains PRAM or weaker writes by the same processor. It turns out that the behavior described can be relaxed further. A complete formalization of this notion of hybrid behavior is forthcoming in [17].

4 Implementation

Having described the behavior of MERMERA, we now present a description of a pilot implementation on a BBN Butterfly TC2000[11] using the Uniform System [27]. We implement PRAM_Write, CO_Write and Read operations. The purpose of this partial implementation, which does not support hybrid computations, is to get a feel for the improvement in memory response time that one may get by tolerating weaker coherence.

[11] An implementation on a network of workstations is under way.

The TC2000 is a distributed memory machine, where a remote memory reference takes about 2.1 microseconds—three times longer than a local memory reference. The particular machine used has 45 processors, which run in a dedicated mode, *i.e.*, one process per processor.

We implement update based protocols for CO_Write and PRAM_Write. Each process has a copy of the shared locations. All writes are broadcast to all processes and reads return the value in the local copy of the location being read.

We have two implementations of CO_Write. The first uses traditional two-phase locking (2PL), and the second employs a novel *pipelined locking* protocol (PLP) that is much more efficient than 2PL under certain loads. These protocols are given in sections 4.1 and 4.2. Section 4.3 describes an algorithm to implement PRAM. All these algorithms are programmed as macros to avoid the penalty of an additional function call for each read or write operation. We find that the PRAM outperforms both the 2PL and PLP implementations of coherent memory, by at least an order of magnitude, even under loads that maximize PLP's ability to pipeline memory requests efficiently.

4.1 Two phase locking (2PL)

We use a traditional 2PL protocol, in which a write obtains locks on all copies of a memory location before it releases any of them, and a read operation locks and reads only the local copy of the memory location. Read and write locks are not distinguished, since concurrent reads never attempt to obtain the same lock. Thus we ensure the serializability of all coherent read and write operations. Sequential consistency is guaranteed by additionally respecting program ordering, by blocking a process that invokes a memory operation until that operation is completed. To avoid deadlocks, we require all processes to obtain the locks in the same fixed order, a technique often termed *conservative* 2PL.

4.2 Pipelined locking protocol (PLP)

In this novel protocol we associate a single lock with the entire copy of the shared memory at each process. When a copy of any shared location in a process has to be modified, the whole copy of shared memory at that process is locked. Writes are propagated to copies in a fixed order, with the lock being acquired before the value is written, and released only after the next copy's lock is acquired. The advantages of this protocol are that shared memory reads do not require any locking, and that write locks are held only for the duration of updating one copy of the memory location. Thus, multiple writes *to the same location* can proceed concurrently in a pipelined fashion, as long

```
#define TIMEOUT 10                          /* Microseconds to wait before retrying for a lock
                                            */
struct MemCopyPtrStruct
{ short *LockPtr;                           /* A lock per memory copy */
  int *LocPtr;};                            /* Pointer to copy of 0^th variable in shared mem-
                                            ory */
struct MemCopyPtrStruct MemCopyPtrs[MAXPROCS];

CO_Write_PLP(int Loc, Val)
{register int i=0, j, k;

  UsLock(MemCopyPtrs[i].LockPtr, TIMEOUT)
  for(i = 1; i < NumProcs; i++)
  { *(MemCopyPtrs[i-1].LocPtr + Loc) = Val;
    UsLock(MemCopyPtrs[i].LockPtr, TIMEOUT);
    UsUnlock(MemCopyPtrs[i-1].LockPtr);
  }
  *(MemCopyPtrs[i-1].LocPtr +Loc) = Val;
  UsUnlock(MemCopyPtrs[i-1].LockPtr);
}

Read(int Loc, *Buffer)
{ *Buffer = *(MemCopyPtrs[UsProc_Node].LocPtr
              + Loc);
}
```

Figure 10: The Pipelined Locking Protocol.

```
struct MemCopyPtrStruct
{ int *LocPtr; }                          /* Pointer to copy 0^th location in shared memory
                                          */
struct MemCopyPtrStruct MemCopyPtrs[MAXPROCS];

PRAM_Write(int Loc, Val)                  /* Use Read from PLP. /*
{ register int i;
  for(i = 0; i < NumProcs; i++)
  *(MemCopyPtrs[i].LocPtr + Loc) = Val;
}
```

Figure 11: The PRAM algorithm.

as they do not require access to the same copy of memory at the same time. This makes PLP suitable for shared memories with many hot spots. Its disadvantage is that concurrent writes to different locations are also pipelined.

The PLP algorithm is given in figure 10, where *UsLock* and *UsUnlock* are atomic operations provided by the Uniform System on BBN Butterfly. *MemCopyPtrs[i]* contains the lock for the copy of shared memory at process i and a pointer to the copy of the 0^{th} location at process i. A *read* operation does not need any locking—it simply returns the value in the local copy of shared memory.

4.3 PRAM algorithm

The algorithm for PRAM_Write, described in figure 11, does not require any locking. The condition that a process' writes be observed in the order they are done at the writer is trivially satisfied because the writer blocks until the value is written in all processes' copies of the shared memory.

5 Preliminary performance results

In this section we present the preliminary findings from our pilot implementation. We measure the performance of PRAM_Write, the two implementations of CO_Write (*i.e.*, 2PL_Write and PLP_Write) and the corresponding Read operations. We give the average of the response times

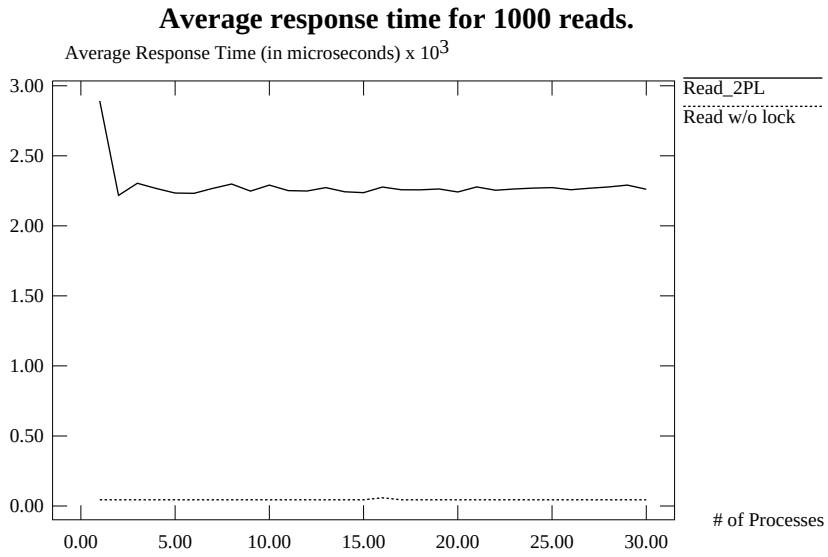

Figure 12: Performance of Read operations. All times are in microseconds.

of all processes for a certain number of *read/write* operations to a single location.[12] Each process does the same operations on the location in shared memory. The total response time seen by each process for a given load is added together and the sum is divided by the number of processes to obtain the average response time for that load.

We report on measurements for three different sets of loads. The first set has 1000 Reads and zero Writes, the second set has 500 Reads and 500 Writes, while the third set has zero Reads and 1000 Writes. We vary the number of processes from 1 to 30, where each process runs on a separate processor. Every measurement was repeated approximately ten times, with negligible variance in the results.

Plots from our measurements are shown in figures 12, 13, and 14. In figure 12 we show the performance of reads that require locking the local copy (*i.e.*, Read_2PL) and of reads that do not (*i.e.*, Read defined in figure 10). This gives us an estimate of the cost of obtaining a local lock and releasing it (about 2.25 microseconds). Consequently, reads can be considered of zero cost relative to writes.

Figures 13 and 14 show the plot of the average response time against the number of processes for two sets of loads. These show the performance of the system under a load of only Write operations

[12]This choice of load, while simplistic, favors coherent memory because it enables PLP to exhibit its best performance.

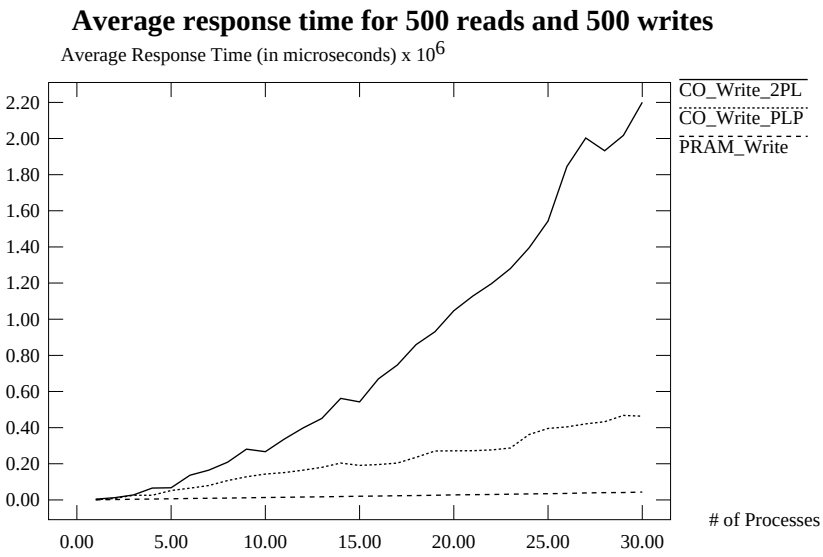

Figure 13: Performance of Write operations. All times are in microseconds.

and under a load in which 50% of the operations are Read operations and 50% are Writes.

In all three methods the cost of writing dominates the cost of doing the corresponding read. Our most important observation is that PRAM_Write is faster[13] than CO_Write by a factor of about 40 in the case of the two phase locking implementation and by a factor of about 10 in the pipelined locking implementation. This indicates that non-coherence can be a performance boon. We expect bigger performance gains in systems where remote updates take longer than three times the local update delay, as is the case in the TC2000. For example, in distributed systems remote updates can take up to 10^3 times longer than local updates.

6 Discussion

In this paper, we formally defined a set of coherent and non-coherent memory behaviors by placing restrictions on a small number of ordering relations. As a side-effect of this exercise, we uncovered a new non-coherent memory that we called Locally Consistent (LC) memory. In section 3.1, we argued for mixing coherent and non-coherent memories, and provided the underpinning for understanding the behavior of such hybrid memories. We speculate that massive parallel speedups on systems with about 10^4 processors (*e.g.*, a few hundred workstations, each having a few tens of

[13] At 30 processors (processes), 500 PRAM Reads and 500 PRAM Writes took 48 milliseconds to finish, while 1000 writes took about 94 milliseconds to conclude.

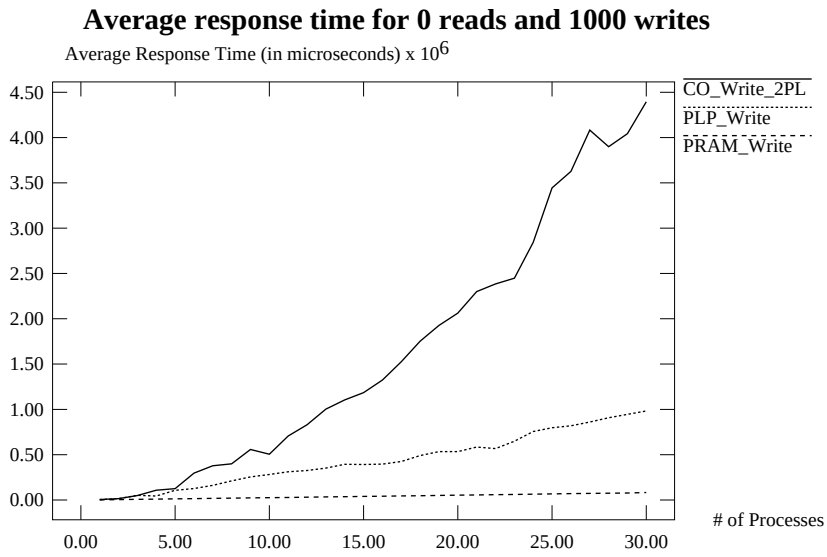

Figure 14: Performance of Write operations. All times are in microseconds.

processors) can be obtained for the significant classes of applications that can tolerate non-coherent memory [16, 2, 26]. On the other hand, we feel that coherent memory facilitates reasoning about parallel programs [15]. Thus, we have designed MERMERA to offer both kinds of behavior so that programs can be developed using coherent memory, and then optimized by modifying portions of the program to exploit non-coherent memory. Our preliminary performance results illuminate parts of a new trade-off in which the programmer can significantly improve shared memory performance at the cost of tolerating varying degrees of non-coherence. We are currently implementing MERMERA on a network of workstations using the Isis toolkit [10] for multicast communication.

Our pilot implementation of MERMERA on a BBN Butterfly TC2000 indicates that non-coherent memory possesses at least a ten-fold advantage over coherent memory in terms of response time. We showed this performance advantage to be robust against significant optimization of coherent memory implementation, and against a low latency differential between remote and local communication. Thus, we expect our forthcoming implementation of Mermera on a (high latency) network of workstations to exhibit much higher performance advantages for non-coherent memories over coherent ones.

Acknowledgments. We wish to thank the anonymous referees, Hagit Attiya and Roy Friedman for valuable comments on earlier drafts of this paper.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proc. 17th Annual International Symposium on Computer Architecture*, May 28–31 1990.

[2] Mustaque Ahamad, Philip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. Technical Report GIT-CC-90-49, Georgia Institute of Technology, School of Information and Computer Science, 1990.

[3] Mustaque Ahamad, Phillip W. Hutto, James E. Burns, and Gil Neiger. Causal memory. In *Proc. of the Fifth International Workshop on Distributed Algorithms*, pages 9–30, Oct. 1991.

[4] H. Attiya and R. Friedman. A correctness condition for high-performance multiprocessors. Technical Report 719, Technion—Israel Institute of Technology, Department of Computer Science, March 1992.

[5] Gerard M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25(2):226–244, April 1978.

[6] Adam Beguelin, Jack Dongarra, Al Geist, Robert Mancheck, and Vaidy Sunderam. A user's guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

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

[8] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[9] Dimitri P. Bertsekas and John N. Tsitsiklis. A survey of some aspects of parallel and distributed iterative algorithms. Technical Report CICS-P-189, Center for Intelligent Control Systems, Cambridge, January 1990.

[10] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Programming with process groups: Group and multicast semantics. Technical Report TR-91-1185, Cornell Univ., Computer Science Dept., Jan. 1991.

[11] Roberto Bisiani and Mosur Ravishankar. Programming the PLUS distributed-memory system. In *Proc. 5th Distributed Memory Computing Conference*, April 8–12 1990.

[12] John B. Carter, John K. Bennett, and Willy Zwaenopoel. Implementation and performance of Munin. In *Proc. 13th ACM Symp. on Operating System Principles, Asilomar, California*, pages 152–164, Oct. 1991.

[13] Jarek Deminet. Experience with multiprocessor algorithms. *IEEE Trans. on Computers*, C-31(4):278–288, April 1982.

[14] Kourosh Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990.

[15] M.G. Gouda. Serializable programs and parallelizable assertions. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business: A Birthday Salute to Edsger W. Dijkstra.* Springer-Verlag, 1990.

[16] Abdelsalam Heddaya and Himanshu S. Sinha. Coherence, Non-coherence and Local Consistency in Distributed Shared Memory for Parallel Computing. Technical Report BU-CS-92-004, Boston University, Computer Sciene Dept., May 1992.

[17] Abdelsalam Heddaya and Himanshu S. Sinha. MERMERA: A System that Combines Coherent and Non-Coherent Memory. Technical report, Boston University, Computer Sciene Dept., 1992. In preparation.

[18] M.P. Herlihy. A quorum-consensus replication method. *ACM Trans. on Computer Systems*, 4(1):32–53, Feb. 1986.

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

[20] P.W. Hutto and M. Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *Proc. 10th IEEE Intl. Conference on Distributed Computing Systems, Paris, France*, June 1990. Also available as Georgia Institute of Technology technical report GIT-ICS-89/39.

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

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

[23] Jerrold S. Leichter and Robert A. Whiteside. Implementing Linda for distributed and parallel processing. Technical Report YALEU/DCS/TR-715, Yale Univ., Dept. of Computer Science, April 1989.

[24] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[25] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, Nov. 1989.

[26] R.J. Lipton and J.S. Sandberg. PRAM: a scalable shared memory. Technical Report CS-TR-180-88, Princeton Univ., Dept. of Computer Science, Sep. 1988.

[27] Mathematics and Computer Science Division, Argonne National Laboratory. *Using the BBN TC2000*, June 1990. Available as ANL/MCS-TM-135.

[28] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, Maryland, 1986.

[29] C. Scheurich and M. Dubois. Concurrent miss resolution in multiprocessor caches. In *Proc. of the 1988 International Conference on Parallel Processing*, August 15-19 1988.

[30] W. E. Weihl and Paul Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proc. IEEE Symposium on Parallel and Distributed Systems, Dallas, Texas*, Dec. 1990.