Coherence, Non-coherence and Local Consistency in Distributed Shared Memory for Parallel Computing*

Abdelsalam Heddaya
heddaya@cs.bu.edu

Himanshu Sinha†
hss@cs.bu.edu

12 May 1992
BU-CS-92-004
Computer Science Department
Boston University
Boston, MA 02215

Abstract

The widespread availability of cheap workstations connected by a network has made possible their collective use as a parallel computer to solve large problems. The problem of implementing shared memory on the high latency networks that connect workstations, without suffering severe degradation in memory access times as the number of workstations increases, poses a special challenge. Traditional coherent memories are not scalable to more than a few tens of machines for most applications. Non-coherent memories that promise good scalability properties have been proposed and studied, in both the distributed systems and parallel architectures communities, albeit in the absence of a single unifying formal framework.

In this paper we present a formal model, based on partial orders, within which we define and relate coherent memory and many of the existing non-coherent memories. Our framework enables us to define the notion of *Local Consistency*, a new non-coherent memory behavior that we view as the weakest constraint that should be required of shared memory.

We briefly argue for mixing coherent and non-coherent memory in the same shared memory system, thus offering the programmer a choice of memory behaviors at different stages of program development or program execution. In a separate paper, we describe MERMERA, a system that develops this approach.

As evidence of the practical utility of non-coherent memory, we include in an appendix, a proof that Slow memory, a known very weak memory, is sufficient for a large class of parallel asynchronous iterative numerical algorithms.

*This research was supported in part by NSF under grants IRI-8910195, IRI-9041581 and CDA-8920936. Thanks also to the Math 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.

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 [BDG$^+$91, LW89]. 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 [LH89] but the approach has not been scalable beyond a few tens of machines. Earlier results from work by Deminet [Dem82] 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 of keeping the shared memory coherent[1] is a major contributor to its poor scalability. Lipton and Sandberg [LS88] have shown that it is impossible to have coherent shared memory whose access time is less than linear in the diameter of the communication network. They then proposed a non-coherent memory, PRAM[2]. Other non-coherent distributed shared memories have been proposed by Hutto and Ahamad [HA90], and Weihl and Wang [WW90]. We expect non-coherent shared memory to be more scalable than coherent 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 has been lacking, within which one can describe and contrast those memories. 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 this paper we present a formal framework within which one can 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 that we call *local consistency*, that is inspired by the idea of *external consistency* in distributed systems [Her86]. We did not set out to propose a new memory behavior, and we feel that local consistency should be viewed as the weakest, but not the only, condition that a system must satisfy. We use our formalism to prove that Slow memory, a highly non-coherent memory proposed in [HA90], is sufficient for a large class of asynchronous iterative methods to solve fixed-point problems [Bau78, BT89, BT90].

The architecture community has also recognized the need to relax consistency constraints for shared memory multiprocessors, resulting in the proposals given in [SD88, AH90, GLL$^+$90]. Systems implementing these weaker constraints have been described in [LLG$^+$92, BR90]. 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. We are designing MERMERA [HS92] to be such a system, in the same vein as Munin [CBZ91].

[1] We will formally define coherence in section 3.1. 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.

Section 2 introduces the notation we use to describe our model. Section 3.1 offers a formal definition of coherence and shows how it compares with Sequential Consistency—a very commonly used notion of coherence [Lam79]. Section 3.2 formally defines a variety of known non-coherent memories. The relationships between these memories is explored in section 4. Section 5 explains the need to combine different kinds of memories in the programming model given to the programmer. We summarize our results in section 6 and discuss our plans for future work in section 7. We prove in appendix A that sequentially consistent memories satisfy our definition of coherence. In appendix B we show that Slow memory suffices to implement *totally asynchronous algorithms* which can be used to solve a large class of *fixed point* problems.

2 Operations and Orderings

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.

In this section, we define the essential components that constitute a computation, and that are general enough to enable the formal description of non-coherent memory behavior. Our model consists of a set of processes P sharing a set of memory locations L, by executing operations on them. In general, we denote an *operation execution* by $x.o(\bar{a}){:}\bar{r}$, where x is the name of the memory location, o stands for the operation (*e.g.*, read or write), and $\bar{a}, \bar{r}$ represent the lists of arguments and results, respectively. An operation execution has an invocation part $x.o(\bar{a})$ and a return part $x.o{:}\bar{r}$.

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 the paper. We denote an event by $x.o_i(\bar{a}){:}\bar{r}$, where i is a unique identifier for the event. With a slight abuse of notation, we will use i to represent the process on behalf of which the event is executed.[3]

In this paper, we restrict our attention to shared memory that supports only read and write operations[4]. Thus, operation executions can take only the forms of $x.r(){:}v$, or $x.w(v){:}\mathrm{OK}$. For brevity, we will henceforth write $x.r{:}v$ and $x.w(v)$.

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 $\bigcup_i \Delta_i$.

A glossary of the notation used in this paper is shown in table 1.

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

[3]The process identifier is typically encoded in the event identifier. For example, a unique identifier for events belonging to sequential processes can be constructed by appending an event sequence number to the process identifier.

[4]It should be interesting to study shared memory that supports arbitrary operations, for we can consider the variety of message passing models to be special cases of shared memory in which the locations are queues with various ordering and composition properties.

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. $\mathbf{R} = \bigcup_i \mathbf{R}_i \cup \mathbf{R}_{wr}$.
$\mathbf{R}_\Theta$	Subset of $\mathbf{R}$ that relates only events in $\Theta \subseteq \Delta$.
Θ^w	*write-closure* of Θ, contains Θ and all write events which are read by events in Θ (def. 5).
$\mathbf{R}^w_\Theta$	$\mathbf{R}_\Theta \cup \{(\alpha,\beta) \vert (\beta \in \Theta) \wedge (\alpha \mathbf{R}_{wr} \beta)\}$, the write-closure of $\mathbf{R}_\Theta$

Table 1: Notation used in this paper.

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*[5] is the last component we need to define our notion of computation. This ordering relates a write event to every other read event that returns its value. Formally,

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

We can now define a *computation* as follows.

Definition 2 *A computation E is a pair $\langle \Delta, \mathbf{R} \rangle$.*

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.

In the following sections 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

[5]Inspired by the reads-from ordering used in defining *view serializability* of database transactions [Pap86].

various ways, we arrive at precise formal definitions of the variety of non-coherent shared memory behaviors that have been proposed. We contrast our definitions with those offered in the literature, most often with inadequate or no formalism.

3 Coherent and Non-Coherent Memories

3.1 Coherent Memories

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. After defining coherence formally we will discuss other notions of coherence and relate our notion of coherence to them.

We first define a projection operator$\mathbf{R}(\Theta)$ that projects pairs from $\mathbf{R}$.

Definition 4 $\forall\Theta \subseteq \Delta, \mathbf{R}(\Theta) = \{(\alpha, \beta \in \Theta) | (\alpha\mathbf{R}\beta)\}$.

For notational convenience we use $\mathbf{R}_\Theta$ to denote $\mathbf{R}(\Theta)$.

We say that a relation constructor $\mathcal{R}$ is *monotonic* iff $\forall\Theta \subseteq \Psi, \mathcal{R}(\Theta) \subseteq \mathcal{R}(\Psi)$.

Observation 1 *The projection operator* $\mathbf{R}(.)$ *is monotonic, i.e.,* $\Theta \subseteq \Psi \subseteq \Delta$ *then* $\mathbf{R}_\Theta \subseteq \mathbf{R}_\Psi$ *which in turn implies* $\mathbf{R}^+_\Theta \subseteq \mathbf{R}^+_\Psi$.

Definition 5 Θ^w *is the* write-closure *of* $\Theta \subseteq \Delta$ *and it is defined as* $\Theta^w = \Theta \cup \{\alpha | (\alpha\mathbf{R}_{wr}\beta) \wedge (\beta \in \Theta)\}$.

Observation 2 *If* $\Theta \subseteq \Psi \subseteq \Delta$ *then* $\Theta^w \subseteq \Psi^w$.

Definition 6 $\mathbf{R}^w_\Theta = \mathbf{R}_\Theta \cup \{(\alpha, \beta) | (\alpha\mathbf{R}_{wr}\beta) \wedge (\beta \in \Theta)\}$.

$\mathbf{R}^w_\Theta$ is different from $\mathbf{R}_{\Theta^w}$ in that it does not include any pair consisting of two writes that do not belong to Θ. For example, consider two writes by process i, $\alpha, \beta \in \Theta^w$ such that $\alpha, \beta \notin \Theta$ and $\alpha\mathbf{R}_i\beta$. In this case $(\alpha, \beta) \in \mathbf{R}_{\Theta^w}$ but not in $\mathbf{R}^w_\Theta$.

We define a relation constructor $\mathbf{R}_{ww}(\Theta)$, that takes a $\Theta \subseteq \Delta$ and returns a set of ordered pairs of distinct write events from Θ^w that write to the same location. The relation returned captures our notion of "observing" write events. An event α is **observed** by another event, β iff $\alpha\mathbf{R}^+\beta$. $\mathbf{R}_{ww}(\Theta)$ returns ordered pairs of write events according to the order they are observed.

Definition 7 $\mathbf{R}_{ww}(\Theta)$ *is defined on* Θ^w *such that* $(\alpha = x.w_i(v_1))\mathbf{R}_{ww}(\Theta)(\beta = x.w_j(v_2))$ *iff* $\exists((\gamma = x.r_k{:}v_2) \in \Theta)|\ (\alpha\mathbf{R}^+_{\Theta^w}\gamma) \wedge (\beta\mathbf{R}_{wr}\gamma)$.

This relation orders two writes to the same location in the order they are observed by process k. Note that not all pairs of write events to a location will be ordered by this relation. $\mathbf{R}_{ww}$ is illustrated in figure 1.

Lemma 1 $\mathbf{R}_{ww}(.)$ *is monotonic, i.e., if* $\Theta \subseteq \Psi \subseteq \Delta$ *then* $\mathbf{R}_{ww}(\Theta) \subseteq \mathbf{R}_{ww}(\Psi)$

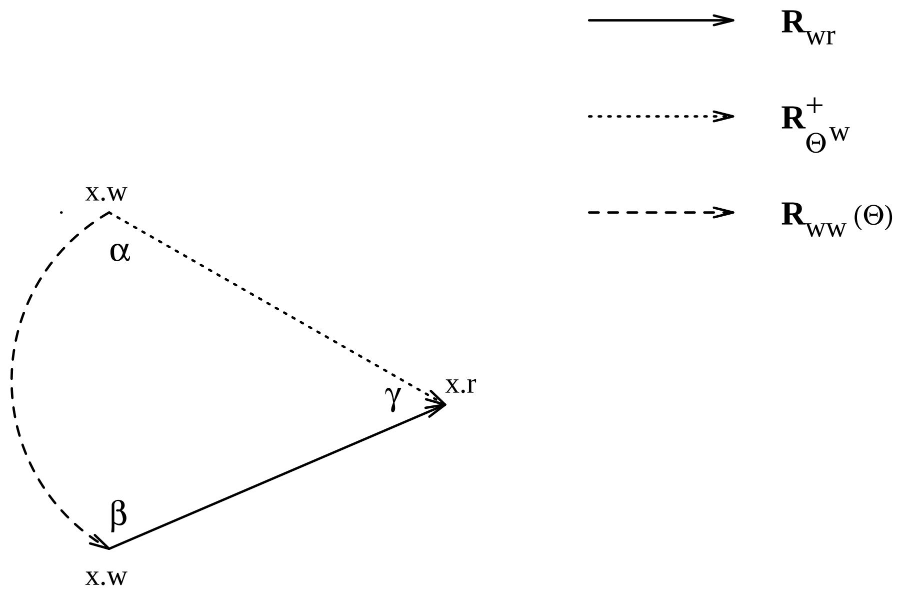

Figure 1: An example of $\alpha\mathbf{R}_{ww}(\Theta)\beta$.

Proof:

Let $(\alpha,\beta) \in \mathbf{R}_{ww}(\Theta)$.

$$\begin{aligned}
&(\alpha,\beta) \in \mathbf{R}_{ww}(\Theta)\\
&\Rightarrow \exists\gamma \in \Theta | (\alpha\mathbf{R}^{+}_{\Theta}\gamma) \wedge (\beta\mathbf{R}_{wr}\gamma) \wedge (\alpha,\beta \in \Theta^{w}) \text{ (Def. 7)}\\
&\Rightarrow \exists\gamma \in \Psi | (\alpha\mathbf{R}^{+}_{\Theta}\gamma) \wedge (\beta\mathbf{R}_{wr}\gamma) \wedge (\alpha,\beta \in \Theta^{w}) \text{ (Because } \Theta \subseteq \Psi)\\
&\Rightarrow \exists\gamma \in \Psi | (\alpha\mathbf{R}^{+}_{\Psi}\gamma) \wedge (\beta\mathbf{R}_{wr}\gamma) \wedge (\alpha,\beta \in \Psi^{w}) \text{ (Observations 1, 2)}\\
&\Rightarrow (\alpha,\beta) \in \mathbf{R}_{ww}(\Psi) \text{ (Def. 7)}
\end{aligned}$$

Definition 8 *A computation $E = \langle\Delta,\mathbf{R}\rangle$ is* coherent (CO) *iff $(\Delta,(\mathbf{R}\cup\mathbf{R}_{ww}(\Delta))^{+})$ is a partial order.*

There are other notions of coherence prevalent in the literature. A hierarchy of coherent memories is presented in [HA90]. The most commonly defined notion is that of sequential consistency (SC). An execution is sequentially consistent if its results were 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 [Lam79]. We define the *result* of an execution to be the values read by the read events. However, shared memory implementations like [LH89] and coherent caches in shared memory multiprocessors implement stronger conditions like static atomicity (SA) and dynamic atomicity (DA) [HA90]. In static atomic executions operations take effect for all observers at some specific component event. In dynamic atomic executions operations take effect at any point (in absolute time) during the operation interval as long as the resulting history is equivalent to some serial execution. External consistency (EC) of an execution E requires the existence of a sequential order as in SC but this order is restricted to preserve the order of non-overlapping (in absolute time) operations in E. It differs from DA in that an operation can take effect outside the interval (in absolute time) defined by its invocation and its return. The difference between SC and EC is illustrated in figure 2. External consistency can be used to guarantee progress of information flow between processes but this is not true for

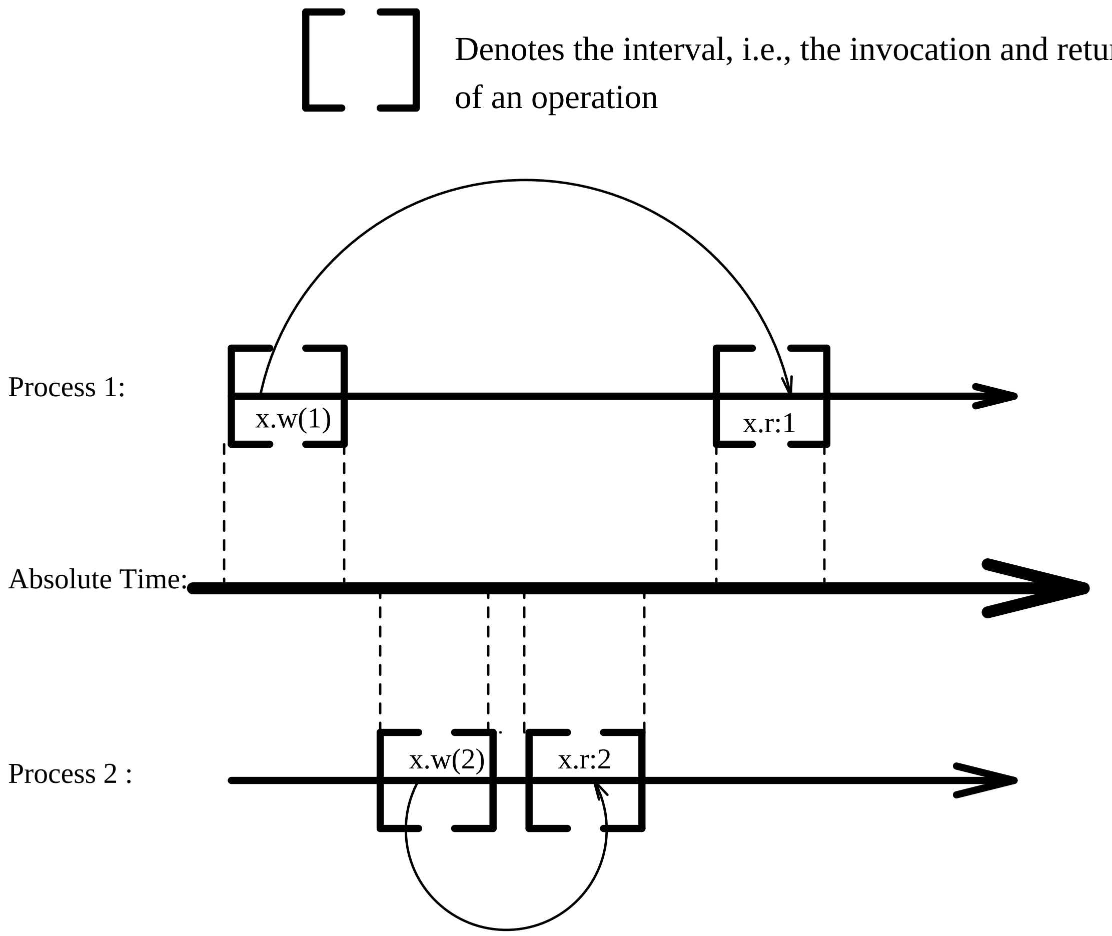

Figure 2: An execution that is sequentially consistent but not externally consistent.

sequential consistency because a process can continue to read only those values that were written by itself and make progress in executing its program without ever reading values written by other processes. It is possible to construct an equivalent serial execution given such an execution.

The description of sequential consistency above allows a process to read values that have not yet (again in absolute time) been written. Hutto and Ahamad define *realizable sequential consistency* that excludes such executions.

Database theory gives us the notion of serializability [BHG87, Pap86] of transactions. Some of the various notions of serializability are final state serializability (FSR), view serializability (VSR) and conflict serializability (CSR). The two phase locking (2PL) protocol allows a subset of the executions allowed by CSR. If we treat each read or write operation on memory as a transaction then 2PL allows only static atomic executions with the release of a lock being the component event at which an operation takes effect. The notion of external consistency is the same as that of strict serializability in [Pap86]. VSR in conjunction with local consistency (LC) (Def 15 below) is the same as sequential consistency.

We will now define sequential consistency in our notation.

Definition 9 *A computation $E = \langle \Delta, \mathbf{R} \rangle$ is* sequentially consistent *iff*

1. $\exists$ *a total order* $\mathbf{R}_s$ *over* $\Delta | (\Delta, \bigcup_i \mathbf{R}_i \cup \mathbf{R}_s)$ *is a partial order and*
2. $\forall x.w(v_1) \mathbf{R}_{wr} x.r{:}v_1 \; \nexists x.w(v_2) | x.w(v_1) \mathbf{R}_s x.w(v_2) \mathbf{R}_s (x.r{:}v_1)$.

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 (SC, Def. 9) is equivalent to our notion of coherence (CO, Def. 8) in the sense that any computation permitted by SC is also permitted by CO and vice versa. In appendix A we prove that a computation that satisfies SC also satisfies CO.

3.2 Non-Coherent Memories

In this section we discuss computations that violate our definition of Coherence (CO). We consider computations allowed by Causal [HA90] memory, pipelined random access memory (PRAM) [LS88], Slow and Weak memories [HA90] and Multiversion memory (MVM) [WW90]. 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 computations denoted by $E = \langle \Delta, \mathbf{R} \rangle$.

3.2.1 Causal Memory

In [HA90] 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."

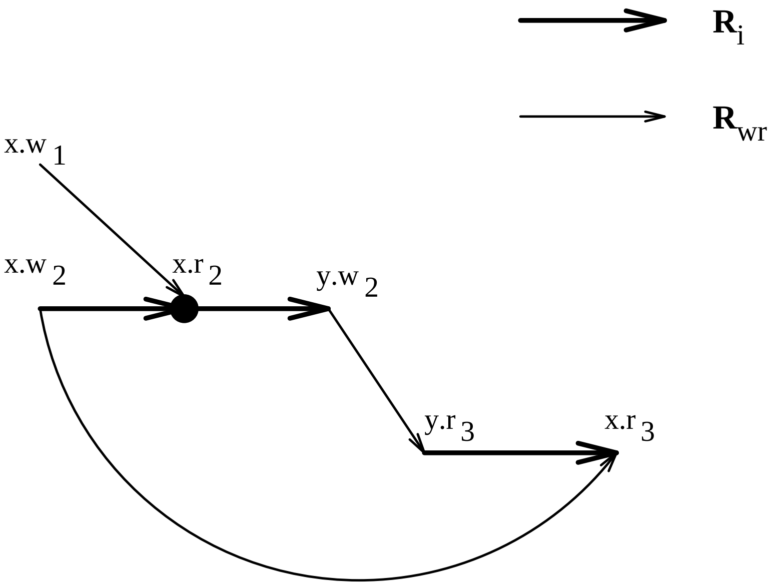

Figure 3: Execution that violates overwrite semantics of memory. $x.w_1$ and $x.w_2$ are not related by potential causality but the value that $x.r_3$ reads has been overwritten.

The notion of potential causality is derived from Lamport's definition [Lam78] of the term for message passing systems. In [HA90] a write event is related to a message-send and a read event is related to a message-receive.

This interpretation has two problems. First, it fails to capture the overwrite semantics of memory and permits the computation shown in figure 3 that should not be permitted.

Secondly, the notion of an order of observation of events is not clear. A more precise definition of causal memory is presented by Ahamad et al. in [AHJ90].

To give a clear definition of a causal computation we define two relation constructors $\mathbf{R}_{w/w}$ and $\mathbf{R}_{r/w}$ which take a set of events $\Theta \subseteq \Delta$ and return ordered pairs from Θ^w.

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

Definition 10 $\mathbf{R}_{w/w}(\Theta) = \{(\alpha = x.w.(v_1)), (\beta = x.w.(v_2)) | (\alpha, \beta \in \Theta) \wedge \exists \gamma = x.rrm{:}v_2 \in \Theta | \beta \mathbf{R}_\Theta^+ \alpha \mathbf{R}_\Theta^+ \gamma\}$

Note that $\beta \mathbf{R}_{wr} \gamma$. It is illustrated in figure 4.

Lemma 2 *With α, β, γ as above, $(\alpha, \beta) \in \mathbf{R}_{w/w}(\Theta) \Rightarrow \alpha \mathbf{R}_{ww}(\Theta) \beta$.*

This follows directly from the definition of $\mathbf{R}_{ww}$. Note that the converse is not necessarily true.

$\mathbf{R}_{r/w}$, illustrated in figure 5 captures a different kind of overwrite semantics than $\mathbf{R}_{w/w}$. While $\mathbf{R}_{w/w}$ relate writes when one of them overwrites the other because of potential causality, $\mathbf{R}_{r/w}$ relates a read with a preceding[6] write that is subsequently read.

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

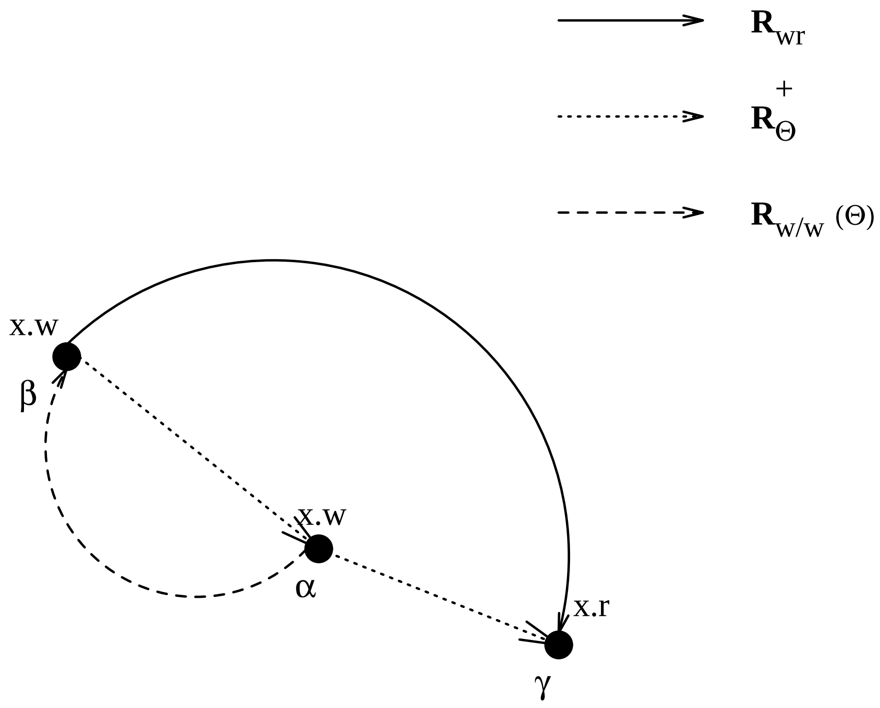

Figure 4: $\alpha \mathbf{R}_{w/w}(\Theta)\beta$

Definition 11 $\mathbf{R}_{r/w}(\Theta) = \{(\alpha = x.r{:}v_1, \beta = x.w(v_2)) | (\alpha \in \Theta, \beta \in \Theta^w) \wedge \exists((\gamma = x.w(v_1) \in \Theta^w) \wedge (\delta = x.r{:}v_2) | (\beta(\mathbf{R}^w_\Theta)^+ \alpha \mathbf{R}^+_\Theta \delta))\}$

Note that $\gamma \mathbf{R}_{wr} \alpha$ and $\beta \mathbf{R}_{wr} \delta$.

Lemma 3 *With $\alpha, \beta, \gamma, \delta$ as above, $(\alpha, \beta) \in \mathbf{R}_{r/w}(\Theta) \Rightarrow (\beta \mathbf{R}_{ww}(\Theta)\gamma) \wedge (\gamma \mathbf{R}_{ww}(\Theta)\beta)$.*

Lemma 4 *$\mathbf{R}_{w/w}$ and $\mathbf{R}_{r/w}$ are monotonic.*

Definition 12 $\mathbf{R}_{Causal} = \mathbf{R}^+ \cup \mathbf{R}_{r/w}(\Delta) \cup \mathbf{R}_{w/w}(\Delta)$. *A computation is* causal *iff $(\Delta, \mathbf{R}_{Causal})$ is a partial order.*

In figure 6 we give an example of a computation that is causal but not coherent (def. 8). It also shows the pairs of events that cause $(\Delta, (\mathbf{R} \cup \mathbf{R}_{ww})^+)$ to not be a partial order. A *memory is causal* iff it permits causal computations only.

3.2.2 PRAM

Pipelined Random Access Memory (PRAM) is proposed in [LS88]. 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 ordering of events that are allowed, we use the following relations. Let $\Delta_{ij} = \Delta_i \cup \Delta_j$.

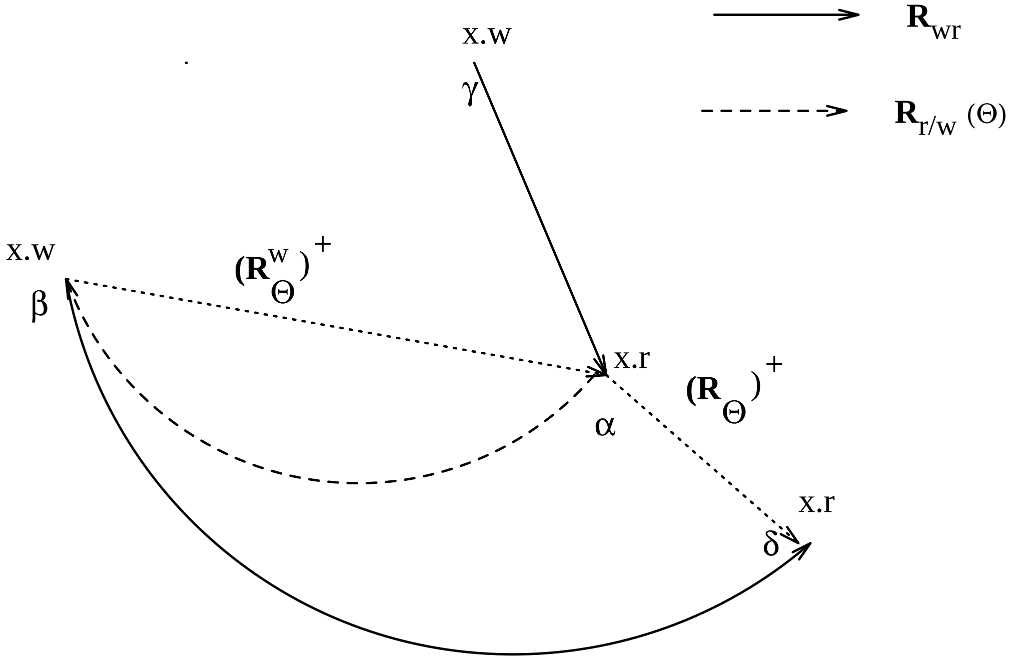

Figure 5: $(\alpha, \beta) \in \mathbf{R}_{r/w}(\Theta)$

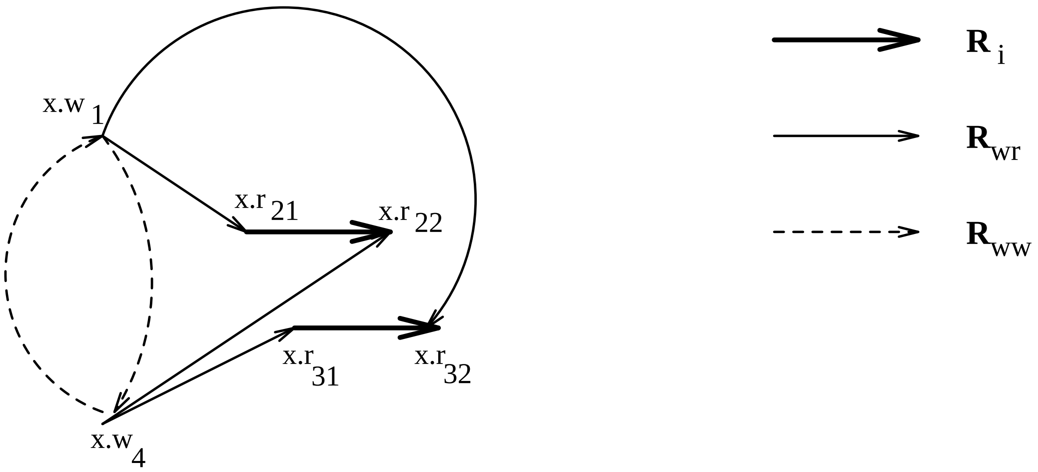

Figure 6: A computation that is causal but not coherent.

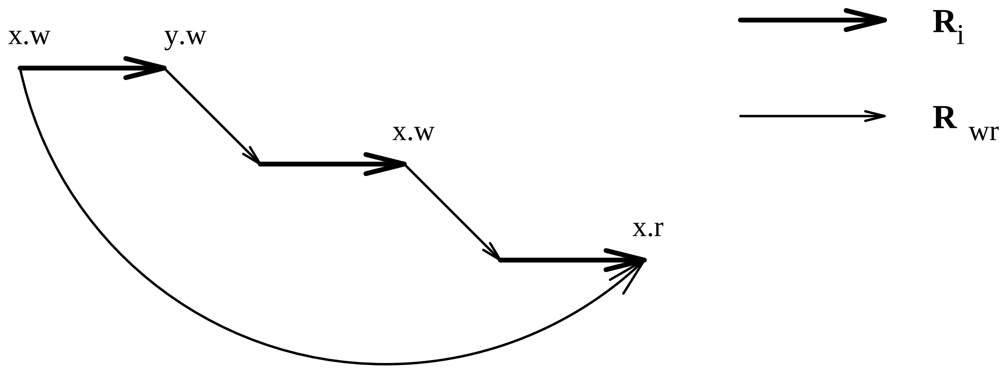

Figure 7: A computation that is allowed by PRAM but is not causal.

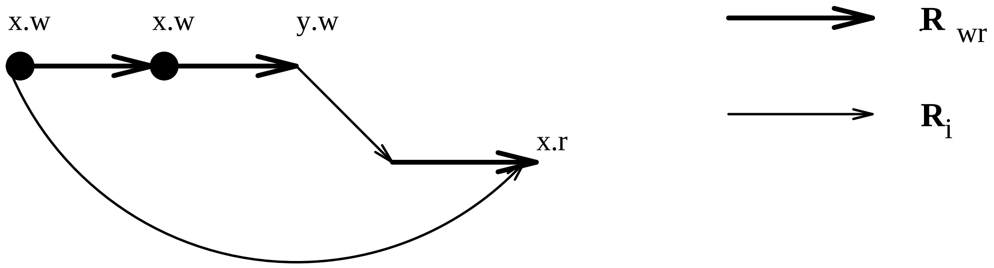

Figure 8: A computation that is Slow but is not allowed by PRAM.

Definition 13 $\mathbf{R}_{PRAM}(\Delta_{ij}) = (\mathbf{R}^w_\Theta)^+ \cup \mathbf{R}_{w/w}(\Theta) \cup \mathbf{R}_{r/w}(\Delta_i))$, *where* $\Theta = \Delta_{ij}$. *A computation is a* PRAM *computation iff* $\forall i, j \in P$*the set of all processes,* $(\Delta^w_{ij}, \mathbf{R}_{PRAM}(\Delta_{ij}))$ *is a partial order.*

A computation that is a PRAM computation but is not a causal computation is shown in figure 7. A memory is *PRAM* iff it permits only PRAM computations.

3.2.3 Slow Memory

Hutto and Ahamad [HA90] 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} the set of events pertaining to location x and belonging to process i and process j, *i.e.*, $\Delta_{ijx} = \{\alpha \in \Delta_{ij} | \alpha = x.o\}$.

Definition 14 $\mathbf{R}_{Slow}(\Delta_{ijx}) = (\mathbf{R}^w_\Theta)^+ \cup \mathbf{R}_{w/w}(\Theta) \cup \mathbf{R}_{r/w}(\Delta_{ix})$, *where* $\Theta = \Delta_{ijx}$. *A computation is a* Slow *computation iff (*$\forall i, j \in P \wedge \forall x \in L(\Delta^w_{ijx}, \mathbf{R}_{Slow}(\Delta_{ijx}))$ *is a partial order.*

A computation that is Slow but not PRAM is shown in figure 8. A *memory is Slow* iff it allows Slow computations only.

Figure 9: An execution that is *locally consistent* but not *slow*.

3.2.4 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 (SC, def 9) 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.

Definition 15 $\mathbf{R}_{LC}(\Delta_{ix}) = (\mathbf{R}^w_\Theta)^+ \cup \mathbf{R}_{w/w}(\Theta^w) \cup \mathbf{R}_{r/w}(\Theta))$*, where* $\Theta = \Delta_{ix}$*. A computation is* Locally Consistent *iff* $\forall i, x(\Delta^w_{ix}, \mathbf{R}_{LC}(\Delta_{ix}))$ *is a partial order.*

A computation that is Locally Consistent but that is not slow is shown in figure 9. A *memory is locally consistent* iff it permits only locally consistent computations only.

3.2.5 Weak Memory

The last type of computation we describe is allowed by *Weak* memory of [HA90]. It requires that a read return any value that is previously written by a write operation. The notion of previous is not defined.

Definition 16 *A computation is* Weak *iff* $\forall(\alpha \in \Delta | \alpha.o = read)\exists(\beta \in \Delta)|\beta\mathbf{R}_{wr}\alpha$.

This is the same condition as our well-formedness condition (def. 3). An execution that is Weak but not locally consistent is shown in figure 10. A memory is weak iff it permits only Weak computations.

A *memory is Weak* iff it allows Weak computations only.

4 Relating Memories

In this section we relate the various kinds of memory we have discussed so far in this paper. For ease of reference we summarize the correctness conditions of the different types of memories in table 2.

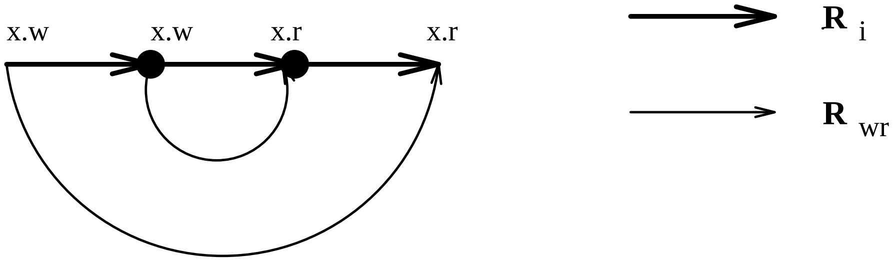

Figure 10: An execution that is Weak but not locally consistent.

Memory Type	Correctness Condition
Coherent	$(\Delta, (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^+)$ is a partial order.
Causal	$(\Delta, (\mathbf{R}^+ \cup \mathbf{R}_{w/w}(\Delta) \cup \mathbf{R}_{r/w}(\Delta)))$ is a partial order.
PRAM	$\forall i, j, ((\Delta^w_{ij}, ((\mathbf{R}^w_{\Delta_{ij}})^+ \cup \mathbf{R}_{w/w}(\Delta_{ij}) \cup \mathbf{R}_{r/w}(\Delta_i)))$ is a partial order).
Slow	$\forall i, j, x((\Delta^w_{ijx}, ((\mathbf{R}^w_{\Delta_{ijx}})^+ \cup \mathbf{R}_{w/w}(\Delta_{ijx}) \cup \mathbf{R}_{r/w}(\Delta_{ix})))$ is a partial order).
LC	$\forall i, x((\Delta^w_{ix}, ((\mathbf{R}^w_{\Delta_{ix}})^+ \cup \mathbf{R}_{w/w}(\Delta^w_{ix}) \cup \mathbf{R}_{r/w}(\Delta_{ix})))$ is a partial order).

Table 2: Summary of correctness conditions for different types of memories

We establish parts of the hierarchy shown in figure 11. In the figure, an $A \rightarrow B$ implies that all computations allowed by A are allowed by B and there exists at least one computation that is allowed by B but not by A.

Theorem 1 *As shown in figure 11,* $CO \rightarrow Causal \rightarrow PRAM \rightarrow Slow \rightarrow LC \rightarrow Weak$.

Proof: From lemma 2 and lemma 3 we can conclude that if $\mathbf{R}_{Causal}$ is not a partial order then $\mathbf{R}_{coherent}$ is not a partial order. Therefore any computation that is not causal cannot be coherent (*i.e.*, a coherent computation is causal). We showed a computation that is causal but not coherent in figure 6. Therefore, $CO \rightarrow Causal$.

From the monotonicity property of $\mathbf{R}_{w/w}$ and $\mathbf{R}_{r/w}$ (lemma 4) it is obvious that if $\mathbf{R}_{PRAM}(\Delta_{ij})$ is not a partial order for some i, j then $\mathbf{R}_{Causal}$ cannot be a partial order because $\Delta_{ij} \subseteq \Delta$. Therefore any computation that is not a PRAM computation cannot be causal (*i.e.*, all causal computations are PRAM computations too). In figure 7 we showed a PRAM computation that is not causal. Therefore, $Causal \rightarrow PRAM$.

Since $\Delta_{ijx} \subseteq \Delta_{ij}$ we can conclude from the same monotonicity property as above that if $\mathbf{R}_{Slow}(\Delta_{ijx})$ is not a partial order then $\mathbf{R}_{pram}(\Delta_{ij})$ is not a partial order. Therefore a computation that is not slow is not a PRAM computation either. In other words all PRAM computations are slow computations. As shown in figure 8, the converse is not true. Therefore $PRAM \rightarrow Slow$.

The argument to show that a locally consistent computation must be slow is slightly different. The first and third terms in the definition of $\mathbf{R}_{LC}(\Delta_{ix})$ are subsets of the corresponding terms of $\mathbf{R}_{Slow}(\Delta_{ijx})$. The second term in $\mathbf{R}_{LC}(\Delta_{ix})$ is $\mathbf{R}_{w/w}(\Delta^w_{ix})$ while the second term in $\mathbf{R}_{Slow}(\Delta_{ijx})$ is $\mathbf{R}_{w/w}(\Delta_{ijx})$. If there exists $(\alpha, \beta) \in \mathbf{R}_{w/w}(\Delta^w_{ix})$ such that $\mathbf{R}^w_{\Delta_{ix}})^+ \cup \mathbf{R}_{w/w}(\Delta^w_{ix})$ is not a partial order (implying that the computation is not locally consistent) then there are two possibilities. We

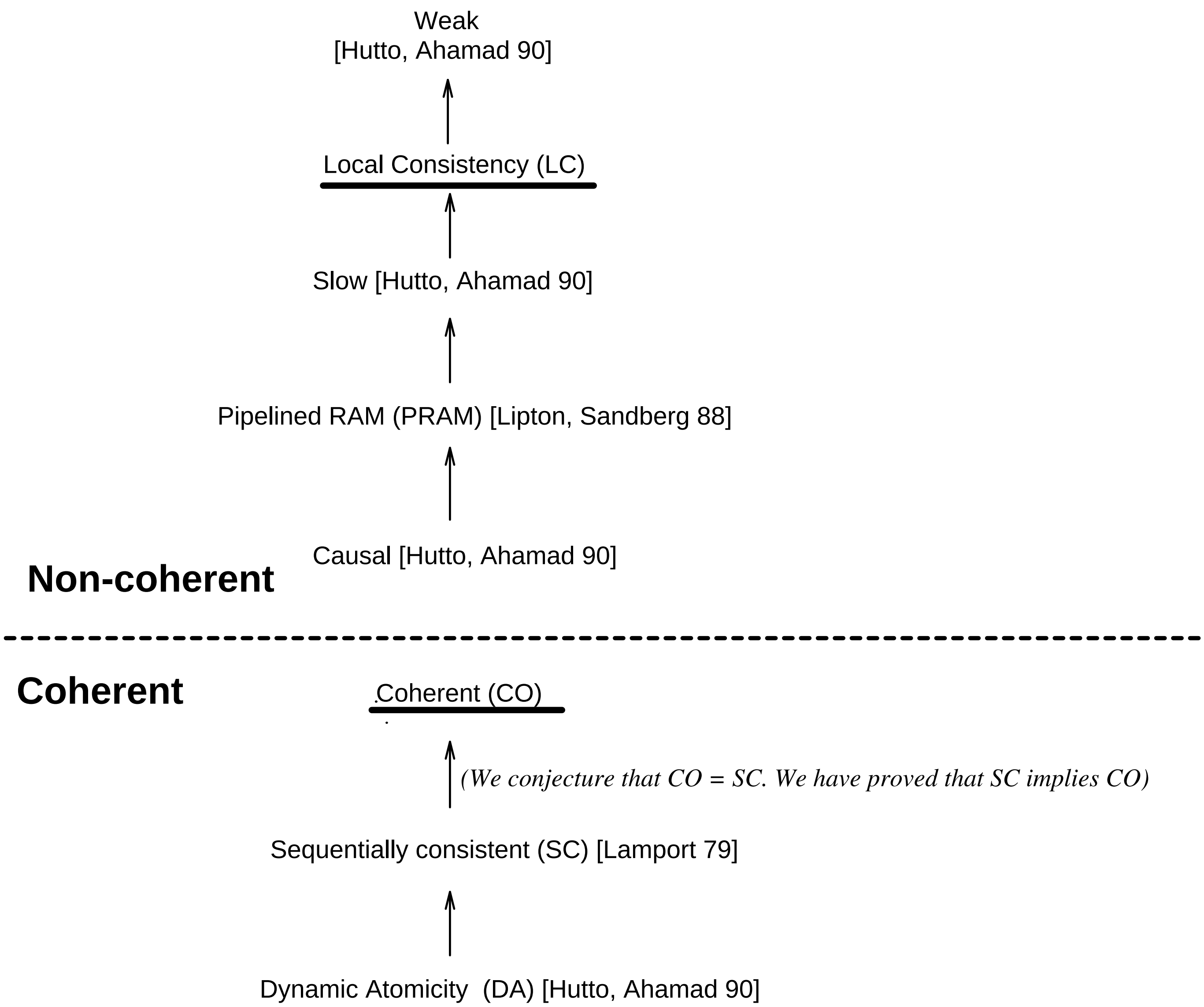

Figure 11: Hierarchy of Memories

show that in both cases there will be a j such that $(\Delta^w_{ijx}, \mathbf{R}_{Slow}(\Delta_{ijx}))$ will not be a partial order.

1. $\beta, \alpha \in \Delta_{ix}$ in which case $(\alpha, \beta) \in \mathbf{R}_{w/w}(\Delta_{ijx})$, *or*
2. $\alpha \in \Delta_{ix}, \beta \in (\Delta^w_{ix} - \Delta_{ix})$ in which case, $\exists j | (\mathbf{R}^w_{\Delta_{ijx}})^+ \cup \mathbf{R}_{w/w}(\Delta_{ijx})$ is not a partial order. The j is the process to which β belongs.

Therefore, a computation that is not locally consistent is not Slow either.

5 Mixing Memories in the Same System

We have described a number of types of memories. What should be made available to the programmer? 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) [WW90] 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.

In [HA90], Hutto and Ahamad show how a memory that handles exactly one operation at a time (*i.e.*, serially) can be simulated by distributed processors using Slow memory. But we want to focus on classes of programs that can run directly on Slow memory so that one can exploit its *potential* performance advantages. We show in appendix B 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 [LS88] 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*[7]. They also say that in these computations, whenever the actual computation dominates the synchronization[8] the overhead, PRAM is much more efficient than Sequentially Consistent memory.

The use of Causal memory to solve the traveling salesman problem, the dictionary problem solving a system of linear equations is described in [AHJ90].

This discussion illustrates that different problems are amenable to efficient solutions on different kinds of memory. So, a system should provide both coherent and non-coherent memories to the programmer and this has been the approach in MVM and Munin [CBZ91]. Munin provides *release consistency* which guarantees that the memory will run programs that require sequential consistency only if the program clearly identifies all synchronization points in the program.

We plan to provide a system which gives the programmer a memory in which a process can treat different variables to be of different types[9] and even allow a given variable's type to change during program execution. The behavior of such a hybrid memory system can be hard to characterize. We address this problem in our design of MERMERA [HS92].

[7] "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." [LS88]

[8] Synchronization in the algorithm *not* in the memory

[9] By type, we mean the type of memory correctness condition it satisfies.

6 Summary

In this paper we presented a formal framework in which we described a number of existing correctness conditions for non-coherent memory. It was very simple to prove the relationships among the different memories. We introduced a new definition of coherence that we prove in appendix A to be strictly weaker than Sequential Consistency, and conjecture to be equivalent to it. We identified the notion of *local consistency* which we believe should be the weakest property that all shared memory systems must satisfy. Our formalism can be used to prove correctness and equivalence of shared memory computations. Memories can be specified in terms of the formalism and programers can write their programs without knowledge of the memory implementation.

In appendix B we show that any shared memory system that implements Hutto and Ahamad's Slow memory is sufficient to solve the large class of fixed point problems amenable to solution via totally asynchronous algorithms [BT89].

7 Future Work

We are working to extend our formalism to describe systems which support more than one of the memories described in this paper. MERMERA, a system that combines coherent memory with non-coherent memory is being designed. It will be implemented on a BBN TC 2000, a distributed memory NUMA machine and on a network of workstations using Isis. We will concentrate on a run time system that uses update based protocols. Preliminary results of a pilot implementation on the TC2000 revealed that for certain memory reference patterns the average latency on a PRAM memory was less than the average latency on an atomic memory implemented by two phase locking by a factor of ≈ 40. We are encouraged by this preliminary finding because on the TC2000 we could not propagate the writes asynchronously, *i.e.*, a write had to wait for the value to be propagated to all copies before it could return. On a network of workstations we will be able to buffer writes and we will also be able to propagate the values asynchronously. We expect the factor to be much higher.

But how much of this improvement in memory latency contributes towards the improvement of the overall application is still an open question. We plan to implement some sample applications on MERMERA to attempt an answer to this question.

References

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 28-31 1990.

[AHJ90] Mustaque Ahamad, Philip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. Technical Report GIT-CC-90-49, College of Computing, Georgia Institute of Technology, 1990.

[Bau78] Gerard M. Baudet. Asynchronous iterative methods for multiprocessors. *JACM*, 25(2):226–244, April 1978.

[BDG+91] 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.

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

[BR90] Roberto Bisiani and Mosur Ravishankar. Programming the PLUS distributed-memory system. In *Proceedings of the Fifth Distributed Memory Computing Conference*, IEEE Computer Society Press, Los Alamitos, California, April 8-12 1990.

[BT89] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[BT90] 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.

[CBZ91] John Carter, John Bennet, and Willy Zwaenopoel. Munin .. In *Proc. 13th ACM Symp. on Operating System Principles,*, October 1991.

[Dem82] Jarek Deminet. Experience with multiprocessor algorithms. *IEEE Trans. on Computers*, C-31(4):278–288, April 1982.

[GLL+90] 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 17th Annual International Symposium on Computer Architecture*, May 28-31 1990.

[HA90] 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 GATech technical report GIT-ICS-89/39.

[Her86] M.P. Herlihy. A quorum-consensus replication method. *ACM Trans. on Computer Systems*, 4(1):32–53, Feb. 1986.

[HS92] Abdelsalam Heddaya and Himanshu S. Sinha. MERMERA:A System that Combines Coherent and Non-Coherent Memory. Technical report, Computer Science Department, Boston University, 1992. Under preparation.

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

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

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, Nov. 1989.

[LLG+92] Daniel Lenoski, James Lauson, Kourosh Gharachorloo, Wolf-Deitrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LS88] 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.

[LW89] Jerrold S. Leichter and Robert A. Whiteside. Implementing Linda for Distributed and Parallel Processing. Technical Report YALEU/DCSTR-715, Department of Computer Science, Yale University, April 1989.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.

[SD88] C. Scheurich and M. Dubois. Concurrent miss resolution in multiprocessor caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, August 15-19 1988.

[WW90] W. E. Weihl and Paul Wang. Multi-version memory: Software cache management for concurrent b-trees. May 1990.

A Sequential Consistency implies Coherence

Lemma 5 *Any computation that is sequentially consistent (SC) is also coherent (CO).*

The definitions of SC and CO are reproduced here.

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

1. $\exists$ a total order $\mathbf{R}_s$ over $\Delta | (\Delta, \bigcup_i \mathbf{R}_i \cup \mathbf{R}_s)$ is a partial order and
2. $\forall ((x.w(v_1)\mathbf{R}_{wr}x.r{:}v_1))\ \nexists x.w(v_2) | x.w(v_1)\mathbf{R}_s x.w(v_2)\mathbf{R}_s x.r{:}v_1)$.

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

Proof: We prove our lemma by proving its contra-positive, *i.e.*, by proving that any computation that is not coherent is not sequentially consistent which in turn is proved by contradiction.

Let us assume that given a non-coherent computation $E = \langle \Delta, \mathbf{R} \rangle$, there exists a total order $\mathbf{R}_s$ on Δ that satifies the conditions of sequential consistency.

For any computation (coherent or not) the following observations can be made. From condition 1 of the definition of SC,

$$(\alpha, \beta) \in \mathbf{R} \Rightarrow (\alpha, \beta) \in \mathbf{R}_s \tag{1}$$

From condition 2 of the definition of SC, and the definition of $\mathbf{R}_{ww}(.)$ (def. 7),

$$(\alpha, \beta) \in \mathbf{R}_{ww}(\Delta) \Rightarrow (\alpha, \beta) \in \mathbf{R}_s \tag{2}$$

This is because $\alpha \mathbf{R}_{ww}(\Delta)\beta$ requires the existence of a read γ that *observes* α but *reads* β which rules out $\beta \mathbf{R}_s \alpha$ because this would violate the second condition in the definition of SC.

Therefore,

$$((\alpha,\beta) \in (\Delta, (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^{+})) \Rightarrow (\alpha,\beta) \in \mathbf{R}_s \tag{3}$$

From the definition of coherence we can conclude that if a computation is non-coherent then $(\Delta, (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^{+})$ is not a partial order which implies that there exists a pair of events α and β such that both (α,β) and $(\beta,\alpha) \in (\Delta, (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^{+})$.

From predicate 3 above, this implies that $(\alpha,\beta) \in \mathbf{R}_s \wedge (\beta,\alpha) \in \mathbf{R}_s$ which violates our assumption that $\mathbf{R}_s$ is a total order.

Therefore, if a computation is not coherent then a total order satisfying the conditions of SC does not exist and a non-coherent computation is not sequentially consistent.

B Totally Asynchronous Iterations on Slow Memory

In this section we show that under certain conditions Slow memory is sufficient for the correctness *totally asynchronous iterative algorithms* summarized in [BT89, BT90]. Such algorithms find a fixed point x of the function $f(x)$ where x is a vector of length n. The i^{th} component of x is denoted by x_i. For simplicity we assume that we have n processors and that the ith processor, denoted by P_i computes x_i.

Let $x_i(t)$ be the value of x_i available to P_i at time t, where t is an integer variable used to index events of interest in the system. T^i is the set of times at which x_i is updated in the following manner.

$$x_i(t+1) = \begin{cases} f_i(x_1(\tau_1^i(t)), x_2(\tau_2^i(t)), \ldots, x_n(\tau_n^i(t))) & \forall t \in T^i \\ x_i(t) & otherwise \end{cases}$$

where $0 \leq \tau_j^i(t) \leq t$.

$\tau_j^i(t)$ is an integer that denotes the the version of x_j used by P_i in computing f_i.

It has been shown in [BT90] that if the instance of the problem satisfies certain conditions and the total asynchronism assumption is satisfied then the iteration described above will converge. The total asynchronism assumption states that

1. the sets T^i are infinite, and,
2. if $\{t_k\}$ is a sequence of T^i which tends to infinity then $\lim_{k\to\infty} \tau_j^i(t_k) = \infty$. Note that if $t_k \to \infty$ iff $t \to \infty$.

We will now show that under certain conditions the total asynchronism assumption is satisfied by a system using Slow memory to store x.

The condition above that the sets $T^i s$ be infinite can be satisfied by the program implementing this iteration. A simple infinite loop that keeps computing f_i satisfies this condition.

To prove the second condition we first show that,

$$\tau_j^i(t+1) \geq \tau_j^i(t)$$

We prove this by contradiction.

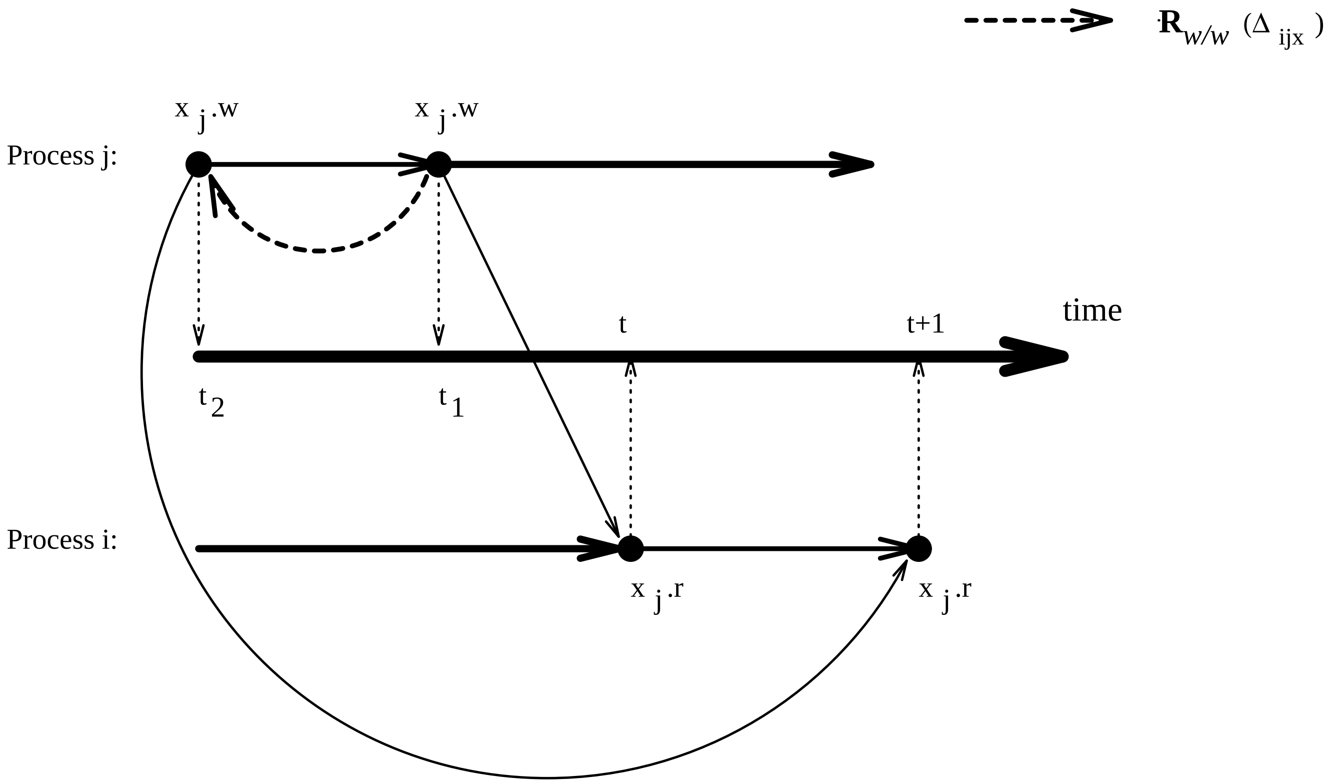

Figure 12: A computation in which $\tau_j^i(t) = t_1 > \tau_j^i(t+1)$. In this figure $\tau_j^i(t) = t_1$ and $\tau_j^i(t+1) = t_2$

Let us assume that there exist i, j and t such that $\tau_j^i(t) > \tau_j^i(t+1)$. Let $\tau_j^i(t) = t_1$ and $\tau_j^i(t+1) = t_2$. An execution that allows this will contain the computation segment shown in figure 12. This is not permitted by Slow memory (def. 14) because the the order imposed by $\mathbf{R}_{w/w}(\Delta_{ijx})$ violates the definition of Slow memory. Therefore, an execution in which $\tau_j^i(t) > \tau_j^i(t+1)$ will not be permitted by Slow memory. This shows that $\tau_j^i(t)$ is a monotonically non-decreasing function. If an implementation of Slow memory guarantees progress, i.e., a write by a processor eventually reaches all other processors then $\tau_j^i(t)$ is guaranteed to increase. Since x_j is written infinitely often (because of condition 1), $\lim_{t\to\infty} \tau_j^i(t) \to \infty$

Examples of iterations that converge when the total asynchronism assumption is satisfied include,

1. A linear system of equations, $x = Ax+b$ such that the spectral radius of A, $\rho(|A|) < 1$. [Bau78]

2. The gradient optimization problem provided the Hessian is diagonally dominant. [BT89]

3. All dynamic programming algorithms.

A comprehensive list of fixed point problems that converge in a totally asynchronous iteration can be found in [BT89].