THE UNIVERSITY OF CALGARY

Ambiguous Memory Consistency Models

by

Nathaly Verwaal

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

AUGUST, 1998

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-35005-3

Canada

Abstract

To enhance performance, multiprocessors incorporate sophisticated memory structures, which allow processes to have inconsistent views of memory, and can result in unexpected program outcomes. A memory consistency model is a set of guarantees describing constraints on the outcome of a program.

This thesis uses a unifying framework that facilitates the description, analysis, and comparison of memory models to formalize several interpretations of two ambiguous memory models: pipelined RAM and processor consistency.

Lipton and Sandberg described a machine that implements pipelined RAM [LS88]. This thesis presents precise descriptions of the three possible interpretations of this machine with an equivalent formal memory model for each.

The VAX 8800 and Stanford's DASH machine implement versions of processor consistency, originally defined by Goodman [Goo89], but their descriptions are ambiguous. A precise description of possible interpretations with matching formal memory models is presented.

I determine, for each pair of models presented, whether one implies the other or whether they are incomparable.

Acknowledgements

I want to thank Lisa Higham for her enormous amount of support and encouragement. She got me interested in graduate studies and supplied much financial, emotional and intellectual support. I want to thank Wayne Eberly for the use of his computer, which allowed me to work at home when that was the only place I was able to do any work. I want to thank Jobien van der Meer and Holly Robbins for the excellent child care they provided, allowing me to work without distractions. And last but not least, a thanks to my parents, Jan Verwaal and Narda van der Meer, for the financial and emotional support given over such long distances.

Contents

List of Tables

List of Figures

List of Computations

List of Relations

For any computation of system (P, J) with set of actions O, $(o_1, o_2) \in (O, \xrightarrow{ppo})$ iff $o_1 \xrightarrow{proc} o_2$, and either

1. o_1, o_2 are both read or both write actions, or
2. o_1 is a read and o_2 is a write action, or
3. o_1, o_2 are actions to the same object, or
4. $\exists o'$ such that $o_1 \xrightarrow{ppo} o' \xrightarrow{ppo} o_2$ 78

For any computation of system (P, J) with set of actions O and for any $S = \{<_p\}$, a set of sequences over the set of actions O such that S contains exactly one sequence for each $p \in P$, $(o_1, o_2) \in (O, \overset{semi(S)}{\longmapsto})$ iff $\exists q, r \in P$ and $\exists x \in J$ such that

1. $o_1, o_2 \in O|q$ and $o_1 \xrightarrow{ppo} o_2$, or
2. $o_1 \in O_w|r$, $o_2 \in (O_r|q)|x)$ with return value v, and $\exists o' \in (O_w|r)|x$ with value v such that $o_1 \xrightarrow{ppo} o'$, or
3. $o_1 \in (O_r|q)|x$, $o_2 \in O_w|r$, and $\exists o' \in (O_w|r)|x$ such that $o_1 <_{L_q} o' \xrightarrow{ppo} o_2$, or
4. $\exists o' \in O$ such that $o_1 \overset{semi(S)}{\longmapsto} o' \overset{semi(S)}{\longmapsto} o_w$ 78

List of Definitions

6.3.1 Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCGharachorloo if for each processor $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ such that

1. $(O|p \cup O_w, \overset{r-prog}{\longrightarrow}) \subseteq (O|p \cup O_w, <_{L_p})$, and
2. $\forall x \in J$
   (a) $((O|p)|x, \overset{prog}{\longrightarrow}) = ((O|p)|x, <_{L_p})$, and
   (b) $\forall q \in P\ (O_w \cap O|x, <_{L_p}) = (O_w \cap O|x, <_{L_q})$, and
3. $(O, \overset{\widehat{pcd}(\{<_{L_q}|q\in P\})}{\longmapsto})$ is cycle-free 63

6.3.8 Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCKohli if for each processor $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ such that

1. $\forall x \in J$ and $\forall q \in P\ (O_w \cap O|x, <_{L_p}) = (O_w \cap O|x, <_{L_q})$, and
2. $(O|p \cup O_w, \overset{semi(\{<_{L_q}|q\in P\})}{\longmapsto}) \subseteq (O|p \cup O_w, <_{L_p})$ 79

6.3.9 Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCAhamad if

1. $\forall x \in J$ there is some linearization $(O|x, <_{L_x})$ such that $(O|x, \xrightarrow{prog}) \subseteq (O|x, <_{L_x})$, and

2. $\forall p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ such that

   (a) $(O|p \cup O_w, \overset{semi(\{<_{L_q}|q \in P\})}{\longmapsto}) \subseteq (O|p \cup O_w, <_{L_p})$, and

   (b) $\forall x \in J (O_w|x, <_{L_p}) = (O_w|x, <_{L_x})$, and

3. $(O, \overset{wo}{\longmapsto})$ is cycle-free 79

6.3.17 Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCDash if for each processor $p \in P$ there is some total order $(O|p \uplus O_w, \xrightarrow{view_p})$ such that

1. $(O_w, \xrightarrow{prog}) \subseteq (O_w, \xrightarrow{view_p})$, and

2. $\forall x \in J$ and $\forall q \in P$ $(O_w|x, \xrightarrow{view_p}) = (O_w|x, \xrightarrow{view_q})$, and

3. $(O|p, \xrightarrow{prog}) = (O|p, \xrightarrow{view_p})$, and

4. if $w \in O_w|p$ then $w_{O|p} \xrightarrow{view_p} w_{O_w}$, and

5. $<_{L_p} = ((O|p \uplus O_w) \backslash (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p})$ is a linearization, and

6. $(O, \overset{pcd(\{<_{L_q}|q \in P\}, \{\xrightarrow{view_q}|q \in P\})}{\longmapsto})$ is cycle-free 93

Chapter 1

Introduction

To enhance performance, multiprocessors incorporate sophisticated memory structures. There might be one or more copies of the shared memory and the entire memory might be in one location or distributed throughout the system. Memory structures may be replicated through constructs such as caches and write buffers, or may use advanced interconnection patterns including multiple buses. Many of these architectural features allow processes to have inconsistent views of memory, which, in turn, can result in unexpected program outcomes.

A memory consistency model is a set of guarantees describing constraints on the outcome of a multiprocessor program. Fewer guarantees allow more performance optimizations but yield complex machines that are difficult to program.

Sequential consistency is the memory consistency model that is generally assumed and that programmers prefer since a distributed system implementing sequential consistency most closely resembles a sequential computer. The easiest visualization of a system implementing sequential consistency is a set of processors interacting through a single globally shared memory where only one processor can access the memory at a time. However, since most distributed systems use much more complicated memory systems, most distributed systems have a much weaker memory consistency model. These systems generally supply the programmer with synchronization primitives such as locks. When these primitives are used correctly, the outcome of a program is the same as the outcome would be when executed on a sequentially con-

sistent machine. The use of such primitives is generally expensive. A programmer needs to understand the memory consistency model of the system being programmed and thus needs a precise description of the memory model of the underlying machine in order to construct correct and efficient programs.

This thesis uses a unifying framework developed at the University of Calgary [HKV98, HKV97] to facilitate the description, analysis, and comparison of memory consistency models. Precision is essential for providing an unambiguous view of the logical behavior of a memory system. Unification provides a common basis with which memory consistency models can be compared. The framework achieves both of these goals while remaining simple.

Using the framework, an actual machine that implements a distributed system and an abstract representation can be described. A distributed system may contain speedup devices such as caches, duplicate copies of the memory, buffers, and any operation may pass through several of these devices. An actual machine is described by stating the sequence of events that happen when an operation is executed and by giving the constraints on the interleaving of these events. The abstract representation of a distributed system, on the other hand, only considers the *outcome* of an execution of a machine. Only the completed operations, not the individual events, are considered. The formal definition of a memory consistency model describes limitations on the view that each processor in the system has of the operations in an execution.

It is generally easier to compare the abstract representations of memory consistency models, instead of the machine descriptions. Two machines might implement the same memory consistency model, using different constructs. Thus an opera-

tion might be executed by going through a completely different set of events on one machine from that on the other. Hence the two machines would be difficult to compare. However, because the formal description only considers operations, the formal description of the memory models of the two machines are easier to compare. When the formal definitions of two memory models are equivalent this implies that any computation arising from an execution on one machine could also arise from an execution on the other machine, even though these might arise from two distinct executions.

Memory consistency models that are derived from a machine description in this thesis are interpreted as restrictions on the sequence of events that can occur on the machine. Then a formal definition of the memory model that the machine implements is presented. For some of the memory consistency models that were not derived from a machine description, a machine is supplied that would implement the memory model. The description of a machine, M, and a memory consistency model, D, are shown to correspond by establishing that every computation that can arise from an execution of M satisfies D and every computation that satisfies D could have been the result of an execution on M.

In this thesis, two informal, ambiguous, memory consistency models are considered: pipelined RAM and processor consistency. The memory consistency condition pipelined RAM is discussed and used in many papers [LS88, ABJ+93, Mos93, JS96], but, upon close examination, not all are discussing the same machine or memory model. Lipton and Sandberg [LS88] originally described a machine that implements pipelined RAM. But three different assumptions can be made about this machine, giving rise to three different machine descriptions. Ahamad et al. [ABJ+93] assumed

one set of conditions and gave a formal definition of the resulting memory model. The machine is formalized in this thesis and Ahamad et al.'s formal definition is translated to the framework. Mosberger [Mos93] assumed another set of conditions, and the resulting machine and memory model are formalized and proven equivalent. A third possible interpretation of the pipelined-RAM machine is also described, and the precise machine description and the formal definition of the memory model it implements are supplied.

Processor consistency was originally defined by Goodman [Goo89]. Many have used the term processor consistency to describe differing memory consistency models [ABJ+93, GLL+90, KNA93, Mos93, GGH93]. Ahamad et al. [ABJ+93] gave a formal interpretation of Goodman's ambiguous definition. They also tried to formalize the memory model implemented by Stanford's DASH machine [GLL+90, GGH93], which is also named processor consistency. I show in this thesis that this formal definition does not capture processor consistency as implemented in the DASH machine. Finally, Goodman stated that the VAX 8800 is a machine that implements processor consistency. I give a formal definition of the memory model implemented by the VAX 8800 and show that it is yet another definition of processor consistency.

Chapter 2 outlines previous work related to memory consistency models; chapter 3 presents the formal framework used throughout this thesis; chapter 4 describes the memory consistency models sequential consistency and coherence; chapter 5 presents all the possible interpretations of the pipelined RAM machine; chapter 6 gives some of the possible interpretations of the memory model processor consistency; and chapter 7 summarizes this work and points toward future work.

Chapter 2

Previous Work on Memory Consistency Models

Lamport gave a precise definition of sequential consistency [Lam79] and launched investigations into relaxations of sequential consistency [Lam78, Lam79, Lam86], which was previously generally assumed. Dubois, Scheurich and Briggs were the first to actually propose a consistency model weaker than sequential consistency, namely *weak ordering* [DSB86]. Their work differentiated between ordinary and synchronization operations. Synchronization operations are guaranteed to be sequentially consistent with respect to each other; however, ordinary operations may be re-ordered in a way that best suits performance. Later, the Stanford team built on weak ordering when developing the DASH multiprocessor with the memory model *release consistency* [GLL+90, GGH93]. Lipton and Sandberg defined and implemented a machine with quite different

memory behavior, which they defined to be the *pipelined RAM* memory model [LS88]. Goodman, in 1989, was the first to explicitly state the notion of *coherence* or *cache consistency* [Goo89], which is generally accepted as a minimum memory consistency requirement. He was also the first to define processor consistency. Currently, there are several distinct variants that use the same name [ABJ+93, GLL+90, KNA93, Mos93, GGH93]. Herlihy and Wing defined the strongest memory consistency model, *linearizability*, which is often assumed by researchers in design and analysis of distributed algorithms [HW90].

A different *programmer-oriented approach* to the problem of non-sequentially-

consistent memory was investigated independently at Stanford [GLL+90, GGH93] and Wisconsin-Madison [Adv96, AH90a, AH93], and lead to the notions of Properly Labeled (PL) and Data Race Free (DRF) programs, respectively. PL is a constrained style of programming for a specific architecture. DRF develops a programming style and then suggests how to tailor the hardware to support it. These two teams later collaborated to combine their approaches [GAG+92].

Attiya, Chaudhuri, and Friedman have taken a complementary approach to DRF, which first specifies the memory consistency model and then develops a programming style [ACFW93].

These memory consistency models arise from a wide variety of sources including architecture, system, and database designers, application programmers, and theoreticians. The descriptions of memory behavior use different types and degrees of formalism. Definitions range from precise and complicated axiomatic specifications to informal and sometimes ambiguous natural language descriptions. This makes the many different memory consistencies difficult to reason about or to compare. Programming for these models becomes inefficient when a new descriptive style must first be mastered for each change of model. A single unified formalization is needed that can specify any memory model addressed in the literature or provided by existing and future machines.

A few research groups have proposed such unifying frameworks to describe the operation of distributed shared memories. Gibbons and Merrit proposed an automata-based framework [GMG91]. They start with a specific automaton to represent basic architectural assumptions. Then they define a memory consistency model as an automaton that is obtained by restricting the actions of the base automa-

ton. They used their formalism to model release consistency and to prove that, as long as a program is properly labeled, release consistency is equivalent to sequential consistency [GM92]. At Xerox's Palo Alto Research Center, Sindhu, Frailong, and Cekleov [SFC91] proposed an axiomatic framework that is based on three sets: memory operations, partial orders defined on memory operations, and axioms which are concerned with the legality of orders. A team at Georgia Institute of Technology (Georgie Tech) developed another framework, which is based on partial orders [KNA93, ABJ+93, ANB+95]. The framework used by this thesis [HKV98, HKV97] is based on the work at Georgia Tech.

Chapter 3

The Framework

The framework has been developed at the University of Calgary [HKV98, HKV97] and models a multiprocessor system, abstractly, as a collection of processes operating on a collection of shared data objects, as is shown in figure 3.1. In all multiprocessor systems considered in this thesis, processes initiate change (represented by the sequence of invocations in figure 3.1) and objects respond to these initiations (represented by the sequence of responses in figure 3.1).

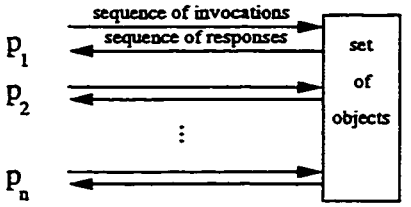


Figure 3.1: *A multiprocessor system*

A data object is defined by the set of all sequences of allowable operations together with their results (similar to the style of Herlihy and Wing [HW90]) as follows. An *action* is a 4-tuple (op, obj, in, out) where "op" is an operation, "obj" is an object name, and "in" and "out" are sequences of parameters. The action (op, obj, in, out) means that the operation "op" with input parameters "in" is applied to the object "obj" yielding the output parameters "out". A *(sequential data) object* is specified by a set of sequences of actions. For example, a shared atomic read-write register x is specified by the set of all sequences $(o_1, o_2, ...)$ such that

1. each o_i is either a read action, denoted by a four-tuple $(read, x, \lambda, v)$[1], that returns the value v of register x, or a write action, denoted $(write, x, v, \lambda)$, that assigns a value v to register x, and

2. for every read action, the value returned is the same as the value written by the most recent preceding write action in the sequence.

A sequence of actions is *valid* for object x if and only if it is in the specification of x.

An action o = (op, obj, in, out) can be decomposed into the two *matching* components, (op, obj, in), called the *action-invocation* and denoted invoc(o), and (op, obj, out), called the matching *action-response* and denoted resp(o). Let $(e_1, e_2, ...)$ be a sequence consisting of action-invocations and action-responses. Then e_j *follows* e_i if and only if $i < j$ and e_j *immediately follows* e_i if and only if $i = j - 1$.

Informally, a process interacts with data objects by issuing a stream of invocations to some subset of them and receiving a stream of responses that are interleaved with its invocations. This is formalized as follows. A *process* is a sequence of action-invocations. A *process execution* is a (possibly infinite) sequence of action-invocations and action-responses such that each response follows its matching invocation. An action is blocking if it has non-empty output and the response of the action immediately follows its matching invocation in the process execution. A process execution is *blocking* if, each action that has a non-empty output is blocking.

Whether blocking or non-blocking, the natural notion of the computation of a process is the sequence of actions that arises as its invocations are processed. Therefore, a *process computation* is the sequence of actions created from a process

[1] λ denotes the empty sequence.

execution by augmenting each invocation in the process sequence with the output of its matching response.

A *(multiprocess) system*, (P, J), is a collection P of processes and a collection J of objects, such that each action-invocation of each process p in P is applied to an object in J. A *(multiprocess) system execution* for a system (P, J), is a collection of process executions, one for each p in P. Similarly, a *system computation* is a collection of process computations, one for each p in P.

Throughout this thesis, it is assumed that each action in a computation is unique and thus the actions of a computation form a set. If some computation contains two or more actions that are identical, the actions could simply be augmented with the process that invoked the action and the index of the action-invocation in the process's program to ensure that each action is unique. In all examples in this thesis, only read and write actions are used. For legibility, $w(x)v$ denotes the action $(write, x, v, \lambda)$ and $r(x)v$ denotes the action $(read, x, \lambda, v)$. The notation $w_p(x)v$ or $r_p(x)v$ is used to emphasize that these actions are performed by process p.

Computation 1 $\begin{cases} p : w(x)1\ r(x)1 \\ q : w(x)0\ r(x)1 \end{cases}$

To illustrate, consider computation 1. This is a computation of a system containing two processes, p and q, and one object, x. Process p is defined by the sequence of invocations: $(write, x, 1)$ $(read, x, \lambda)$. Process q is defined by the sequence of invocations: $(write, x, 0)$ $(read, x, \lambda)$. Note that p's process computation is a sequence of actions that is valid for x, since the read action returns the value of the last preceding write, and that q's process computation is not valid for x. The two sequences could be merged together such that the resulting sequence is valid for x: $w_q(x)0\ w_p(x)1$

$r_q(x)1$ $r_q(x)1$.

Let (P, J) be a multiprocess system, and O be all the actions in a computation of this system. $O|p$ denotes all the actions that are in the process computation of p in P. $O|x$ are all the actions that are applied to object x in J.

Several partial orders and relations[2] on the actions of a system are used throughout this thesis to define the various memory consistency models. The following partial order is the only one defined in this chapter as it is used throughout the thesis. Others are defined as needed.

Action o_1 *program-precedes* o_2, denoted $o_1 \xrightarrow{prog} o_2$, if and only if invoc($o_2$) follows invoc($o_1$) in the definition of p (equivalently, o_2 follows o_1 in the computation of p). The partial order $(O, \xrightarrow{prog})$ is called the *program order*. Observe that for each process p in P, the program order is the process computation of p, a total order[3] on $O|p$.

For the definition of some memory consistency models it is necessary to distinguish the actions that change (write) a shared object from those that only inspect (read) a shared object. Let O_w denote that subset of O consisting of those actions in O that update a shared object, and O_r denote that subset consisting of the actions that only inspect a shared object.

Given any collection of actions O on a set of objects J, a *linearization of* O is a total order $(O, <_L)$ such that for each object x in J, the subsequence $(O|x, <_L)$ of $(O, <_L)$ is valid for x. Thus, the sequence $w_q(x)0$ $w_p(x)1$ $r_q(x)1$ $r_p(x)1$, containing the actions of computation 1, is a linearization.

---

[2] A partial order is an antisymmetric, transitive relation on a set. We denote a relation by a pair (S, R) where S is a set and $R \subset S \times S$. The notation $s_1 R s_2$ means $(s_1, s_2) \in R$. When the set S is understood, R denotes the relation. If $S' \subset S$ then (S', R) denotes the relation $(S', R \cap (S' \times S'))$.

[3] A total order is a partial order (S, R) such that $\forall x, y \in S$ $x \neq y$, either xRy or yRx and $\forall x$ $(x, x) \notin R$.

A *memory (consistency) model* is a set of constraints on system computations. A computation C satisfies some consistency condition D if the computation meets all the conditions of D. A multiprocessor system provides memory model D if every computation that can arise from the system satisfies the memory model D.

The preceding definitions allow us to formalize memory consistency models in terms of constraints on computations. An additional goal is to associate common memory consistency models with machine architectures. A *(multiprocessor) machine* is a collection of processors together with various memory components. A machine *implements an action* by proceeding through a sequence of *events* that depend on the particular machine and that occur at the various components of the machine such that the first event is the initiation of the action by one of the processors. A processor of a machine *implements a process* by initiating the implementation of the actions corresponding to the action-invocations of the process in program order. A multiprocessor machine *implements a system* (P, J) by having each process in P implemented by some processor in the machine. A *machine execution* is described by the sequence of resulting machine events.[4] For any machine execution E, let $a \xrightarrow{E} b$ iff a precedes b in E.

Finally, the following conventions are used throughout this thesis. Two events are *matching* when both are part of the implementation of a single action. An event and an action are *corresponding* when the event is part of the implementation of the action. An execution on a machine satisfies the *constraints* of the machine. A

[4]Events in a multiprocessor can be simultaneous. For example, two different caches may be simultaneously updated. However, because the same outcome would arise if these simultaneous events were ordered one after the other in arbitrary order, we can assume that the outcome of a machine computation arises from a *sequence* of events.

computation satisfies the *conditions* of a memory consistency model. A machine M *implements exactly* a memory consistency model D, if every computation that arises from some execution on M satisfied D, and every computation that satisfies D is the result of some execution on M.

Chapter 4

Basic Memory Consistency Models

The memory consistency model sequential consistency, is the model most programmers prefer to reason with because it most closely resembles a sequential machine. Coherence is typically assumed to be a necessary memory model requirement of any reasonable machine. These two memory models are described here since they are the building blocks of the other memory consistency models discussed in this thesis. Each is illustrated using a machine that implements the model. These results also appear in a technical report [HKV98].

4.1 Sequential Consistency

Sequential consistency (henceforth abbreviated SC), defined by Lamport [Lam79], is the most widely used memory consistency model. According to Lamport, a multiprocessor is said to be SC if:

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

Several other papers [ABJ+93, HW90, Goo89, Mos93] describe SC; some use a different name or a different, but equivalent, definition.

Definition 4.1.1 *Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is* sequentially consistent *if there is a linearization $(O, <_L)$ such that $(O, \xrightarrow{prog}) \subseteq (O, <_L)$.*

Dubois, Scheurich and Briggs define *strong ordering* as a sufficient condition for SC [DSB86] and Goodman states that "A system that adheres to this level of consistency is said to be a strongly ordered system" [Goo89]. However, Adve and Hill show that strong ordering and SC are similar, but are not equivalent [AH90b].

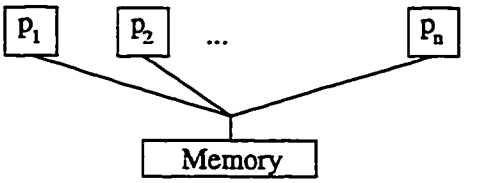


Figure 4.1: *M_{SC}, a machine that implements SC*

Figure 4.1 depicts a simple machine, M_{SC}, where each processor is connected through bi-directional First-In-First-Out (bi-FIFO) channels to a switch, which is connected to memory via a single bi-FIFO channel. M_{SC} implements a read action (read, x, λ, v) by processor p, with the ordered events

1. processor p sends a processor-read-request(x) followed by (not necessarily immediately)

2. a memory-read-reply(x, v) to p.

A write action (write, x, v, λ) is implemented with the ordered events

1. processor p sends a processor-write-request(x, v), followed by (not necessarily immediately)

2. a memory-update(x, v).

For any computation C arising from an execution of M_{SC}, construct a sequence S of the actions of C by placing the actions in the order in which the corresponding *memory* events occurred. Sequence S must be in program order since the memory only services one action at a time, and the channels are FIFO. Moreover, it must be valid since each memory-read-reply will return the value of the last memory-update of the same cell. Thus S is a linearization that satisfies Definition 4.1.1. A simple argument in reverse can be used to show that any SC computation could have been executed on M_{SC}, thus establishing the following claim.

Claim 4.1.2 *M_{SC} implements exactly sequential consistency.*

4.2 Coherence

Coherence, also called cache consistency [Goo89], is among the weakest consistency conditions. Goodman states that coherence "only guarantees that accesses to a given memory location are strongly ordered" [Goo89]. Mosberger indicates that "Coherence only requires that accesses are SC on a *per-location* basis" [Mos93].

Definition 4.2.1 *Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is* coherent *if for each object $x \in J$ there is some linearization $(O|x, <_{L_x})$ satisfying $(O|x, \xrightarrow{prog}) \subseteq (O|x, <_{L_x})$.*

Computation 2 is coherent but not SC. The linearizations for objects x and y are $<_{L_x} = w_p(x)0\ r_q(x)0\ w_p(x)1$ and $<_{L_y} = w_q(y)0\ r_p(y)0\ w_q(y)1$. However, there is no single linearization of all these actions that maintains program order.

Computation 2 $\begin{cases} p: w(x)0\ w(x)1\ r(y)0 \\ q: w(y)0\ w(y)1\ r(x)0 \end{cases}$

Often coherence is described as a system where all processes view all the write actions to the same object in the same order [AH90a, GLL+90, KNA93], or "all writes to the same location are serialized in some order and are performed in that order with respect to any processor" [GLL+90]. Furthermore, it is implicit in these informal descriptions that program order is maintained on a per object basis.

Definition 4.2.2 *Let O be all the actions of a computation C of some system (P, J). Then C is* coherent-var1 *if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying*

1. $(O|p, \xrightarrow{prog}) = (O|p, <_{L_p})$, *and*
2. $\forall q \in P$ *and* $\forall x \in J$ $(O_w|x, <_{L_p}) = (O_w|x, <_{L_q})$.

Claim 4.2.3 *A computation is coherent if and only if it is coherent-var1.*

Proof: Consider a coherent computation of a system (P, J). For each $p \in P$, construct a sequence of actions ρ_p from the total order $(O|p, \xrightarrow{prog})$ and from all the linearizations $(O|x, <_{L_x})$ that satisfy Definition 4.2.1. Initially, let $\rho_p = (O|p, \xrightarrow{prog})$. Consider each $x \in J$ in turn. All actions, say $\{o_1, o_2, ..., o_k\}$, in $(O|p)|x$ appear in $(O|x, <_{L_x})$ in program order. Hence, $(O|x, <_{L_x}) = S_0^x, o_1, S_1^x, o_2, ..., o_k, S_k^x$ for some subsequences $S_0^x, S_1^x, ..., S_k^x$ of $(O|x, <_{L_x})$. Insert each sub-segment $S_{i_w}^x$, consisting of the sequence of writes in S_i^x, into ρ_p anywhere between o_i and o_{i+1} (maintaining the order of $S_{i_w}^x$). The resulting sequence, ρ_p, clearly contains all actions in $(O|p \cup O_w)$ and satisfies program order for p. Also, the subsequence of ρ_p consisting of writes

to x is exactly the sequence $(O_w|x, <_{L_x})$ so it is the same for each process. Finally, the subsequence of ρ_p consisting of actions to x is a subsequence of $(O|x, <_{L_x})$ with only some reads removed, so ρ_p is a linearization. Thus for each $p \in P$, ρ_p satisfies the requirements of the definition of coherent-var1.

Now consider a coherent-var1 computation of a system (P, J), and any $x \in J$. For each $p \in P$, let $(O|p \cup O_w, <_{L_p})$ be the linearization satisfying Definition 4.2.2. By part 2 of Definition 4.2.2, all processors have the same ordering of writes to x and by part 1 this ordering satisfies program order. Let ρ_x be this sequence of writes. For each process p, the sequence of writes in $((O|p \cup O_w))|x, <_{L_p})$ is exactly ρ_x. Let r be any read of x by some process q. Suppose, in the linearization $((O|q \cup O_w)|x, <_{L_q})$, the closest write preceding r is $w(x)_i$, and the closest write succeeding r is $w(x)_{i+1}$. Since $w(x)_i$ and $w(x)_{i+1}$ are adjacent writes in ρ_x, r can be inserted between them while preserving validity and program order. In this way, every $r \in O_r|x$ can be inserted into ρ_x yielding a linearization of $O|x$ that preserves program order, and satisfies the requirements of Definition 4.2.1. ■

There is also a third possible definition of coherence, which arises because there are no restrictions on the ordering of actions to different objects.

Definition 4.2.4 *Let O be all the actions of a computation E of some system (P, J). Then E is* coherent-var2 *if there is some linearization $(O, <_L)$ such that $\forall x \in J$ $(O|x, \xrightarrow{prog}) \subseteq (O|x, <_L)$.*

Claim 4.2.5 *A computation is coherent if and only if it is coherent-var2.*

Proof: For any coherent computation, create one sequence by placing the object linearizations, which satisfy Definition 4.2.1, one after the other. The result

is a linearization that satisfies Definition 4.2.4 since it orders all the actions in the execution, such that program order on a per object basis is maintained. For any computation that satisfies Definition 4.2.4, and for each object x, set $(O|x, <_{L_x})$ equal to $(O|x, <_L)$. Then $(O|x, <_{L_x})$ is valid and maintains program order so it satisfies the requirements of Definition 4.2.1. ■

Since coherence, coherence-var1 and coherence-var2 are all equivalent, henceforth only the term coherence is used.

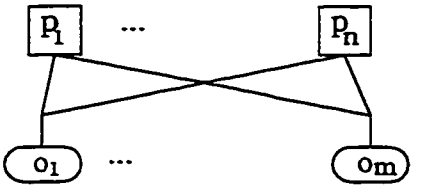


Figure 4.2: *M_C, a machine that implements coherence*

In the machine, M_C, in Figure 4.2, there is one switch for each object, and each processor is connected by a bi-FIFO channel to each switch. M_C implements a read action (read, x, λ, v) by process p, with the ordered events 1) a processor-read-request is placed on the channel connecting p to x, and 2) an object$_x$-read-reply(v) is placed on the channel from x to p. A write action (write, x, v, λ) by p is implemented with the ordered events 1) a processor-write-request(v) is placed on the channel connecting p to x, and 2) object$_x$-update(v) is performed by object x. In any execution E of M_C, each object$_x$-read-reply event will contain the value of the last preceding object$_x$-update event that precedes it in E.

Claim 4.2.6 *The machine M_C, with non-blocking reads, implements exactly coherence.*

Proof: For any computation C arising from an execution of M_C, and for each object x, construct a sequence S_x of the actions of C on x by placing these actions in the order in which the corresponding *object* events occurred. Since each object services only one request at a time, and the channels are FIFO, S_x must satisfy program order. Also, S_x must be valid since each object$_x$-read-reply will return the value of the last object$_x$-update. Thus for each x, S_x is a linearization that satisfies Definition 4.2.1.

Now, for any computation C that satisfies Definition 4.2.4, construct a sequence S of events that reflects how C could have arisen from an execution of M_C as follows. First, for each process p, implemented by processor $\hat{p}$, construct a sequence S_p of processor-request events that corresponds to p. That is, the i^{th} event of S_p is a processor-read-request (respectively, processor-write-request) placed on the channel from processor $\hat{p}$ to x, if and only if the i^{th} action-invocation in process p is a read of (respectively, write to) object x. Also, from the linearization L that satisfies Definition 4.2.4 construct a sequence Q consisting of object-read-reply events and object-update events by setting the i^{th} event in Q to be the object event that corresponds to the i^{th} action in L. Now form S by concatenating the S_p's for each p (in any order) followed by Q. According to S, each processor issues requests in program order and replies are executed in program order per object. Although the order in which processors place requests to some object x (represented by all the subsequences S_p) may not agree with the order of the corresponding replies (represented by Q), the asynchronous FIFO channels permit an interleaving of all the requests to x from different processors into an order that agrees with the order of replies. And, since L is a linearization, each object$_x$-read-reply will return the value of the last preceding

object$_x$-update. Therefore, the sequence of events, S, could have occurred on M_C. ■

Chapter 5

Pipelined RAM

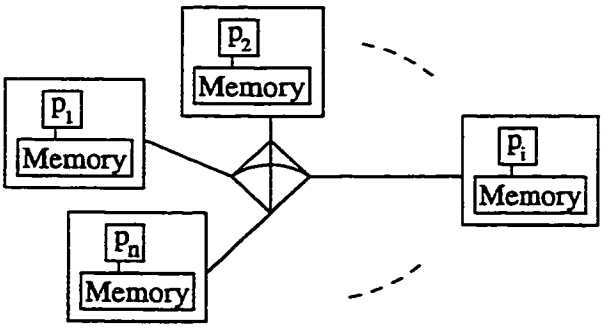


Figure 5.1: *The pipelined RAM machine*

Lipton and Sandberg [LS88] described the Pipelined Random Access Machine (pipelined RAM) with an architecture as shown in Figure 5.1. Each processor p has its own copy, μ_p, of the shared memory, and a FIFO channel connects every processor to every other processor's copy of memory. According to Lipton and Sandberg, read and write actions by process p are implemented in this machine as follows:

- Processor p implements a read(i) "by performing a normal read from location i" of μ_p.

- Processor p implements a write(i, v) "by performing a *local* action and initializing a *global* action. Locally, it does a normal write to $[\mu_p]$ at location i with value v. Globally, it sends a message $< i, v >$ to all the other processors."

They emphasize that, upon a write, a processor does not wait for the update to take effect in the copies of memory at other processors. What is not completely clear

from this description, however, is whether or not reads and/or "local" writes are blocking; however, the use of "performing" (as opposed to "initializing") seems to indicate a blocking activity. It is also unclear whether a processor first completes the update of its own copy of memory and then initiates the updates of other copies, or whether these events can happen in arbitrary order. Different memory models arise depending upon what assumptions are made concerning both of these issues. Lipton and Sandberg implicitly assume that each copy of the memory receives and implements all actions in program order. I have maintained this assumption.

Regardless of the assumption made, on a pipelined RAM machine a read action $(read, x, \lambda, v)$ is implemented by the following ordered sequence of events: a) a processor$_p$-read-request(x) by p to μ_p and b) from μ_p a matching μ_p-reply(x, v) to p, and a write action $(write, x, \lambda, v)$ is implemented by the ordered sequence of events: a) a processor$_p$-write-request(x, v) by p to all copies of the memory and b) for each processor q, a matching μ_q-update$_p$(x, v). All variants of the pipelined RAM machine ensure that for any execution E, and for any processors p and q,

1. processor$_p$ events are in the same order as the matching μ_q events (if they exist), and

2. for any location x, any values u, v and for each μ_p-reply(x, v) event e_r, if μ_p-update(x, u) is the last μ_p-update event to x preceding e_r in E, then $v = u$.

Call this machine M_{PRAMA}.

Let M_{PRAMR} be the machine M_{PRAMA} with the additional constraint that read actions are blocking. Thus, any execution E on M_{PRAMR} meets conditions 1 and 2 of M_{PRAMA} and

3 there are no processor$_p$ events between a processor$_p$-read-request and the matching μ_p-reply event.

Finally, let M_{PRAMW} be M_{PRAMA} with the addition constraint that, during the implementation of a write action, the writer's copy of the memory is updated before any other processor's copy of the memory. Thus, any execution E on M_{PRAMW} satisfies the two constraints of M_{PRAMA} and

4 a μ_p-update$_p$ event is ordered before any matching μ_q-update$_p$ event for any processor $q \neq p$.

The following three sections give formal definition for each of these three machines and the final section in this chapter compares these with the previously discussed memory consistency models.

5.1 Pipelined RAM as interpreted by Ahamad et al.

Ahamad et al. [ABJ+93] at Georgia Tech formalized one version of the Pipelined RAM machine, which, in the framework, becomes the following widely quoted definition:

Definition 5.1.1 *Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is* P-RAM-A *if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$.*

Claim 5.1.2 *The machine M_{PRAMA} implements exactly P-RAM-A.*

Proof: Consider any execution E on M_{PRAMA} with resulting computation C containing actions O. Let $P = \{\bar{p} \mid \text{processor } p \text{ is in } M_{PRAMA}\}$ be a set of processes such that the i^{th} action-invocation in each process $\bar{p}$ corresponds to the i^{th} processor$_p$ event in E. The set of objects J is the set of all memory locations in M_{PRAMA}. For all processors p in M_{PRAMA} construct a subsequence for each processor p by including only the "view of p's memory". That is, for each processor p, let E_p be the subsequence of E containing exactly all μ_p events. Define, for each process $\bar{p}$, a sequence $<_{L_p}$ of reads and writes as follows. The i^{th} action in $<_{L_p}$ is $r_{\bar{p}}(x)v$ (respectively, $w_{\bar{q}}(x)v$) if and only if the i^{th} event in E_p is μ_p-reply(x, v) (respectively, μ_p-update$_q(x, v)$). Then $<_{L_p}$ clearly satisfies program order by constraint 1 of M_{PRAMA}, consists of exactly $O|\bar{p} \cup O_w$ by construction, and is valid by constraint 2 of M_{PRAMA}. And thus, for each process $\bar{p}$, $(O|\bar{p} \cup O_w, <_{L_p})$ is a linearization that satisfies P-RAM-A.

Now consider any computation, C, of system (P, J) containing actions O, that satisfies definition 5.1.1. Let the processor that implements process $p \in P$ be named $\hat{p}$ and the objects in J are the locations in M_{PRAMA}. Choose any set of linearizations, one for each process $p \in P$, that satisfy P-RAM-A and construct a corresponding sequence of events E for M_{PRAMA} from an n-way merge of these n linearizations $(O|p \cup O_w, <_{L_p}) = o_1^p, o_2^p, \ldots, o_k^p$ as follows. Initially, $E = \lambda$ and, for each processor $\hat{p}$, implementing process $p \in P$, $L_p = (O|p \cup O_w, <_{L_p})$ and $P_p = (O|p, \xrightarrow{prog})$. Consider the first remaining action o_i^p of each sequence L_p in turn (the first time this will be action o_1^p for each process p). If the processor event corresponding to o_i^p is not in E yet and $o_i^p \in O|q$ for some process q, append to E, in order, all events corresponding to actions in P_q up to and including o_i^p and remove these actions from P_q. Now append the $\mu_{\bar{p}}$ event corresponding to o_i^p and remove o_i^p from L_p.

Clearly, processor$_{\tilde{p}}$ events are in program order and precede the matching $\mu_{\tilde{q}}$ events. Hence the ordering of events corresponding to an action is satisfied. All $\mu_{\tilde{p}}$ events are in the same order as the corresponding actions are ordered in $(O|p \cup O_w, <_{L_p})$, which satisfies program order, and all processor$_{\tilde{p}}$ events are also in program order in E by construction of E. Thus, condition 1 of M_{PRAMA} is satisfied. Since $(O|p \cup O_w, <_{L_p})$ is a linearization, constraint 2 of M_{PRAMA} is also satisfied. Hence, E could have occurred on M_{PRAMA}. ■

5.2 Pipelined RAM as interpreted by Mosberger

In contrast to Ahamad et al., Mosberger assumed that upon a write-request, a processor first updates its own memory and subsequently broadcasts this update to all other processors or " ... on a read, a [Pipelined RAM] would simply return the value stored at the local copy of the memory. On a write, it would update the local copy first and broadcast the new value to the other process." [Mos93].

Thus, if p observed a write, w_q, by some other process q before its own write w_p, then q must also observe w_q before w_p. That is, in the linearizations $(O|p \cup O_w, <_{L_p})$ that capture processes' views of the computation, $w_q <_{L_p} w_p \Rightarrow w_q <_{L_q} w_p$. Furthermore, if p observed that w_q occurred before w_p and some other process r observed w_p before its own write w_r, then q must also observe that w_q occurred before w_r. That is $w_q <_{L_p} w_p <_{L_r} w_r \Rightarrow w_q <_{L_q} w_r$. In general, the antecedent of this implication can be any finite length.

Definition 5.2.1 *Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is* P-RAM-W *if for each process $p \in P$ there is a linearization*

$(O|p \cup O_w, <_{L_p})$ *satisfying*

1. $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$, *and*
2. *for any* $m \geq 1$ *and for all* $w_{p_0}, w_{p_1}, \ldots, w_{p_m} \in O_w$, *where for any* i, $w_{p_i} \in O_w|p_i$ *and* $p_i \in P$, *if* $w_{p_0} <_{L_{p_1}} w_{p_1} <_{L_{p_2}} w_{p_2} <_{L_{p_3}} \ldots <_{L_{p_m}} w_{p_m}$ *then* $w_{p_0} <_{L_{p_0}} w_{p_m}$.

Claim 5.2.2 M_{PRAMW} *implements exactly P-RAM-W.*

Proof: Let E be any M_{PRAMW} execution, thus satisfying constraints 1, 2 and 4, and let C be the computation resulting from E with set of actions O. Let $P = \{\bar{p} \mid$ processor p is in M_{PRAMW} $\}$ be a set of processes such that the i^{th} action-invocation in each $\bar{p}$ corresponds to the i^{th} processor$_p$ event in E. The set of objects J is the set of all locations in M_{PRAMW}. From E, construct a subsequence for each processor p by including only the "view of p's memory". That is, for each processor p, let E_p be the subsequence of E containing exactly all μ_p events. Define, for each process $\bar{p}$ a sequence $<_{L_p}$ of reads and writes as follows. The i^{th} action in $<_{L_p}$ is $r_{\bar{p}}(x)v$ (respectively, $w_{\bar{q}}(x)v$) if and only if the i^{th} event in E_p is μ_p-reply(x, v) (respectively, μ_p-update$_q(x, v)$). Then $<_{L_p}$ satisfies program order by constraint 1 and consists of exactly $O|\bar{p} \cup O_w$ by construction and is valid by constraint 2.

To show that the collection of sequences $<_{L_p}$ also satisfy condition 2, assume that there exist $m \geq 1$, and $w_{\bar{p}_0}, w_{\bar{p}_1}, \ldots, w_{\bar{p}_m} \in O_w$, where $w_{\bar{p}_i}$ is a write by process $\bar{p}_i$ such that $w_{\bar{p}_0} <_{L_{\bar{p}_1}} w_{\bar{p}_1} <_{L_{\bar{p}_2}} w_{\bar{p}_2} <_{L_{\bar{p}_3}} \ldots <_{L_{\bar{p}_m}} w_{\bar{p}_m}$. Then in E, for all i, where $1 \leq i \leq m$, μ_{p_i}-update$_{p_{i-1}} \xrightarrow{E} \mu_{p_i}$-update$_{p_i}$. By constraint 4, μ_{p_i}-update$_{p_i} \xrightarrow{E} \mu_{p_{i+1}}$-update$_{p_i}$ and thus μ_{p_0}-update$_{p_0} \xrightarrow{E} \mu_{p_1}$-update$_{p_0} \xrightarrow{E} \mu_{p_1}$-update$_{p_1} \xrightarrow{E} \mu_{p_2}$-update$_{p_1}$ $\xrightarrow{E} \mu_{p_2}$-update$_{p_2} \xrightarrow{E} \ldots \xrightarrow{E} \mu_{p_m}$-update$_{p_{m-1}} \xrightarrow{E} \mu_{p_m}$-update$_{p_m} \xrightarrow{E} \mu_{p_0}$-update$_{p_m}$.

That is μ_{p_0}-update$_{p_0} \xrightarrow{E} \mu_{p_0}$-update$_{p_m}$. Hence, by construction, $w_{\bar{p}_0} <_{L_{\bar{p}_0}} w_{\bar{p}_m}$. And thus, for each process $\bar{p}$, $(O|\bar{p} \cup O_w, <_{L_p})$ is a linearization that satisfies P-RAM-W.

Now consider any computation, C, of system (P, J) containing set of actions O, that satisfies Definition 5.2.1. Let the processor that implements process $p \in P$ be named $\hat{p}$ and let the objects in J be the locations in M_{PRAMW}. Choose any set of linearizations, one for each process $p \in P$, that satisfy P-RAM-W and construct a corresponding sequence of events E for M_{PRAMW} from an n-way merge of these n linearizations $(O|p \cup O_w, <_{L_p}) = o_1^p, o_2^p, \ldots, o_k^p$ as follows. Initially, $E = \lambda$ and $L_p = (O|p \cup O_w, <_{L_p})$ for each processor $\hat{p}$ implementing process $p \in P$. Consider the first remaining action o_i^p of each sequence L_p in turn (the first time this will be action o_1^p for each process p). If $o_i^p \in O|p$ then append to E the corresponding processor$_{\hat{p}}$ and $\mu_{\hat{p}}$ events in that order and remove the action o_i^p from L_p. Otherwise, $o_i^p \in O_w|q$ for some process $q \neq p$. If the corresponding $\mu_{\hat{q}}$-update$_{\hat{q}}$ event is already in E, then append the $\mu_{\hat{p}}$-update$_{\hat{q}}$ event corresponding to o_i^p to E, and remove the action o_i^p from L_p, otherwise leave L_p and E unchanged. Now consider the first remaining action of the next sequence $L_{p'}$.

If this construction exhausts each L_p, then clearly, processor$_{\hat{p}}$ events are in program order and precede the matching $\mu_{\hat{q}}$ events. Hence the ordering of events corresponding to an action is satisfied. All $\mu_{\hat{p}}$ events are in the same order as the corresponding actions are ordered in $(O|p \cup O_w, <_{L_p})$, which are in program order, and processor$_{\hat{p}}$ events are in program order in E by construction of E. Thus, constraint 1 is satisfied by E. Since $(O|p \cup O_w, <_{L_p})$ is a linearization, constraint 2 is also satisfied. And by construction of E, the $\mu_{\hat{p}}$-update$_{\hat{p}}$ event precedes the matching $\mu_{\hat{q}}$-update$_{\hat{p}}$ for all processors $\hat{q} \neq \hat{p}$, satisfying constraint 4. Hence, E could have

occurred on M_{PRAMW}.

So it remains to verify that all events associated with the execution are necessarily added to E. Assume instead that at some point in the construction of E, each sequence L_p begins with a write $o_i^p \in O_w|q$ for some $q \neq p$ and the corresponding processor$_{\tilde{q}}$ event is not yet in E. Then there must be a cyclic sequence $p_{\alpha_1}, p_{\alpha_2}, \ldots, p_{\alpha_k}$ of processes such that $L_{p_{\alpha_i}}$ begins with a write by process $p_{\alpha_{i+1}}$ labeled $w_{\alpha_{i+1}}$ and L_{p_k} begins with a write by process p_{α_1}. Also, w_{α_i} must be in $L_{p_{\alpha_i}}$ since otherwise the construction could proceed. But this contradicts condition 2 of P-RAM-W. Hence the construction of E completes, and E describes an execution on M_{PRAMW} of C. ■

5.3 The pipelined RAM Machine with Blocking Reads

Finally, consider the pipelined RAM machine with blocking reads. In any computation resulting from some execution on M_{PRAMR}, let a read by some processor p, r_p, precede a write by p, w_p. If, furthermore, some other processor q observes that w_p occurred before its own read action r_q which in turn precedes q's write action w_q, then p must view that r_p preceded w_q, because the write by p could not have been initiated until the read by p was completed. Using a similar argument as used for P-RAM-W, the following formal definition results:

Definition 5.3.1 *Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is* P-RAM-R *if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying*

1. $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$, *and*

2. *for any* $m \geq 1$ *and for all* $w_{p_0}, w_{p_1}, \ldots, w_{p_m} \in O_w$ *and for all* $r_{p_0}, r_{p_1}, \ldots, r_{p_m} \in O_r$, *where for each* i, $w_{p_i} \in O_w|p_i$, $r_{p_i} \in O_r|p_i$ *and* $p_i \in P$, *if* $r_{p_0} <_{L_{p_0}} w_{p_0} <_{L_{p_1}} r_{p_1} <_{L_{p_1}} w_{p_1} <_{L_{p_2}} r_{p_2} <_{L_{p_2}} w_{p_2} <_{L_{p_3}} \cdots <_{L_{p_m}} r_{p_m} <_{L_{p_m}} w_{p_m}$ *then* $r_{p_0} <_{L_{p_0}} w_{p_m}$.

Claim 5.3.2 *M_{PRAMR} implements exactly P-RAM-R .*

Proof: Let E be a M_{PRAMR} execution, thus satisfying constraints 1, 2, and 3, with resulting computation C containing set of actions O. Let $P = \{\bar{p} \mid$ processor p is in $M_{PRAMR}\}$ be a set of processes such that the i^{th} action-invocation in each $\bar{p}$ corresponds to the i^{th} processor$_p$ event in E. The set of objects J is the set of all locations in M_{PRAMR}. From E, construct a subsequence for each processor p by including only the "view of p's memory". That is, for each processor p, let E_p be the subsequence of E containing exactly all μ_p events. Define, for each $\bar{p} \in P$, a sequence $<_{L_p}$ of reads and writes as follows. The i^{th} action in $<_{L_p}$ is $r_{\bar{p}}(x)v$ (respectively, $w_{\bar{q}}(x)v$) if and only if the i^{th} event in E_p is μ_p-reply(x, v) (respectively, μ_p-update$_q(x, v)$). Then $<_{L_p}$ clearly satisfies program order by constraint 1 and consists of exactly $O|\bar{p} \cup O_w$ (by construction) and is valid by constraint 2.

To show that the collection of sequences $<_{L_p}$ also satisfy condition 2 of P-RAM-R, assume that there exist $m \geq 1$, and $w_{\bar{p}_0}, w_{\bar{p}_1}, \ldots, w_{\bar{p}_m} \in O_w$, and $r_{\bar{p}_0}, r_{\bar{p}_1}, \ldots, r_{\bar{p}_m} \in O_r$, where $w_{\bar{p}_i}$ ($r_{\bar{p}_i}$ respectively) is a write (read, respectively) by process $\bar{p}_i$ such that $r_{\bar{p}_0} <_{L_{\bar{p}_0}} w_{\bar{p}_0} <_{L_{\bar{p}_1}} r_{\bar{p}_1} <_{L_{\bar{p}_1}} w_{\bar{p}_1} <_{L_{\bar{p}_2}} r_{\bar{p}_2} <_{L_{\bar{p}_2}} w_{\bar{p}_2} <_{L_{\bar{p}_3}} r_{\bar{p}_3} <_{L_{\bar{p}_3}} \cdots w_{\bar{p}_{m-1}} <_{L_{\bar{p}_m}} r_{\bar{p}_m} <_{L_{\bar{p}_m}} w_{\bar{p}_m}$. Then, by construction of each $<_{L_{\bar{p}_i}}$, μ_{p_0}-reply$\xrightarrow{E}\mu_{p_0}$-update$_{p_0}$ and for each i, where $1 \leq i \leq m$, μ_{p_i}-update$_{p_{i-1}} \xrightarrow{E} \mu_{p_i}$-reply $\xrightarrow{E} \mu_{p_i}$-update$_{p_i}$. Let e_r^p be any μ_p-reply event and e_w^p be any μ_p-update$_p$ event. Furthermore, let α_r^p

be the processor$_p$-read-request event matching e_r^p, let α_w^p be the processor$_p$-write-request event matching e_w^p and, for all processors q, let $e_w^{p,q}$ be the μ_q-update$_p$ event matching e_w^p. Then condition 1 of M_{PRAMA} implies that if $e_r^p \xrightarrow{E} e_w^p$ then $\alpha_r^p \xrightarrow{E} e_w^p$. Furthermore, by constraint 3, if $\alpha_r^p \xrightarrow{E} \alpha_w^p$ then $\alpha_r^p \xrightarrow{E} e_r^p \xrightarrow{E} \alpha_w^p$. And thus, $e_r^p \xrightarrow{E} e_w^{p,q}$ since $\alpha_w^p \xrightarrow{E} e_w^{p,q}$. Thus, for each i, where $0 \leq i \leq m$, μ_{p_i}-reply $\xrightarrow{E}$ $\mu_{p_{i+1}}$-update$_{p_i}$, where indices are reduced modulo $m+1$. That is, μ_{p_0}-reply $\xrightarrow{E}$ μ_{p_1}-update$_{p_0}$ $\xrightarrow{E}$ μ_{p_1}-reply $\xrightarrow{E}$ μ_{p_2}-update$_{p_1}$ $\xrightarrow{E}$ μ_{p_2}-reply $\xrightarrow{E}$ μ_{p_3}-update$_{p_2}$ $\xrightarrow{E}$ $\ldots$ μ_{p_m}-update$_{p_{m-1}}$ $\xrightarrow{E}$ μ_{p_m}-reply $\xrightarrow{E}$ μ_{p_0}-update$_{p_m}$. Thus, by construction of $<_{L_{\bar{p}_0}}$, $r_{\bar{p}_0} <_{L_{\bar{p}_0}} w_{\bar{p}_m}$, satisfying condition 2 of P-RAM-R and, for each process $\bar{p}$, $(O|\bar{p} \cup O_w, <_{L_p})$ is a linearization that satisfies P-RAM-R.

Now consider any computation, C, containing set of actions O, of system (P, J) that satisfies Definition 5.3.1. Let the processor that implements process $p \in P$ be named $\hat{p}$ and let the objects in J be the locations in M_{PRAMR}. Choose any set of n linearizations, one for each process $p \in P$, that satisfy P-RAM-R and construct a corresponding sequence of events E for M_{PRAMR} from an n-way merge of these n linearizations $(O|p \cup O_w, <_{L_p}) = o_1^p, o_2^p, \ldots, o_k^p$ as follows. Initially, $E = \lambda$ and for each processor $\hat{p}$, implementing process $p \in P$, $L_p = (O|p \cup O_w, <_{L_p})$ and $P_p = (O|p, \xrightarrow{prog})$. Consider the first remaining action o_i^p of each sequence L_p in turn (the first time this will be action o_1^p for each process p). If $o_i^p \in O|p$, append to E the corresponding processor$_{\hat{p}}$ event unless it is in E already and then append the $\mu_{\hat{p}}$ event corresponding to o_i^p. Remove the action o_i^p from L_p and P_p (if it still is in P_p). Otherwise $o_i^p \in O_w|q$ for some process $q \neq p$. There are 3 possible cases: 1) if o_i^p is the first action in P_q, then append the processor$_{\hat{q}}$-write-request and $\mu_{\hat{p}}$-update$_{\hat{q}}$ events corresponding to o_i^p to E in that order and remove o_i^p from L_p and P_q; 2) if the $\mu_{\hat{q}}$-update$_{\hat{q}}$ event

corresponding to o_i^p is already in E, then append the matching $\mu_{\tilde{p}}$-update$_{\tilde{q}}$ to E, and remove the action o_i^p from L_p; 3) otherwise leave L_p and E unchanged. Now consider the first remaining action of the next sequence $L_{p'}$. It is straightforward to check that the sequence E so constructed satisfies the conditions of M_{PRAMR}, provided that this construction exhausts each L_p.

So it remains to verify that all events associated with the execution are necessarily added to E. Assume instead that at some point in the construction of E, each sequence L_p begins with a write $o_i^p \in O_w|q$ for some $q \neq p$ and the corresponding processor$_{\tilde{q}}$ event is not yet in E. Furthermore, o_i^p is preceded in P_q by some read action r_q otherwise construction could proceed. (It is not possible that o_i^p is preceded by a write action in P_q, since this write action must also precede o_i^p in L_p and would already be removed from P_q.) Then there must be a cyclic sequence $p_{\alpha_1}, p_{\alpha_2}, \ldots p_{\alpha_k}$ of processes such that $L_{p_{\alpha_i}}$ begins with a write by process $p_{\alpha_{i+1}}$ labeled $w_{\alpha_{i+1}}$ and $L_{p_{\alpha_k}}$ begins with a write by process p_{α_1}. Also, w_{α_i} must be in $L_{p_{\alpha_i}}$, and there must be a read action r_{α_i} by p_{α_i} between $w_{\alpha_{i+1}}$ and w_{α_i} in $L_{p_{\alpha_i}}$ since otherwise the construction could proceed. Thus, $w_{\alpha_k} <_{L_{\alpha p_1}} r_{\alpha_1} <_{L_{\alpha p_1}} w_{\alpha_1} <_{L_{\alpha p_2}} r_{\alpha_2} <_{L_{\alpha p_2}} w_{\alpha_2} <_{L_{\alpha p_2}} \ldots r_{\alpha_k}$ $<_{L_{\alpha p_k}} w_{\alpha_k}$. But this contradicts condition 2 of P-RAM-R. Hence the construction of E completes, and E describes an execution on M_{PRAMR} of C. ■

5.4 Memory Consistency Models Compared

If a computation is P-RAM-R or P-RAM-W, then it clearly is also P-RAM-A. Furthermore, any P-RAM-W computation is P-RAM-R. Consider any P-RAM-W computation C of some system (P, J) containing set of actions O. Let $\{(O|p \cup O_w, <_{L_p}) \mid$

$p \in P\}$ be any set of linearizations, containing exactly one linearization for each process p, that satisfy the conditions of P-RAM-W. These linearizations will also satisfy condition 1 of P-RAM-R trivially. To show that these linearizations satisfy condition 2 of P-RAM-R, let for any $m \geq 1$ and any $w_{p_0}, w_{p_1}, \ldots, w_{p_m} \in O_w$ and any $r_{p_0}, r_{p_1}, \ldots, r_{p_m} \in O_r$, where each w_{p_i} (respectively, r_{p_i}) is a write (respectively, read) by process p_i, and $r_{p_0} <_{L_{p_0}} w_{p_0} <_{L_{p_1}} r_{p_1} <_{L_{p_1}} w_{p_1} <_{L_{p_2}} r_{p_2} <_{L_{p_2}} w_{p_2} <_{L_{p_3}} \ldots <_{L_{p_m}} r_{p_m} <_{L_{p_m}} w_{p_m}$. This implies that $w_{p_o} <_{L_{p_1}} w_{p_1} <_{L_{p_2}} w_{p_2} <_{L_{p_3}} \ldots <_{L_{p_m}} w_{p_m}$. Hence, by condition 2 of P-RAM-W, $w_{p_0} <_{L_{p_0}} w_{p_m}$ and $r_{p_0} <_{L_{p_0}} w_{p_m}$ since $r_{p_0} <_{L_{p_0}} w_{p_0}$. Thus C is P-RAM-R.

Computation 3 $\begin{cases} p : w(y)0\ r(y)1\ w(x)1 \\ q : w(x)0\ r(x)1\ w(y)1 \end{cases}$

In fact, the following two computations, computations 3 and 4 show that P-RAM-W is strictly stronger than P-RAM-R, which is strictly stronger than P-RAM-A. Consider computation 3. The linearizations $<_{L_p} = w_p(y)0\ w_q(x)0\ w_q(y)1\ r_p(y)1\ w_p(x)1$ and $<_{L_q} = w_q(x)0\ w_p(y)0\ w_p(x)1\ r_q(x)1\ w_q(y)1$ satisfy the conditions of P-RAM-A since each maintains program order. Notice, however, that necessarily $w_q(y)1 <_{L_p} r_p(y)1 <_{L_p} w_p(x)1$ and $w_p(x)1 <_{L_q} r_q(x)1 <_{L_q} w_q(y)1$ to satisfy validity and program order. Hence it is impossible to find linearizations that satisfy condition 2 of definition 5.3.1 of P-RAM-R. Thus computation 3 is P-RAM-A but not P-RAM-R.

Computation 4 $\begin{cases} p : w(x)1\ w(y)1\ r(y)0\ r(x)1 \\ q : w(x)0\ w(y)0\ r(y)1\ r(x)0 \end{cases}$

Consider computation 4. The linearizations $<_{L_p} = w_q(x)0\ w_p(x)1\ w_p(y)1\ w_q(y)0\ r_p(y)0\ r_p(x)1$ and $<_{L_q} = w_p(x)1\ w_q(x)0\ w_q(y)0\ w_p(y)1\ r_q(y)1\ r_q(x)0$ clearly satisfy condition 1 of P-RAM-R and also satisfy condition 2 since no write in computation 4

is preceded by a read in program order. However, according to process p, necessarily $w_p(y)1 <_{L_p} w_q(y)0 <_{L_p} r_p(y)0$. And since the linearizations must also maintain program order, $w_q(x)0 <_{L_p} w_q(y)0$. Hence, in any linearization for p, $w_q(x)0 <_{L_p} r_p(x)1$. Thus, to ensure that the sequence is valid, $w_q(x)0 <_{L_p} w_p(x)1$. Using a similar argument, necessarily $w_p(x)1 <_{L_q} w_q(x)0$ which violates condition 2 of P-RAM-W. Thus P-RAM-W is a strictly stronger memory consistency condition than P-RAM-R.

Neither P-RAM-A nor P-RAM-R nor P-RAM-W is comparable with coherence.

Computation 5 $\begin{cases} p : w(x)0\ r(x)1 \\ q : w(x)1\ r(x)0 \end{cases}$

In Computation 5, let $<_{L_p} = w(x)0\ w(x)1\ r(x)1$ and $<_{L_q} = w(x)1\ w(x)0\ r(x)0$. Then $<_{L_p}$ (respectively, $<_{L_q}$) is a linearization of the actions by p (respectively, q) together with all the write actions, which maintains program order and satisfies condition 2 of P-RAM-W. Hence Computation 5 is P-RAM-A and P-RAM-R and P-RAM-W. Since it is not possible to construct a linearization of all the actions to location x that maintains program order, it is not coherent.

Computation 6 $\begin{cases} p : w(x)0\ w(x)1\ w(y)2 \\ q : r(y)2\ r(x)0 \end{cases}$

For Computation 6, let the object linearizations be: $<_{L_x} = w(x)0\ r(x)0\ w(x)1$ and $<_{L_y} = w(y)2\ r(y)2$. Both these linearizations maintain program order and thus the computation is coherent. Since it is not possible to construct a linearization for the actions by q together with the writes by p that extends program order, the computation is not P-RAM-A and thus neither P-RAM-R nor P-RAM-W.

Chapter 6

Processor Consistency

The term processor consistency was first used by Goodman [Goo89] to define a memory consistency model for distributed systems. Many have used the term processor consistency [ABJ+93, GLL+90, KNA93, Mos93, GGH93] but do not actually define equivalent memory models, although the differences between them are often subtle. All have in common Goodman's original intentions of modeling a system that is coherent but not as strong as sequential consistency.

Goodman defines processor consistency as being stronger than weak ordering (meaning coherence). Furthermore,

> the order in which writes from two processors occur, as observed by themselves or a third processor need not be identical, but writes issuing from any processor may not be observed in any order other than that in which they are issued. [Goo89]

Thus Goodman allows the interleaving of writes by two different processors to be viewed differently by each processor, as long as program order is maintained. But this description of processor consistency is somewhat vague. For example, it is not clear whether reads must be executed in program order or if all writes must be visible to all processes.

The following sections explore the different definitions of processor consistency, translate each into the framework, and illustrate the differences between them.

6.1 Processor Consistency as defined by Ahamad et al.

Ahamad et al. [ABJ+93] formalized Goodman's definition. This formalism, translated to the framework, becomes,

Definition 6.1.1 *Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCG if for each processor $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ such that*

1. $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$, *and*
2. $\forall q \in P$ *and* $\forall x \in J$ $(O_w|x, <_{L_p}) = (O_w|x, <_{L_q})$.

James and Singh also use this definition of processor consistency and state that it is the intersection of coherence and P-RAM-A [JS96]. This is incorrect. Any computation for which there is a linearization for each process that satisfies both of the conditions of P-RAM-A and coherence (definition 4.2.2) is also PCG, since these same process linearizations will satisfy the conditions of PCG. However, there are computations where there is one possible set of linearizations that satisfies P-RAM-A, and another set of linearizations that satisfies coherence, but no single set of linearizations that satisfies both.

Computation 7 $\begin{cases} p : w(x)0\ w(y)0 \\ q : r(y)0\ w(x)1 \\ r : r(x)1\ r(x)0 \end{cases}$

Consider, for example, Computation 7. The object linearizations of the actions of Computation 7 $<_{L_x} = w(x)1\ r(x)1\ w(x)0\ r(x)0$ and $<_{L_y} = w(y)0\ r(y)0$ satisfy definition 4.2.1 of coherence since each maintains program order. Thus Computation 7 is coherent. The process linearizations $<_{L_p}= w(x)0\ w(y)0\ w(x)1$, $<_{L_q}= w(x)0$

$w(y)0\ r(y)0\ w(x)1$, and $<_{L_r}= w(x)1\ r(x)1\ w(x)0\ r(x)0\ w(y)0$ contain the processes' own actions and others' writes while maintaining program order. Thus Computation 7 is P-RAM-A. And, since p and q agree on the ordering of all write actions and since r's process computation contains no write actions, condition 2 of P-RAM-W is also satisfied. Hence Computation 7 is also P-RAM-R and P-RAM-W. However, it is necessary to change the order in which the writes to location x are perceived by process r to get a linearization for r. Since condition 2 of PCG requires that the writes to the same object must be perceived by each process in the same order, Computation 7 is not PCG.

Any linearization that satisfies definition 6.1.1 will also satisfy definition 4.2.2 of coherence. Computation 6 on page 34 is coherent, but is not PCG, since it is not P-RAM-A. Hence, PCG is strictly stronger than coherence.

Computation 3 on page 33, which is neither P-RAM-W nor P-RAM-R, is PCG, since the sequences $<_{L_p} = w_p(y)0\ w_q(x)0\ w_q(y)1\ r_p(y)1\ w_p(x)1$ and $<_{L_q} = w_p(y)0\ w_q(x)0\ w_p(x)1\ r_q(x)1\ w_q(y)1$ are linearizations that satisfy program order and agree on the ordering of writes to the same object. Hence, PCG and P-RAM-R and PCG and P-RAM-W are incomparable memory consistency models.

6.2 Processor Consistency as implemented on the VAX 8800

Goodman states that the VAX 8800 [FKH87] is an example of a system that satisfies processor consistency [Goo89]. But he does not clarify if a system that is processor consistent must guarantee at least as much as the VAX 8800 or if something weaker is also acceptable. The VAX architecture is described by Fu, Keller and Haduch

[FKH87] and is summarized here, first informally, and then more precisely. A formal definition of the memory consistency model that is implemented by the VAX 8800 is then presented.

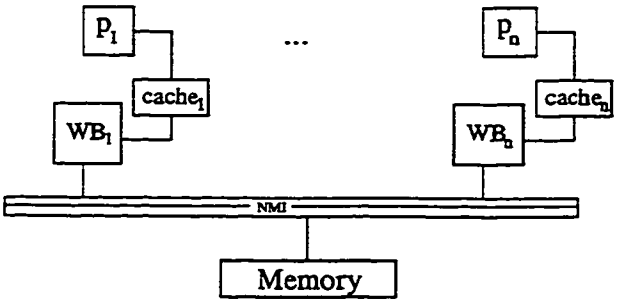


Figure 6.1: *M_{VAX} the VAX 8800 machine*

Each processor in the VAX 8800 has a cache and a write buffer and is connected to the NMI bus which is connected to the memory [FKH87]. The NMI bus has two channels, one for reads and one for writes. The reads are sent into the NMI bus in program order and the writes are sent into the NMI bus in program order, but, since they are on different channels, reads with respect to writes do not necessarily remain in program order. The NMI bus does, however, ensure that two events to the same location remain in program order.

The write buffer can contain one octaword. Consecutive writes that change the same block (but not necessarily the same location) are placed in the buffer. If a write tries to update a location outside of that block, the buffer is emptied (the pending writes are placed on the NMI bus and sent to the memory), and this new write replaces it.

When a processor does a write, the cache is first checked. If the location to be updated is in the cache, the cache is updated, otherwise, the cache is left unchanged.

In either case, the write is then placed in the write buffer. Upon a read action, first the cache is checked. If the location to be read is in the cache the value in the cache is used. If it is not in the cache the write buffer is checked. If there is a pending write in the write buffer to the location being read, the write buffer is emptied and the read is subsequently sent to memory. If there is no such pending write, the read is simply sent to the memory. If the location read was not in the cache, the cache is updated to include the location read such that it contains the value read. It is not clear whether the processors wait for the response to a read before initiating the next action; I have assumed that they do. I have also assumed that a processor blocks until the cache has processed an action before initiating the next. Finally, I assume that the cache size is unbounded.

The cache is kept current by using a snoopy bus. If processor p has object x in the cache, and some other processor q puts a write to x on the NMI bus, object x is marked invalid in p's cache.

6.2.1 M_{VAX} and the memory model it implements

I will now present a precise definition of the VAX 8800. Recall the use of "matching" and "corresponding" from section 3. Let M_{VAX} be the machine in figure 6.1 and as described above. A read action $(read, x, \lambda, v)$ of processor p on M_{VAX} is implemented by the following ordered sequence of events:

1. p sends processor$_p$-read-request(x), followed by

2. either a matching cache$_p$-reply(x, v) to p, or a matching memory-reply$_p$(x, v) to p.

A write action $(write, x, v, \lambda)$ of p is implemented by the following ordered sequence of events:

1. p sends processor$_p$-write-request(x, v), then possibly
2. a matching cache$_p$-update(x, v), followed by
3. the matching memory-update$_p(x, v)$.

Furthermore, any execution E on M_{VAX}, which is a sequence of these six events, will also meet all the following constraints, for all processors p, locations x and values $u, v, v', \bar{v}$:

1. Cache events are in program order. That is, cache$_p$ events are in the same order in E as the matching processor$_p$ events.
2. Reads are received by memory in program order. That is, memory-reply$_p$ events are in the same order in E as the matching processor$_p$ events.
3. Writes are received by memory in program order. That is, memory-update$_p$ events are in the same order as the matching processor$_p$ events in E.
4. Actions to the same location are received by memory in program order. That is, the set of all memory-reply$_p$ and memory-update$_p$ events to the same location are in the same order in E as the matching processor$_p$ events.
5. A location x can only be in cache at some point γ in the execution if there is some preceding read of that location by the processor, the location has not been invalidated, and all actions to x between the read and γ result in a cache-hit. That is, for each cache$_p$-reply(x, v) or cache$_p$-update(x, v) event, c, with

matching processor$_p$ event α_c, there is some memory-reply$_p(x, u)$ event m, with matching processor$_p$-read-request event α_m, such that $m \xrightarrow{E} c$ and such that

(a) for any processor $q \neq p$ there does not exist any memory-update$_q(x, v')$ event, e, such that $m \xrightarrow{E} e \xrightarrow{E} c$, and

(b) for each processor$_p$-read-request(x) or processor$_p$-write-request$(x, \bar{v})$, α, where $\alpha_m \xrightarrow{E} \alpha \xrightarrow{E} \alpha_c$, there is some matching cache$_p$ event.

6. A read of cache returns the value that is in cache at the location read. That is, for each cache$_p$-reply(x, v) event, c, find the subsequence of events in E that precede c in E containing all memory-reply events from x and all cache$_p$-update events of x. If memory-reply(x, u) is the last event in the subsequence then $v = u$, otherwise, if cache$_p$-update(x, v') is the last event in the subsequence, then $v = v'$.

7. A read of the memory returns the value that is in memory at the location read. That is, for each memory-reply$_p(x, v)$ event, m_r, if there is some memory-update$_q$ event, for any processor q, that precedes m_r in E and if m_u is the latest such memory-update$_q(x, u)$ event, then $v = u$. If no such m_u exists, then v is equal to the initialized value of x.

8. Actions at the cache are blocking. That is, for each cache$_p$ event, c, with matching processor$_p$ event α_c, there is no processor$_p$ event between α_c and c in E.

9. Read actions are blocking. That is, for each memory-reply$_p$ event, m, with matching processor$_p$-read-request event, α_m, there is no processor$_p$ event be-

tween α_m and m in E.

Notice that in M_{VAX}, the value written by some write by some processor q to some location x will never be read by p if there is some write by p to x, such that the memory-update$_q$ event is between the processor$_p$-write-request event and the memory-update$_p$ event in the execution. Hence, those writes could be treated as invisible writes in p's view. Furthermore, consider any two reads by some processor p, both of location x. Both could return values written by the same write by p, but one might receive the value from the cache and the other from the memory. Hence, from p's viewpoint writes by p can occur twice. In the following, $(A \uplus B)$ denotes the disjoint union of A and B, and if $x \in A \cap B$ then the copy of x in A is denoted x_A and the copy of x in B is denoted x_B. For any computation C containing set of actions O of some system (P, J), let, for some $p \in P$, $(O|p \uplus O_w, \longrightarrow)$ be any total order. Then $O_{\overrightarrow{cache_p}} = \{r \mid \exists r'$ such that $r, r' \in (O_r|p)|x \wedge r' \overset{prog}{\longrightarrow} r \wedge \forall q \neq p \nexists w \in (O_w|q)|x$ such that $r' \longrightarrow w \longrightarrow r\}$ and $O_{\overrightarrow{invisible_p}} = \{w \mid w \in (O_w|x \backslash O|p) \wedge \exists w' \in (O_w|p)|x$ such that $w'_{O|p} \longrightarrow w \longrightarrow w'_{O_w}\}$ and $O_{memupdates_p} = \{w_{O_w} \mid w \in O_w|p\}$

Definition 6.2.1 *Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCVax if $\forall p \in P$ there is some total order $(O|p \uplus O_w, \overset{view_p}{\longrightarrow})$ such that*

1. $(O_w, \overset{prog}{\longrightarrow}) \subseteq (O_w, \overset{view_p}{\longrightarrow})$, *and*
2. $(O|p, \overset{prog}{\longrightarrow}) = (O|p, \overset{view_p}{\longrightarrow})$, *and*
3. $\forall q \in P\ (O_w, \overset{view_p}{\longrightarrow}) = (O_w, \overset{view_q}{\longrightarrow})$, *and*
4. $\forall w \in O_w|p\ w_{O|p} \overset{view_p}{\longrightarrow} w_{O_w}$, *and*

5. $\forall x \in J$, $\forall r \in ((O_r \setminus O_{cache_p}^{\overrightarrow{view_p}})|p)|x$ *and* $\forall w \in (O_w|p)|x$ *if* $w \overrightarrow{prog} r$ *then* $w_{O_w} \overrightarrow{view_p} r$, *and*

6. *if* $((O|p \uplus O_w) \setminus (O_{invisible_p}^{\overrightarrow{view_p}} \cup O_{memupdates_p}), \overrightarrow{view_p})$ *is a linearization.*

In the remainder of this section, $O_{invisible_p}$ and O_{cache_p} are used to denote $O_{invisible_p}^{\rightarrow}$ and $O_{cache_p}^{\rightarrow}$ respectively when the total order $\longrightarrow$ is understood.

6.2.2 M_{VAX} implements exactly PCVax

Theorem 6.2.2 *M_{VAX} implements exactly PCVax.*

The theorem follows directly from the following two lemmas.

Lemma 6.2.3 *Any PCVax computation is the result of some execution on M_{VAX}.*

Proof: To show that all PCVax computations arise from some execution on M_{VAX}, I first construct an execution merging the total orders $(O|p \uplus O_w, \overrightarrow{view_p})$ that satisfy PCVax. I then show that this execution satisfies all the constraints of an execution on M_{VAX}.

Let C be any PCVax computation of some system (P, J) containing actions O. Let the processor that implements process $p \in P$ be named $\hat{p}$ and let the objects in J be the locations in M_{VAX}. Construct the sequence E containing exactly all the processor, cache and memory events using the total orders $(O|p \uplus O_w, \overrightarrow{view_p})$ for each process p that satisfy PCVax in the following manner. Informally, the $\overrightarrow{view_p}$ total orders are merged, such that the writes from O_w represent the update events at the memory, the writes from $O|p$ represent the processor$_p$-write-request events, the reads from $O|p \cap O_{cache_p}$ are the cache$_p$-reply events and such that the remaining reads

from $O|p$ are the memory-reply$_p$ events. The processor$_p$-read-request events are then added such that they immediately precede the matching reply events and the necessary cache$_p$-update events are subsequently added such that they immediately follow the matching processor$_p$-write-request events. Formally, if $(O_w, \xrightarrow{view_p}) = w_1, w_2, \ldots w_k$ (which will be the same sequence independent of which process p is chosen by condition 3 of PCVax) then, initially $E = m_w^1, m_w^2, \ldots m_w^k$, a sequence of all memory-update events associated with the actions of O_w in C such that m_w^i is the memory-update$_{\hat{p}}$ event corresponding to the write action w_i by process p implemented by processor $\hat{p}$. Let $S_0^p, S_1^p, \ldots, S_k^p$ be the subsequences of $(O|p \uplus O_w, \xrightarrow{view_p})$ defined by $(O|p \uplus O_w, \xrightarrow{view_p}) = S_0^p, w_1^p, S_1^p, w_2^p, \ldots, w_k^p, S_k^p$. Let E_i^p be the sequence of memory-reply$_{\hat{p}}$ events, cache$_{\hat{p}}$-reply and processor$_{\hat{p}}$-write-request events corresponding to actions in $O|p$ such that the j^{th} event in E_i^p corresponds to the j^{th} action in S_i^p. Furthermore if the j^{th} action in S_i^p is o_j and

1. if o_j is a write action by p then the j^{th} event in E_i^p is a processor$_{\hat{p}}$-write-request event, otherwise

2. if $o_j \in O_r \cap O_{cache_p}$ then the j^{th} event in E_i^p is a cache$_{\hat{p}}$-reply event, otherwise

3. if $o_j \in O_r \setminus O_{cache_p}$ then the j^{th} event in E_i^p is a memory-reply$_{\hat{p}}$ event.

Insert each sequence E_i^p into E anywhere between the events, m_w^i and m_w^{i+1} (maintaining the order of E_i^p). To add the necessary cache$_{\hat{p}}$-update events for each processor $\hat{p}$, consider each processor$_{\hat{p}}$-write-request event in E in turn. Let α be any processor$_{\hat{p}}$-write-request event to some location x. If there is some memory-reply$_{\hat{p}}$ event from location x, m_r, preceding α in E such that there is no memory-update

event of location x by some processor $\hat{q} \neq \hat{p}$ between α and m_r in E, add the cache$_{\hat{p}}$ event matching α to E such that it immediately follows α. Finally, for each processor $\hat{p}$ and each memory-reply$_{\hat{p}}$ and cache$_{\hat{p}}$-reply event in E, place the matching processor$_{\hat{p}}$-read-request event immediately in front of the reply event in E.

The construction of E satisfies the ordering requirement of M_{VAX}. Notice that processor$_{\hat{p}}$ events are ordered in the same order as the corresponding actions from $O|p$ are ordered in $(O|p \uplus O_w, \xrightarrow{view_p})$, which are in program order by condition 2 of PCVax. Constraint 1 of M_{VAX} is also satisfied by E since cache$_{\hat{p}}$ events immediately follow the matching processor$_{\hat{p}}$ events. Since memory$_{\hat{p}}$-reply events are in the same order as the corresponding reads are ordered in $(O|p \uplus O_w, \xrightarrow{view_p})$, and using condition 2 of PCVax, constraint 2 of M_{VAX} is also satisfied by E. Memory-update events are in the same order as the corresponding actions from the set O_w are ordered in $(O_w, \xrightarrow{view_p})$, which are in program order by condition 3 of PCVax. Thus constraint 3 of M_{VAX} is also satisfied by E.

The memory events in E correspond to either write actions from the set O_w or read actions not in any O_{cache_p}. By condition 1 of PCVax, memory-update$_{\hat{p}}$ events at the same location will be in the same order as the matching processor$_{\hat{p}}$-write-request events. By condition 2 of PCVax, memory$_{\hat{p}}$-reply events from location x will be in the same order as the matching processor$_{\hat{p}}$-read-request events. Now consider any memory$_{\hat{p}}$-reply event from location x, m_r, and any memory-update$_{\hat{p}}$ event at location x, m_u. Let α_r and α_u be the processor$_{\hat{p}}$ events matching, respectively, m_r and m_u and let r and w be the read and write actions corresponding, respectively, to m_r and m_u. If $\alpha_u \xrightarrow{E} \alpha_r$ then $w \xrightarrow{prog} r$ and by condition 5 of PCVax, $w_{O_w} \xrightarrow{view_p} r$. Thus, by the construction of E, $m_u \xrightarrow{E} m_r$. If $\alpha_r \xrightarrow{E} \alpha_w$ then $r \xrightarrow{prog} w$ and by condition 2 of

PCVax, $r \xrightarrow{view_p} w_{O|p}$. By condition 6 of PCVax, $w_{O|p} \xrightarrow{view_p} w_{O_w}$. Thus, $m_r \xrightarrow{E} m_u$ since $r \xrightarrow{view_p} w_{O_w}$. Hence, constraint 4 of M_{VAX} is satisfied by E.

By construction of E reads and cache actions are clearly blocking, hence conditions 8 and 9 of M_{VAX} are also satisfied by E.

To show that constraint 5 of M_{VAX} is satisfied, consider any cache$_{\hat{p}}$ event, c, with matching processor$_{\hat{p}}$ event α_c and corresponding action o. If c is a cache$_{\hat{p}}$-reply event from location x, then, by construction of E, $o \in O_{cache_p}$. By the definition of O_{cache_p}, there exists some read by p of x preceding o in $(O|p \uplus O_w, \xrightarrow{view_p})$ such that there are no writes to x by any process $q \neq p$ between this read and o in $(O|p \uplus O_w, \xrightarrow{view_p})$. Consider the earliest such read, r'. By definition of O_{cache_p}, $r' \notin O_{cache_p}$. Hence, E contains the memory-reply$_{\hat{p}}$ event, m, corresponding to r' and $m \xrightarrow{E} c$. And since there are no writes by any process $q \neq p$ between r' and r in $(O|p \uplus O_w, \xrightarrow{view_p})$, there are no memory-update$_{\hat{q}}$ events between m and c in E. Hence, c satisfies constraint 5a of M_{VAX}. Let α_m be the processor$_{\hat{p}}$-read-request event matching m. Clearly, $\alpha_m \xrightarrow{E} \alpha_c$. Consider any processor$_{\hat{p}}$ event to x, α, such that $\alpha_m \xrightarrow{E} \alpha \xrightarrow{E} \alpha_c$. If α is a processor$_{\hat{p}}$-read-request with corresponding read action $\hat{r}$, then, since processor$_{\hat{p}}$ events are in program order by the construction of E, $r' \xrightarrow{prog} \hat{r} \xrightarrow{prog} o$. Hence, by condition 2 of PCVax, $r' \xrightarrow{view_p} \hat{r} \xrightarrow{view_p} o$, and $\hat{r} \in O_{cache_p}$ by definition of O_{cache_p}. Hence, α is immediately followed by its matching cache$_{\hat{p}}$-reply event in E, satisfying 5b of M_{VAX}. If α is a processor$_{\hat{p}}$-write-request then, since α appears between m and c, there is no memory-update event to x by some processor $\hat{q} \neq \hat{p}$ between m and α and thus, by the construction of E, α is immediately followed by its matching cache$_{\hat{p}}$-update event in E, satisfying 5b of M_{VAX}. Using a similar

argument, it can be shown that c also satisfies constraint 5 if c is a cache$_{\hat{p}}$-update event.

To show that constraint 6 of M_{VAX} is also satisfied, consider, for any processor $\hat{p}$, any cache$_{\hat{p}}$-reply event from location x, c_r and let m'_r be the last preceding memory-reply$_{\hat{p}}$ event from x in E. Let r and r' be the read actions corresponding to c_r and m'_r respectively. Since $m'_r \xrightarrow{E} c_r$, by construction of E $r' \xrightarrow{view_p} r$. Also by the construction of E, $r \in O_{cache_p}$ and r' is the last read action of x not in O_{cache_p} preceding r in $(O|p \uplus O_w, \xrightarrow{view_p})$. Thus, by the definition of O_{cache_p}, there is no $w \in (O_w|q)|x$ for any $q \neq p \in P$ such that $r' \xrightarrow{view_p} w \xrightarrow{view_p} r$. Hence, by the construction of E, there does not exist a memory-update$_{\hat{q}}$ event of x, m_w, where $\hat{q} \neq \hat{p}$, such that $m'_r \xrightarrow{E} m_w \xrightarrow{E} c_r$. Consider two cases:

1. If there is some cache$_{\hat{p}}$-update event of x between m'_r and c_r, let c_u be the last such event and let α_u be the processor$_{\hat{p}}$-write-request event matching c_u. By construction of E, $\alpha_u \xrightarrow{E} c_u$, and, since cache$_{\hat{p}}$-update events are immediately preceded by the matching processor event, also $m'_r \xrightarrow{E} \alpha_u \xrightarrow{E} c_u \xrightarrow{E} c_r$. Let w be the write action corresponding to c_u. Since $m'_r \xrightarrow{E} \alpha_u \xrightarrow{E} c_r$, by the construction of E, $r' \xrightarrow{view_p} w_{O|p} \xrightarrow{view_p} r$. Because cache$_{\hat{p}}$-update events are in the same order as the corresponding write actions, w is the last write to x such that the copy from $O|p$ appears between r' and r in $(O|p \uplus O_w, \xrightarrow{view_p})$. Furthermore, there are no writes to x by any process $q \neq p$ between $w_{O|p}$ and r in $(O|p \uplus O_w, \xrightarrow{view_p})$ and the writes by p from O_w are in $O_{memupdates_p}$ thus w will be the last write to precede r in $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p})$, which is a linearization. Hence, r contains the value that w wrote and thus c_r returns the value that c_u

wrote, satisfying constraint 6 of M_{VAX}.

2. If there is no cache$_{\hat{p}}$-update of x between m'_r and c_r, then there is also no write by p of x whose copy from $O|p$ appears between r' and r in $(O|p \uplus O_w, \xrightarrow{view_p})$. Since there is no write by some process $q \neq p$ to x between r' and r in $(O|p \uplus O_w, \xrightarrow{view_p})$ and all writes by p to x between r' and r will be in $O_{memupdates_p}$, there will be no write to x between r' and r in $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p})$. Hence, r returns the same value as r', and c_r returns the same value as m'_r, satisfying constraint 6 of M_{VAX}.

Finally, to show that constraint 7 of M_{VAX} is satisfied, consider, for any processor $\hat{p}$ and any location x, any memory-reply$_{\hat{p}}$ event from location x, m_r. Let m_u be the last memory-update event of x preceding m_r in E and let r and w be the read and write actions corresponding, respectively, to m_r and m_u. If m_u is a memory-update$_{\hat{q}}$ event for some processor $\hat{q} \neq \hat{p}$, then w will be the last write to x preceding r in $((O|p \uplus O_w) \setminus O_{memupdates_p}, \xrightarrow{view_p})$ since any $w' \in (O_w|p)|x$ such that $w \xrightarrow{view_p} w'_{O|p} \xrightarrow{view_p} r$ would imply, by condition 2 of PCVax, that $w' \xrightarrow{prog} r$. By construction of E, $r \notin O_{cache_p}$, hence, by condition 5 of PCVax, $w'_{O_w} \xrightarrow{view_p} r$. By condition 4 of PCVax, $w'_{O|p} \xrightarrow{view_p} w'_{O_w} \xrightarrow{view_p} r$. Hence, $w \xrightarrow{view_p} w'_{O|p} \xrightarrow{view_p} w'_{O_w} \xrightarrow{view_p} r$ implying that m_u was not the last memory-update preceding m_r in E. Thus w will be the last write to x preceding r in $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p})$, unless $w \in O_{invisible_p}$. Assume, to reach a contradiction, that $w \in O_{invisible_p}$. Hence, there exists $w' \in (O_w|p)|x$ such that $w'_{O|p} \xrightarrow{view_p} w \xrightarrow{view_p} w'_{O_w}$. Since w is the last write to x to precede r in $(O|p \uplus O_w, \xrightarrow{view_p})$, $w'_{O|p} \xrightarrow{view_p} w \xrightarrow{view_p} r \xrightarrow{view_p} w'_{O_w}$. Hence, by condition 5 of PCVax, $r \xrightarrow{prog} w'$. But then $w'_{O|p} \xrightarrow{view_p} r$ violates condition 2 of PCVax, hence $w \notin O_{invisible_p}$. If m_u matches

a processor$_{\bar{p}}$ event then all writes by processes other than p between $w_{O|p}$ and w_{O_w} in $(O|p \uplus O_w, \stackrel{view_p}{\longrightarrow})$ will be in $O_{invisible_p}$, and since all writes by p from the set $O|p$ are in program order by condition 2 of PCVax, and since all writes by p from the set O_w are in program order by condition 1 of PCVax, w will also be the last write to precede r in $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \stackrel{view_p}{\longrightarrow})$. Hence, m_r satisfies constraint 7 of M_{VAX} and E is an execution satisfying all the constraints of M_{VAX}.

■

Lemma 6.2.4 *Any computation arising from an execution E of M_{VAX} is PCVax.*

Proof: To show that any computation C arising from an execution E of M_{VAX} is PCVax, I first construct, for each process p, a total order $(O|p \cup O_w, \stackrel{view_p}{\longrightarrow})$, using the subsequence of E containing the processor$_p$, cache$_p$, memory-reply$_p$ and memory-update events. I then show that these total orders satisfy PCVax.

Consider any computation C arising from an execution E of M_{VAX} containing actions O. Let $P = \{\bar{p} \mid \text{processor } p \text{ is in } M_{VAX}\}$ be a set of processes such that the i^{th} action-invocation in each $\bar{p}$ corresponds to the i^{th} processor$_p$ event in E. The set of objects J is the set of all locations in M_{VAX}. For each processor p, let E_p be the subsequence of E containing exactly all cache$_p$-reply, memory-reply$_p$, memory-update and processor$_p$-write-request events. For each process $\bar{p}$, let $(O|\bar{p} \uplus O_w, \stackrel{view_p}{\longrightarrow})$ be the sequence such that the i^{th} action in $(O|\bar{p} \uplus O_w, \stackrel{view_p}{\longrightarrow})$ corresponds to the i^{th} event in E_p. Specifically, if the i^{th} event in E_p is e_i and

1. if e_i is a memory-update event, then the i^{th} action in $(O|\bar{p} \uplus O_w, \stackrel{view_p}{\longrightarrow})$ is from O_w,

2. if e_i is a cache$_p$-update event, then the i^{th} action in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ is from $O|\bar{p}$,

3. if e_i is a memory-reply$_p$ event, then the i^{th} action in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ is from $O|\bar{p}$,

4. if e_i is a processor$_p$-write-request, then the i^{th} action in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ is a write action from $O|\bar{p}$.

The subsequence $(O_w, \xrightarrow{view_p})$ maintains program order by constraint 3 of M_{VAX}, hence condition 1 of PCVax is satisfied. To show that condition 2 of PCVax is also satisfied, consider any $o_1, o_2 \in O|\bar{p}$ such that $o_1 \xrightarrow{prog} o_2$. Let the i and j, respectively, be the positions of $o_{1_{O|\bar{p}}}$ and $o_{2_{O|\bar{p}}}$ in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$. The i^{th} and j^{th} events in E_p can each either be a cache$_p$-reply event, a memory-reply$_p$ event or a processor$_p$-write-request event by the construction of $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$. Thus, there are 9 possible combinations:

1. If the i^{th} and j^{th} events in E_p are cache$_p$-reply events, then these two events are in the same order as the processor$_p$ events by constraint 1 of M_{VAX}. Hence $o_1 \xrightarrow{view_p} o_2$.

2. If the i^{th} and j^{th} events in E_p are processor$_p$-write-request events, then clearly $o_1 \xrightarrow{view_p} o_2$.

3. If the i^{th} and j^{th} events in E_p are memory$_p$-reply events then $o_1 \xrightarrow{view_p} o_2$ since, by constraint 2 of M_{VAX} memory$_p$-reply events are in the same order as the matching processor$_p$-read-request events, which are in program order.

4. If the i^{th} event in E_p is a cache$_p$-reply event, c, and the j^{th} event in E_p is a processor$_p$-write-request event, α, let α_c be the processor event matching c. Since $o_1 \xrightarrow{prog} o_2$, $\alpha_c \xrightarrow{E} \alpha$. By constraint 8 of M_{VAX}, $\alpha_c \xrightarrow{E} c \xrightarrow{E} \alpha$, thus $o_1 \xrightarrow{view_p} o_2$.

5. If the i^{th} event in E_p is a memory-reply$_p$ event and the j^{th} event in E_p is a processor$_p$-write-request event then by a similar argument as the previous case (using constraint 9 instead of 8), $o_1 \xrightarrow{view_p} o_2$.

6. If the i^{th} event in E_p is a processor$_p$-write-request event, α, and the j^{th} event in E_p is a cache$_p$-reply event, r, then let α_r be the processor$_p$ event matching r. Since $o_1 \xrightarrow{prog} o_2$, $\alpha \xrightarrow{E} \alpha_r$. Thus, by the ordering of events corresponding to a read action, $\alpha \xrightarrow{E} \alpha_r \xrightarrow{E} r$. Hence $o_1 \xrightarrow{view_p} o_2$.

7. If the i^{th} event in E_p is a processor$_p$-write-request event and the j^{th} event in E_p is a memory-reply$_p$ then by the same argument as the previous case $o_1 \xrightarrow{view_p} o_2$.

8. If the i^{th} event in E_p is a cache$_p$-reply event, c, and the j^{th} event in E_p is a memory-reply$_p$ event, m, then let α_c and α_m be the processor event matching c and m respectively. Since $o_1 \xrightarrow{prog} o_2$, $\alpha_c \xrightarrow{E} \alpha_m$. By constraint 8 of M_{VAX}, $\alpha_c \xrightarrow{E} c \xrightarrow{E} \alpha_m$, and, by the ordering of events corresponding to a read action, $\alpha_c \xrightarrow{E} c \xrightarrow{E} \alpha_m \xrightarrow{E} m$. Hence $o_1 \xrightarrow{view_p} o_2$.

9. If the i^{th} event in E_p is a memory-reply$_p$ event and the j^{th} event in E_p is a cache$_p$-reply event then, by using a similar argument as the previous case (using constraint 9 instead of 8), $o_1 \xrightarrow{view_p} o_2$.

Thus condition 2 of PCVax is also satisfied by $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$.

Since each E_p has the same subsequence of memory-update events, condition 3 of PCVax is clearly also satisfied by $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$. Furthermore, since processor$_p$-write-request events precede matching memory-update$_p$ events by the ordering of events constraint of M_{VAX}, condition 4 of PCVax is also satisfied by each $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$.

To show that each $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ also satisfies condition 5 of PCVax, consider, for any object x, any $r \in ((O_r \setminus O_{cache_{\bar{p}}})|\bar{p})|x$ and any $w \in (O_w|\bar{p})|x$ such that $w \xrightarrow{prog} r$. By the definition of $O_{cache_{\bar{p}}}$, either there is no $r' \in (O_r|\bar{p})|x$ such that $r' \xrightarrow{prog} r$, or if $\exists r' \in (O_r|\bar{p})|x$ such that $r' \xrightarrow{prog} r$ then there is some $w' \in (O_w|\bar{q})|x$, $\bar{q} \neq \bar{p}$, such that $r' \xrightarrow{view_p} w' \xrightarrow{view_p} r$. Hence, by constraint 5 of M_{VAX}, r does not match a cache$_p$-reply event in E_p. By constraint 4 of M_{VAX}, the memory events corresponding to w and r must be in program order, hence, by construction of each $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$, $w_{O_w} \xrightarrow{view_p} r$.

Finally, to show that C is PCVax, each $<_{L_p} = ((O|\bar{p} \uplus O_w) \setminus (O_{invisible_{\bar{p}}} \cup O_{memupdates_{\bar{p}}}), \xrightarrow{view_p})$ must be a linearization. For any $x \in J$, consider any read action $r \in (O_r|\bar{p})|x$. Let w be the last write action to x preceding r in $<_{L_p}$. Let m_w and m_r be the memory events corresponding to w and r respectively, let c_w and c_r be the cache events corresponding to w and r respectively, and let α_w and α_r be the processor events corresponding to w and r respectively and let w and r be the i^{th} and j^{th} actions in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ respectively. There are 4 possible cases:

1. If the i^{th} and j^{th} events in E_p are both memory events in E_p, then $w \xrightarrow{view_p} r$ implies that $m_w \xrightarrow{E} m_r$. If $w \in O|\bar{p}$, then w is the copy from O_w by construction of $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$, and is in the set $O_{memupdates_{\bar{p}}}$, which is not possible since it precedes r in $<_{L_p}$. Hence $w \in O_w|\bar{q}$ for some process $\bar{q} \neq \bar{p}$ and m_w is some

memory-update$_q$ event. Consider now the following two facts: 1) since $w \notin O_{invisible_{\bar{p}}}$, there does not exist any $w' \in (O_w|\bar{p})|x$ such that $w'_{O|\bar{p}} \xrightarrow{view_p} w \xrightarrow{view_p} w'_{O_w}$; and 2) since w is the last write to x to precede r in $<_{L_p}$, there is no $\hat{w}$ such that $w \xrightarrow{view_p} \hat{w}_{O|\bar{p}} \xrightarrow{view_p} r$. These two facts imply that there is no write by $\bar{p}$ such that the copy from O_w appears between w and r in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$. They also imply that there is no $\bar{w} \in (O_w|\bar{s})|x$, for some process $\bar{s} \neq \bar{p}$ such that $w \xrightarrow{view_p} \bar{w} \xrightarrow{view_p} r$. If such a $\bar{w}$ did exists then necessarily $\bar{w} \in O_{invisible_{\bar{p}}}$ since $\bar{w} \notin <_{L_p}$. Hence, by the definition of $O_{invisible_{\bar{p}}}$, there is some $w' \in (O_w|\bar{p})|x$ such that $w'_{O|\bar{p}} \xrightarrow{view_p} \bar{w} \xrightarrow{view_p} w'_{O_w}$. But, by fact 1, it is not possible that $w'_{O|\bar{p}}$ precedes w in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$, and, by fact 2, $w'_{O|\bar{p}}$ can't occur between w and $\bar{w}$ in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$. Thus such a w' does not exist and thus $\bar{w}$ does not exist. Hence, m_w is also the last memory-update of location x to precede m_r in E, and by constraint 7 of M_{VAX}, m_r returns the value that m_w wrote, and hence r returns the 'correct' value in $<_{L_p}$.

2. If the i^{th} and j^{th} events in E_p are a cache$_p$-update and a memory-reply$_p$ event respectively, then w is necessarily by $\bar{p}$ and is not in the set $O_{memupdates_{\bar{p}}}$. Furthermore, $w \xrightarrow{view_p} r$ implies that $\alpha_w \xrightarrow{E} m_r$. By the ordering of events, $\alpha_w \xrightarrow{E} c_w$ and $\alpha_r \xrightarrow{E} m_r$. By constraints 8 and 9, read and cache actions are blocking, hence $\alpha_w \xrightarrow{E} c_w \xrightarrow{E} \alpha_r \xrightarrow{E} m_r$ and $\alpha_w \xrightarrow{E} \alpha_r$. Thus, by constraint 4 of M_{VAX}, $m_w \xrightarrow{E} m_r$. Since w is the last write by $\bar{p}$ to x preceding r in $<_{L_p}$, there does not exist any $w' \in (O_w|\bar{p})|x$ such that $w_{O|\bar{p}} \xrightarrow{view_p} w'_{O|\bar{p}} \xrightarrow{view_p} r$. Hence, there is no processor$_p$-write-request event to x between α_w and α_r, and thus, by constraint 4 of M_{VAX}, there is no memory-update$_p$ event of x between m_w and m_r. Fur-

thermore, any $\hat{w} \in O_w|\bar{q}$, where $\bar{q} \neq \bar{p} \in P$, appearing between w_{O_w} and r in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ could not be in the set $O_{invisible_{\bar{p}}}$ (and thus cannot exist by the choice of w) since any such $\hat{w}$ must be surrounded in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ by a write, $\bar{w}$, from the set $O|\bar{p}$ and the matching write from the set O_w. This implies that $w_{O_w} \xrightarrow{view_p} \bar{w}_{O_w}$. Hence, also $w_{O|\bar{p}} \xrightarrow{view_p} \bar{w}_{O_w}$ and by the choice of $\bar{w}$, $w_{O|\bar{p}} \xrightarrow{view_p} \bar{w}_{O|\bar{p}} \xrightarrow{view_p} \hat{w} \xrightarrow{view_p} r$, and it has already been shown that such a $\bar{w}$ does not exist. Thus, there are no memory-update events between m_w and m_r in E, and by constraint 7 of M_{VAX}, m_r returns the update value of m_w and r and w contain the same value.

3. If the i^{th} and j^{th} events in E_p are cache$_p$-update and cache$_p$-reply events respectively, then, by the construction of $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$, $c_w \xrightarrow{E} c_r$. Consider the subsequence S of E containing all memory-reply$_p$ events from x and all cache$_p$-update events of x that precede c_r in E. By the choice of w, there are clearly no other cache$_p$-update events of x between c_w and c_r in E. Thus, the last event in S is either c_w or some memory-reply$_p$ event. If c_w is the last event in S, then by constraint 6 of M_{VAX}, c_r returns the update value of c_w and thus r and w contain the same value. If some memory-reply$_p$, m'_r, is the last event in E, let α'_r be the matching processor$_p$-read-request event. By constraint 6 of M_{VAX}, c_r returns the same value as m'_r. It remains to show that m'_r returns the update value of c_w. Since $c_w \xrightarrow{E} m'_r \xrightarrow{E} c_r$ and since cache and read actions are blocking, $\alpha_w \xrightarrow{E} c_w \xrightarrow{E} \alpha'_r \xrightarrow{E} m'_r \xrightarrow{E} \alpha_r \xrightarrow{E} c_r$. Thus, by constraint 4 of M_{VAX}, $m_w \xrightarrow{E} m'_r$. Let r' be the read action corresponding to m_r. Then, by construction of $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$, $w_{O|\bar{p}} \xrightarrow{view_p} w_{O_w} \xrightarrow{view_p} r' \xrightarrow{view_p} r$ and $w <_{L_p} r' <_{L_p} r$.

By the choice of w, w is also the last write to x to precede r in $<_{L_p}$. The previous case already showed that r' and w contain the same value, thus r and w contain the same value.

4. If the i^{th} and j^{th} events in E_p are a memory-update and a cache$_p$-reply event respectively then $m_w \xrightarrow{E} c_r$. As was shown in case 1, $w \in O|\bar{q}$ for some process $\bar{q} \neq \bar{p}$. Thus, by constraint 5 of M_{VAX} there exists some memory-reply$_p$ event from x such that it appears between m_w and c_r in E. Let m'_r be the last such memory-reply$_p$ event and r' be the read action corresponding to m'_r. Since $m_w \xrightarrow{E} m'_r \xrightarrow{E} c_r$, by the construction of $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$, $w \xrightarrow{view_p} r' \xrightarrow{view_p} r$ and $w <_{L_p} r' <_{L_p} r$. By the choice of w, w is also the last write to x to precede r' in $<_{L_p}$ and case 1 shows that w and r will contain the same value. Furthermore, since w is the last write to x to precede r in $<_{L_p}$, there is no $w' \in (O_w|\bar{p})|x$ such that $w <_{L_p} r <_{L_p} w'_{O|\bar{p}} <_{L_p} r$. Thus there is no cache$_p$-update event of x between m'_r and c_r in E and by constraint 6 of M_{VAX}, c_r returns the same value as m'_r. Thus w and r contain the same value.

■

6.2.3 Comparing PCVax with other memory models

Computation 8
$$\begin{cases} p : w(x)1 \\ q : w(y)1 \\ r : w(x)0\ r(x)1\ r(y)0 \\ s : w(y)0\ r(y)1\ r(x)0 \end{cases}$$

Computation 8 is coherent, PCG, P-RAM-W (and thus also P-RAM-R and P-RAM-A) but is not PCVax nor SC. It is obviously not SC, since it is not possible to build a linearization of all the actions of Computation 8 that extends the program

order. The process linearizations $<_{L_p} = w_r(x)0\ w_s(y)0\ w_p(x)1\ w_q(y)1$, $<_{L_q} = w_r(x)0\ w_s(y)0\ w_p(x)1\ w_q(y)1$, $<_{L_r} = w_r(x)0\ w_s(y)0\ w_p(x)1\ r_r(x)1\ r_r(y)0\ w_q(y)1$ and $<_{L_s} = w_r(x)0\ w_s(y)0\ w_q(y)1\ r_s(y)1\ r_s(x)0\ w_p(x)1$ satisfy the definition of coherence (definition 4.2.2), of P-RAM-W, and of PCG. But notice that in Computation 8 the set O_{cache} is empty for each process since none of the actions is preceded in program order by a read action of the same object. Furthermore, processes r and s cannot agree on an ordering of writes (in particular the writes $w(x)1$ and $w(y)1$) and thus Computation 8 is not PCVax.

Computation 9 $\begin{cases} p : w(x)0\ r(y)1\ r(y)2\ r(y)3\ r(x)0 \\ q : w(y)1\ w(y)2\ w(x)6\ w(y)3 \end{cases}$

Computation 9 [HKV97, HKV98] is PCVax but not P-RAM-A (and thus also not P-RAM-R nor P-RAM-W nor PCG). The sequences $\xrightarrow{view_p} = w(x)0_{O|p}\ w(y)1\ r(y)1\ w(y)2\ r(y)2\ w(x)6\ w(y)3\ r(y)3\ w(x)0_{O_w}\ r(x)0$ and $\xrightarrow{view_q} = w(y)1_{O|p}\ w(y)1_{O_w}\ w(y)2_{O|p}\ w(y)2_{O_w}\ w(x)6_{O|p}\ w(x)6_{O_w}\ w(y)3_{O|p}\ w(y)3_{O_w}\ w(x)0$ maintain program order with respect to writes from O_w and all actions by the process itself. And $O_{cache_p} = O_{cache_q} = \phi$. All the read actions in each sequence maintain program order with respect to all writes from the set O_w, hence the sequences satisfy conditions 1, 2, and 5 of PCVax. Condition 3 is also satisfied since both $\xrightarrow{view_p}$ and $\xrightarrow{view_q}$ order the writes from the set O_w in the following sequence: $w(y)1\ w(y)2\ w(x)6\ w(y)3\ w(x)0$. And clearly condition 4 of PCVax is also satisfied. Note that $O_{invisible_p} = w(x)6$ and $O_{invisible_q} = \phi$, hence $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p}) = w(x)0\ w(y)1\ r(y)1\ w(y)2\ r(y)2\ w(y)3\ r(y)3\ r(x)0$ and $((O|q \uplus O_w) \setminus (O_{invisible_q} \cup O_{memupdates_q}), \xrightarrow{view_q}) = w(y)1\ w(y)2\ w(x)6\ w(y)3\ w(x)0$. These are both linearizations, hence Computation 9 is PCVax. But it is not P-RAM-A nor PCG because it is not possible to build a

linearization of $O|p \cup O_w$ that maintains program order.

This establishes that PCVax is strictly weaker than SC and is incomparable with PCG, P-RAM-W, P-RAM-R and P-RAM-A. The following claim, together with Computation 8, shows that PCVax is strictly stronger than coherence.

Claim 6.2.5 *Any computation that is PCVax is also coherent.*

Proof: Let C be any PCVax computation on system (P, J) with set of actions O. Choose any set of sequences $(O|p \uplus O_w, \xrightarrow{view_p})$, one for each process $p \in P$, that satisfy definition 6.2.1 to build the object sequences S_x, one for each $x \in J$ that contain all the actions to x in O. Initially, $S_x = (O_w|x, \xrightarrow{view_p})$ for any process p. (By condition 3 of PCVax, S_x will be the same sequence, regardless of the choice of p). Now, for each $x \in J$ and each $p \in P$, let $S_x^p = ((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p})|x, \xrightarrow{view_p}) = R_0^{x,p}, w_1^{x,p}, R_1^{x,p}, w_2^{x,p}, R_2^{x,p}, \ldots, w_{k_p}^{x,p}, R_{k_p}^{x,p}$, where each $w_i^{x,p}$ is a write action and each $R_i^{x,p}$ is a sequence of read actions. Note that each write action in S_x^p also appears in S_x (but not vice versa), but it is not obvious that these write actions appear in the same order in S_x and S_x^p. For each $0 < i \leq k_p$, insert, in order, $R_i^{x,p}$ into S_x directly after $w_i^{x,p}$ and insert $R_0^{x,p}$ into S_x, in order, before the first action of S_x.

Clearly, each S_x is a linearization. To show that each S_x also maintains program order, consider any two actions $o_1, o_2 \in (O|p)|x$ such that $o_1 \xrightarrow{prog} o_2$. If $o_1, o_2 \in O_w$, then by condition 1 of PCVax, o_1 precedes o_2 in S_x. If $o_1, o_2 \in O_r$, then there are 2 possible cases: 1) If $o_1, o_2 \in R_i^{x,p}$ for some index i then by condition 2 of PCVax, o_1 precedes o_2 in S_x; 2) If $o_1 \in R_i^{x,p}$ and $o_2 \in R_j^{x,p}$, for some indices i and j, then assume, to reach a contradiction, that o_2 precedes o_1 in S_x. By condition 2 of PCVax, o_1 precedes o_2 in S_x^p, thus $i < j$. Furthermore, exactly one of $w_i^{x,p}$ and $w_j^{x,p}$ is not by p.

Otherwise, $w_i^{x,p}$ and $w_j^{x,p}$ would appear in the same order in both S_x and S_x^p which would imply that o_1 precedes o_2 in S_x. Since $w_{j_{O_w}}^{x,p} \xrightarrow{view_p} w_{i_{O_w}}^{x,p}$, and since also $w_i^{x,p} \xrightarrow{view_p} w_j^{x,p}$, clearly $w_i^{x,p}$ is the write action by p and $w_{i_{O_p}}^{x,p} \xrightarrow{view_p} w_j^{x,p} \xrightarrow{view_p} w_{i_{O_w}}^{x,p}$. But this implies that $w_j^{x,p} \in O_{invisible_p}$ and thus $w_j^{x,p} \notin S_x^p$, leading to a contradiction.

If $o_1 \in O_w$ and $o_2 \in O_r$, let $o_2 \in R_i^{x,p}$. If $o_1 = w_i^{x,p}$ then o_1 precedes o_2 in S_x by construction of S_x. Otherwise, if $w_i^{x,p} \in O|p$ then by condition 2 of PCVax, $w_i^{x,p} \xrightarrow{prog} o_2$. Since $w_i^{x,p}$ is the last write to x by p to precede o_2 and since $o_1 \xrightarrow{prog} o_2$, also $o_1 \xrightarrow{prog} w_i^{x,p}$. Hence o_1 precedes $w_i^{x,p}$ in S_x and by construction of S_x, $w_i^{x,p}$ precedes o_2 in S_x. If $w_i^{x,p} \in O|q$ for some process $q \neq p$ then assume, to reach a contradiction, that $w_i^{x,p}$ precedes o_1 in S_x. Hence, $o_{1_{O|p}} \xrightarrow{view_p} w_i^{x,p} \xrightarrow{view_p} o_2 \xrightarrow{view_p} o_{1_{O_w}}$ and $w_i^{x,p} \in O_{invisible_p}$ and thus $w_i^{x,p} \notin S_x^p$, leading to a contradiction. Thus o_1 precedes o_2 in S_x.

Finally, consider the case when $o_1 \in O_r$ and $o_2 \in O_w$. Since $o_1 \xrightarrow{prog} o_2$, by condition 2 of PCVax, $o_1 \xrightarrow{view_p} o_{2_{O|p}}$ and by condition 4 of PCVax, $o_1 \xrightarrow{view_p} o_{2_{O|p}} \xrightarrow{view_p} o_{2_{O_w}}$. Let $o_1 \in R_i^{x,p}$. If $w_i^{x,p} \in O|p$ then $w_{i_{O|p}}^{x,p} \xrightarrow{view_p} o_1 \xrightarrow{view_p} o_{2_{O|p}} \xrightarrow{view_p} o_{2_{O_w}}$. Since $w_{i_{O|p}}^{x,p} \xrightarrow{view_p} o_{2_{O|p}}$ implies that $w_i^{x,p} \xrightarrow{prog} o_2$, $w_i^{x,p}$ precedes o_2 in S_x by condition 1 of PCVax. And thus by construction of S_x, o_1 precedes o_2 in S_x. If $w_i^{x,p} \in O|q$ for some process $q \neq p$, then clearly $w_i^{x,p}$ precedes o_2 in S_x and thus o_1 precedes o_2 in S_x. ■

It must be remembered that the description of the VAX 8800 [FKH87] is ambiguous. The formal definition would change if it was assumed that in the VAX 8800 processes do not block on read actions. Furthermore, one could assume that a write action is always slower than a read action, which could change the definition again.

6.3 Processor Consistency as implemented in the DASH Machine

Stanford's DASH machine implements a memory consistency model called Release Consistency. Release Consistency distinguishes between ordinary and special operations. The special operations are guaranteed to be processor consistent with respect to one another. This definition of processor consistency is yet another interpretation of Goodman's original definition. The Stanford team made several attempts to define processor consistency as implemented in the DASH and a group at Georgia Tech attempted to formalize one of these. The following subsections examine these attempts, formalizes each version, and shows the differences between them.

6.3.1 The first attempt to define Processor Consistency as implemented in the DASH

In 1990, Gharachorloo et al. at Stanford University described processor consistency as implemented in the DASH machine [GLL+90] and stated the following conditions:

1. "memory is kept coherent, that is, all writes to the same location are serialized in some order and are performed in that order with respect to any processor"

2. "before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be performed"

3. "before a STORE is allowed to perform with respect to any other processor, all previous accesses (LOADs and STOREs) must be performed"

Point 1 states that the memory is coherent, but it is not sufficient that all processors order all writes to the same location in the same order. Definition 4.2.2 of coherence also requires that program order is maintained when actions are to the same object. I will assume that the DASH machine does indeed accomplish this.

Gharachorloo et al. state that a load/read of object x is *performed with respect to* some processor p, when a store/write action by p to x does not affect the value returned by the load/read. Similarly a store/write of some value at object x *is performed with respect to* processor p when a load/read by p of object x returns the value stored/written or some subsequent value written to x [GLL+90]. These subsequent values are well defined because all processors view writes to the same location in the same order. An action is *performed* when it is performed with respect to all processors in the system. Let us now examine closely the conditions 2 and 3.

Since each process may view an action as being performed at different times, each processor p may have its own view of the memory, or its own copy of the shared memory, μ_p. Thus, similarly to the pipelined-RAM machine, each write action by some processor p has n corresponding μ_q-update$_p$ events, one for each processor q in the machine. Furthermore, if any processor views that the value of a read by some processor p is determined, all other reads preceding it in program order must also already be determined. If any processor views that an update corresponding to some write action w by some processor p has been implemented, then all processors must view that all updates corresponding to all other write actions preceding w in p's program have already been implemented and that all values of reads preceding w in p's program have already been determined. Gharachorloo et al. state that these conditions are minimum requirements for any machine implementing processor

consistency. Call such a machine $M_{Gharachorloo}$.

6.3.1.1 $M_{Gharachorloo}$ and the memory model it implements

In $M_{Gharachorloo}$ a read action $(read, x, \lambda, v)$ is implemented by the following ordered sequence of events:

1. a processor$_p$-read-request(x) to p's copy of the memory, and
2. a matching μ_p-reply(x, v) from p's copy of the memory to p.

A write action $(write, x, v, \lambda)$ is implemented by the ordered sequence of events:

1. processor$_p$-write-request(x, v) to μ_p, and
2. a matching μ_q-update$_p$(x, v) at each processor q's copy of the memory.

Furthermore, any execution E on $M_{Gharachorloo}$, which is a sequence of these 4 types of events, will also meet all the following constraints, for all processors p, q, r, s, locations x and values u, v:

1. A read returns the value observed in the memory. That is, for each μ_p-reply(x, v) event if μ_p-update(x, u) is the last preceding μ_p-update event to location x then $v = u$.
2. Each copy of the memory implements writes to the same location in the same order. That is, if μ_p-update$_q(x, v) \xrightarrow{E} \mu_p$-update$_r(x, u)$ then the matching μ_s events satisfy μ_s-update$_q(x, v) \xrightarrow{E} \mu_s$-update$_r(x, u)$.
3. Each copy of the memory, μ_p, maintains program order with respect to actions to the same location by p. That is, any two processor$_p$ events to the same location x are in the same order as the matching μ_p events.

4. Read actions are received by the memory in program order. That is, for any two processor$_p$-read-requests, α_1 and α_2, where r_1, r_2 are the matching μ_p-reply events respectively , if $\alpha_1 \xrightarrow{E} \alpha_2$, then $r_1 \xrightarrow{E} r_2$.

5. If an action o precedes a write in program order, then all events corresponding to o precede any update event corresponding to the write. That is, for any processor$_p$ event α and any processor$_p$-write-request α_w, and for all μ_q event(s) m matching α and for all μ_q events m_w matching α_w, if $\alpha \xrightarrow{E} \alpha_w$ then $m \xrightarrow{E} m_w$.

Note that constraints 4 and 5 do not require that this machine maintains program order if a read follows a write. Let the relaxed program order $(O, \overset{r-prog}{\longrightarrow})$ be defined as follows: $(o_1, o_2) \in (O, \overset{r-prog}{\longrightarrow})$ iff $o_1 \overset{prog}{\longrightarrow} o_2$ and either $o_1 \in O_r$ or $o_1, o_2 \in O_w$. Relaxed program order does not include the requirement that actions to the same object must remain in program order, as implied by constraint 3 of $M_{Gharachorloo}$. This is because constraint 3 only restricts the ordering at the memory of the processor that initiated the two actions to the same location, not at all copies of the memory.

The formal definition of the memory model implemented by $M_{Gharachorloo}$ requires another relation. Consider any computation of system (P, J) containing the set of actions O. Let $S = \{<_p\}$ be any set of sequences over the set of actions O such that it contains exactly one sequence for each $p \in P$. Then the relation $(O, \overset{\widehat{pcd(S)}}{\longmapsto})$ is defined as follows: $o_1 \overset{\widehat{pcd(S)}}{\longmapsto} o_2$ iff $\exists p \in P$ and $\exists x \in J$ such that one of the following conditions holds:

1. $o_1, o_2 \in O|p$ and $o_1 \overset{r-prog}{\longrightarrow} o_2$.

2. $o_1 <_p o_2$, and either $o_1 \in O|x$ and $o_2 \in (O_r|p)|x$ or $o_1, o_2 \in O_w|x$.

3. $o_1 \in (O_r|p)|x$, $o_2 \in O_w$ and $\exists o' \in O_w|x$ such that $o_1 <_p o' \stackrel{r-prog}{\longrightarrow} o_2$.

For any cycle $\beta = o_0, o_1, \ldots o_k$ in $(O, \stackrel{\widehat{pcd(S)}}{\longmapsto})$, let the *extended cycle* $\widehat{\beta}$ be β augmented as follows. For every i such that $o_i \stackrel{\widehat{pcd(S)}}{\longmapsto} o_{i+1}$ arises from part 3 of the definition of $\stackrel{\widehat{pcd(S)}}{\longmapsto}$ (that is, such that $o_i \in (O_r|p)|x$, $o_{i+1} \in O_w$ and $\exists o' \in O_w|x$ such that $o_i <_p o' \stackrel{r-prog}{\longrightarrow} o_{i+1}$), insert between o_i and o_{i+1} the o' action that is mentioned in part 3. Call any such o' an *inserted* action.

Definition 6.3.1 *Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCGharachorloo if for each processor $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ such that*

1. $(O|p \cup O_w, \stackrel{r-prog}{\longrightarrow}) \subseteq (O|p \cup O_w, <_{L_p})$, *and*
2. $\forall x \in J$
   (a) $((O|p)|x, \stackrel{prog}{\longrightarrow}) = ((O|p)|x, <_{L_p})$, *and*
   (b) $\forall q \in P$ $(O_w \cap O|x, <_{L_p}) = (O_w \cap O|x, <_{L_q})$, *and*
3. $(O, \stackrel{\widehat{pcd}(\{<_{L_q}|q \in P\})}{\longmapsto})$ *is cycle-free.*

6.3.1.2 $M_{Gharachorloo}$ implements exactly PCGharachorloo

In the following discussion, for any set of actions O, any object x, any process p, and any linearization $(O|p \cup O_w, <_{L_p})$, $o_1 <_p^x o_2$ iff $(o_1, o_2) \in ((O|p \cup O_w)|x, <_{L_p})$. The notation $(O, \stackrel{\widehat{pcd}}{\longmapsto})$ is used to abbreviate $(O, \stackrel{\widehat{pcd(S)}}{\longmapsto})$. Also, all results of calculations on indices are reduced modulo $k+1$. The following two lemmas are used to prove that $M_{Gharachorloo}$ implements exactly PCGharachorloo.

Lemma 6.3.2 *Let O be some set of actions, P some set of processes, J some set of objects, $\{(O|p \cup O_w, <_{L_p}) \mid p \in P\}$ a set of sequences satisfying conditions 1 and 2 of definition 6.3.1, and $\beta = o_0, o_1, \ldots, o_k$ any extended cycle of β', a cycle in $(O, \overset{\widehat{pcd}}{\longmapsto})$. If $\exists p, q \in P$, and $\exists x \in J$, and $\exists i$, $0 \leq i \leq k$ such that $o_i <_p^x o_{i+1} <_q^x o_{i+2}$, then either*

1. *there exists some shorter extended cycle of $(O, \overset{\widehat{pcd}}{\longmapsto})$, or*
2. *$o_i <_p^x o_{i+1} \overset{r-prog}{\longrightarrow} o_{i+2}$, or*
3. *$o_i \overset{r-prog}{\longrightarrow} o_{i+1} <_q^x o_{i+2}$.*

Proof: If o_{i+1} is a read, then $p = q$ since a read action only appears in the linearization of the reading process and thus $o_{i+1} <_p^x o_{i+2}$ implying that $o_i <_p^x o_{i+2}$. If $o_{i+2} \in O|p$ then $o_i <_p^x o_{i+1} \overset{r-prog}{\longrightarrow} o_{i+2}$, establishing the lemma. If $o_{i+2} \in O_w|q$ for some $q \neq p$, then o_{i+2} must be an inserted action by the definition of $(O, \overset{\widehat{pcd}}{\longmapsto})$, and $o_i <_p^x o_{i+1} <_p^x o_{i+2} \overset{r-prog}{\longrightarrow} o_{i+3}$. If $o_i \in O_w$ then removing o_{i+1} from β' still results in a cycle in $(O, \overset{\widehat{pcd}}{\longmapsto})$, and removing o_{i+1} from β results in a shorter extended cycle, establishing the lemma. If $o_i \in O_r|p$, then $o_i <_p^x o_{i+2} \overset{r-prog}{\longrightarrow} o_{i+3}$ implies that $o_i \overset{\widehat{pcd}}{\longmapsto} o_{i+3}$ by point 3 of $(O, \overset{\widehat{pcd}}{\longmapsto})$. Hence removing o_{i+1} from β also results in a shorter extended cycle.

Now consider the case when o_{i+1} is a write action by some process s. If $o_{i+2} \in O_w|s$ then $o_i <_p^x o_{i+1} \overset{r-prog}{\longrightarrow} o_{i+2}$, establishing the lemma. If $o_{i+2} \in O_w|t$ for some process $t \neq s$ then o_{i+1} cannot be an inserted action since $(o_{i+1}, o_{i+2}) \notin (O, \overset{r-prog}{\longrightarrow})$. Hence $o_i \notin O_r$. If o_i is an inserted action, then $o_i \overset{r-prog}{\longrightarrow} o_{i+1} <_q^x o_{i+2}$, establishing the lemma. Otherwise, since all processes p agree on the ordering of writes to the same location, o_{i+1} can be removed from β to form a shorter extended cycle. Finally, if $o_{i+2} \in O_r|q$ then o_{i+1} is not an inserted action. If $o_i \in O_r$, then $o_i \overset{r-prog}{\longrightarrow} o_{i+1} <_q^x o_{i+2}$

by the definition of $(O, \widehat{\overset{pcd}{\longmapsto}})$ and since o_{i+1} is not an inserted action. If $o_i \in O_w$ and if o_i is an inserted action then $o_i \overset{r-prog}{\longrightarrow} o_{i+1} <_q^x o_{i+2}$, otherwise, since $o_i <_p^x o_{i+1}$ implies that $o_i <_q^x o_{i+1}$ when both o_i and o_{i+1} are write actions, o_{i+1} can be removed from β to form a shorter extended cycle. ■

Lemma 6.3.3 *Let O be some set of actions, P some set of processes, J some set of objects, $\{(O|p \cup O_w, <_{L_p}) \mid p \in P\}$ a set of sequences satisfying conditions 1 and 2 of definition 6.3.1, and $\beta = o_0, o_1, \ldots, o_k$ any extended cycle of $(O, \widehat{\overset{pcd}{\longmapsto}})$. If, for some i, $0 \leq i \leq k$, $o_i \overset{r-prog}{\longrightarrow} o_{i+1} \overset{r-prog}{\longrightarrow} o_{i+2}$, then $o_0, o_1, \ldots, o_i, o_{i+2}, \ldots, o_k$ is also an extended cycle in $(O, \overset{pcd}{\longmapsto})$.*

Proof: First note that $o_i \overset{r-prog}{\longrightarrow} o_{i+1} \overset{r-prog}{\longrightarrow} o_{i+2}$ implies that $o_1 \overset{r-prog}{\longrightarrow} o_{i+2}$. If neither o_i nor o_{i+2} is an inserted action, then removing o_{i+1} from β results in a shorter extended cycle in $(O, \widehat{\overset{pcd}{\longmapsto}})$. If o_{i+1} is a read action and o_{i+2} is an inserted action, then $o_i \overset{r-prog}{\longrightarrow} o_{i+1} \overset{r-prog}{\longrightarrow} o_{i+2} \overset{r-prog}{\longrightarrow} o_{i+3}$ and removing o_{i+1} from β results in a shorter extended cycle in $(O, \widehat{\overset{pcd}{\longmapsto}})$. Finally consider the case when o_{i+1} is a write action and o_i is an inserted action. Since o_{i+2} must be a write action (otherwise $(o_{i+1}, o_{i+2}) \notin (O, \overset{r-prog}{\longrightarrow})$), removing o_{i+1} will still allow o_i to be one of the o' actions. Hence, removing o_{i+1} from β results in a shorter extended cycle of $(O, \widehat{\overset{pcd}{\longmapsto}})$. ■

These preceding two lemmas are now combined to conclude that, for any shortest extended cycle, the relation between consecutive actions in the cycle alternates between $\overset{r-prog}{\longrightarrow}$ and $<_p^x$. Say that an extended cycle *alternates* if $\forall i$ either $o_i <_p^x o_{i+1} \overset{r-prog}{\longrightarrow} o_{i+2}$ or $o_i \overset{r-prog}{\longrightarrow} o_{i+1} <_p^x o_{i+2}$, where o_i, o_{i+1} and o_{i+2} are consecutive actions in the extended cycle.

Corollary 6.3.4 *Let O be some set of actions, P some set of processes, J some set of objects, and $\{(O|p \cup O_w, <_{L_p}) \mid p \in P\}$ a set of sequences satisfying conditions 1 and 2 of definition 6.3.1. Then any shortest extended cycle of $(O, \stackrel{\widehat{pcd}}{\longmapsto})$ alternates.*

Proof: Let $\beta = o_0, o_1, \ldots, o_k$ be a shortest extended cycle of $(O, \stackrel{\widehat{pcd}}{\longmapsto})$. By the definition of $(O, \stackrel{\widehat{pcd}}{\longmapsto})$, each action in β is related to its successor and predecessor in β by either $\stackrel{r-prog}{\longrightarrow}$ or by $<_p^x$ for some process p and some object x. By lemma 6.3.2, there is some i such that $o_i \stackrel{r-prog}{\longrightarrow} o_{i+1}$. Without loss of generality, suppose $o_0 \stackrel{r-prog}{\longrightarrow} o_1$ (otherwise renumber the actions in the cycle). By lemma 6.3.3, $o_1 <_p^y o_2$ for some process p and some object x. Thus β alternates from action o_0 to action o_2. Suppose that alternation first fails between o_i and o_{i+1} for $2 \leq i \leq k-1$. As a consequence of lemma 6.3.3, it must be $o_{i-1} <_p^x o_i <_q^x o_{i+1}$ for some process p and q and some object x and $o_{i-2} \stackrel{r-prog}{\longrightarrow} o_{i-1}$. By lemma 6.3.2, either $o_{i-1} \stackrel{r-prog}{\longrightarrow} o_i <_q^x o_{i+1}$ or $o_{i-1} \stackrel{r-prog}{\longrightarrow} o_i <_q^x o_{i+1}$ holds. However, since $o_{i-2} \stackrel{r-prog}{\longrightarrow} o_{i-1}$, it is impossible that $o_{i-1} \stackrel{r-prog}{\longrightarrow} o_i$ by lemma 6.3.3. Hence, $o_{i-1} <_p^x o_i \stackrel{r-prog}{\longrightarrow} o_{i+1}$, contradicting the assumption of non-alternation around action o_i. Hence, β alternates from o_0 to o_k. Furthermore, by lemma 6.3.3, $o_k <_q^x o_0$ for some process q and some object x. Hence, by lemma 6.3.3, $o_{k-1} \stackrel{r-prog}{\longrightarrow} o_k$, otherwise, by lemma 6.3.2 $o_{k-1} <_r^x o_k <_q^x o_0$ could be replaced to create 2 consecutive $\stackrel{r-prog}{\longrightarrow}$ relations, contradicting that β is any of the shortest extended cycles in $(O, \stackrel{\widehat{pcd}}{\longmapsto})$ by lemma 6.3.3. Thus β alternates.

■

Theorem 6.3.5 *$M_{Gharachorloo}$ implements exactly PCGharachorloo.*

The theorem follows immediately from the following two lemmas.

Lemma 6.3.6 *Any computation arising from an execution on $M_{Gharachorloo}$ is PCGharachorloo.*

Proof: To show that any computation C arising from an execution E of $M_{Gharachorloo}$ is PCGharachorloo, I first construct, for each process p, a total order $(O|p \cup O_w, <_{L_p})$, using the subsequence of E containing the μ_p, such that each memory event represents the corresponding action. I then show that these total orders satisfy PCGharachorloo.

Let E be any execution on $M_{Gharchorloo}$ with resulting computation C and set of actions O. Let $P = \{\bar{p} \mid \text{processor } p \text{ is in } M_{Gharachorloo}\}$ be a set of processes such that the i^{th} action-invocation in each $\bar{p}$ corresponds to the i^{th} processor$_p$ event in E. The set of objects J is the set of all locations in $M_{Gharachorloo}$. For each processor p let E_p be the subsequence of E containing only all μ_p events and, for each $\bar{p}$, let $<_{L_p}$ be the sequence of all write actions and actions by $\bar{p}$ in O such that the i^{th} action in $<_{L_p}$ corresponds to the i^{th} event in E_p. Then each $<_{L_p}$ satisfies condition 1 of PCGharachorloo by constraints 4 and 5 of $M_{Gharachorloo}$. Condition 2a of PCGharachorloo is satisfied by each $<_{L_p}$ by constraint 3 of $M_{Gharachorloo}$ and condition 2b of PCGharachorloo is satisfied by each $<_{L_p}$ by constraint 2 of $M_{Gharachorloo}$. Furthermore, $<_{L_p}$ is linearization, by constraint 1 of $M_{Gharachorloo}$. Thus it remains to show that $(O, \widehat{\overset{pcd}{\longmapsto}})$ is cycle-free to establish that computation C is PCGharachorloo. Assume for a contradiction that it is not cycle-free. Let $\beta = o_1, o_2, o_3, \ldots, o_m$ be any of the shortest extended cycles in $(O, \widehat{\overset{pcd}{\longmapsto}})$. By corollary 6.3.4 (and by selectively choosing which action is named o_1 in β) the cycle is of the form $o_1 <^{x_1}_{\bar{p}_1} o_2 \overset{r-prog}{\longrightarrow} o_3 <^{x_3}_{\bar{p}_3} o_4 \overset{r-prog}{\longrightarrow} \ldots <^{x_{m-1}}_{\bar{p}_{m-1}} o_m \overset{r-prog}{\longrightarrow} o_1$ for some processes $\bar{p}_1, \bar{p}_3, \ldots, \bar{p}_{m-1} \in P$ and some objects $x_1, x_3, \ldots x_{m-1} \in J$.

For any i and j, let $\mu^j_{p_i}$ be the memory event at p_i's copy of the memory, corresponding to the action o_j. By construction of each $<_{L_{\hat{p}_i}}$, $o_i <^{x_i}_{\hat{p}_i} o_{i+1}$ implies that $\mu^i_{p_i} \xrightarrow{E} \mu^{i+1}_{p_i}$. Furthermore, if $o_i \xrightarrow{r-prog} o_{i+1}$ then, by constraints 4 and 5 of $M_{Gharachorloo}$, $\mu^i_{p_{i-1}} \xrightarrow{E} \mu^{i+1}_{p_{i+1}}$. Thus $o_1 <^{x_1}_{\hat{p}_1} o_2 \xrightarrow{r-prog} o_3 <^{x_3}_{\hat{p}_3} o_4 \xrightarrow{r-prog} \ldots <^{x_{m-1}}_{\hat{p}_{m-1}} o_m \xrightarrow{r-prog} o_1$ implies that $\mu^1_{p_1} \xrightarrow{E} \mu^2_{p_1} \xrightarrow{E} \mu^3_{p_3} \xrightarrow{E} \mu^4_{p_3} \xrightarrow{E} \ldots \xrightarrow{E} \mu^m_{p_{m-1}} \xrightarrow{E} \mu^1_{p_1}$. Thus there is a cycle in the execution E and this is clearly impossible. Hence computation C is PCGharachorloo.

■

Lemma 6.3.7 *Any PCGharachorloo computation is the result of some execution of* $M_{Gharachorloo}$.

Proof: To show that all PCGharachorloo computations arise from some execution on $M_{Gharachorloo}$, I first construct an execution by merging the linearizations and program orders of each process. I then show that this execution satisfies all the constraints of an execution on $M_{Gharachorloo}$.

Consider any PCGharachorloo computation C containing set of actions O of system (P, J). Let the processor that implements process $p \in P$ be named $\hat{p}$ and let the objects in J be the locations in $M_{Gharachorloo}$. For all $p \in P$, initially $L_p = (O|p \cup O_w, <_{L_p}) = o^p_1, o^p_2, \ldots o^p_{k_p}$ and $P_p = (O|p, \xrightarrow{prog}) = a^p_1, a^p_2, \ldots a^p_{l_p}$ and $i_p = 1$. Also, initially $i = 1$ and $E = \lambda$. Then the algorithm

while $(\forall k_p\ i <= k_p)$ **do**
  while $(\exists p \in P$ such that $o^p_i \in L_p)$ **do**
    for $m \leftarrow 1$ **to** n **do**
      ConsiderAdding(i, p_m)
    end for
  end while
  $i \leftarrow i + 1$
end while

constructs the sequence E containing all processor and $\mu_{\hat{p}}$ events corresponding to the actions in O where the procedure ConsiderAdding is shown in figure 6.3.1.2.

To show that E is an execution that could have occurred on $M_{Gharachorloo}$, assume first that this algorithm exhausts each L_p. Thus, all events corresponding to actions in O are in E since lines 2 to 9 add events corresponding to a read action (and possibly some extra processor events) and that lines 12 to 18 add events corresponding to a write action (and possibly some extra processor events). Furthermore, processor events are obviously appended to E in program order. Also the algorithm ensures that the processor events precede the matching memory events. Lines 1 and 10 ensure that $\mu_{\hat{p}}$ memory events to the same location are in the same order in E as the corresponding actions are ordered in $(O|p \cup O_w, <_{L_p})$. Since each linearization agrees on the ordering of writes to the same location, updates of the same location will also agree in E across all copies of the memory, and thus E satisfies constraint 2 of $M_{Gharachorloo}$. Since each linearization maintains program order with respect to actions to the same object, constraint 3 is also satisfied by E. Furthermore, since line 1 of the algorithm ensures that $\mu_{\hat{p}}$-reply events are appended to E in the same order as the corresponding reads appear in the linearization $(O|p \cup O_w, <_{L_p})$, by condition 1 of PCGharachorloo, E satisfies constraint 4. Similarly, the if-statement of line 10 ensures that if $o_1 \xrightarrow{prog} o_2$ and, for some process q, $o_1 \in O|q$ and $o_2 \in O_w|q$, then all memory events corresponding to o_1 are in E before any memory event corresponding to o_2 is appended to E, thus constraint 5 of $M_{Gharachorloo}$ is also satisfied by E. Finally, constraint 1 of $M_{Gharachorloo}$ is satisfied by E since all $\mu_{\hat{p}}$ memory events are in the same order as the corresponding actions are ordered in the linearization $(O|p \cup O_w, <_{L_p})$.

```
     procedure ConsiderAdding( index j, process p )
1        if (o_j^p ∈ L_p and ∃x ∈ J such that o_j^p ∈ (O_r|p)|x and
           w_x, the last write to x preceding o_j^p in (O|p ∪ O_w, <_{L_p}), is no longer in L_p
           and all events corresponding to r, the last read preceding o_j^p in program
           order, are in E ) then
2           if (the processor_p̂ event corresponding to o_j^p is not in E yet) then
3              repeat
4                 append to E the processor_p̂ event corresponding to a_{i_p}^p
5                 remove a_{i_p}^p from P_p
6                 i_p ← i_p + 1
7              until (a_{i_p-1}^p = o_j^p)
               end if
8           append to E the μ_p̂-reply event corresponding to o_j^p
9           remove o_j^p from L_p
            end if
10       if (o_j^p ∈ L_p and ∃x ∈ J and ∃q ∈ P such that o_j^p ∈ (O_w|q)|x and
           all events corresponding to o, the last action preceding o_j^p in program
           order, are all in E and o_x, the last action to x preceding o_j^p in
           (O|p ∪ O_w, <_{L_p}), is no longer in L_p) then
11          if (the processor event corresponding to o_j^p is not in E yet) then
12             repeat
13                append to E the processor_q̂ event corresponding to a_{i_q}^q
14                remove a_{i_q}^q from P_q
15                i_q ← i_q + 1
16             until (a_{i_q-1}^q = o_j^p)
               end if
17          append to E the μ_p̂-update_q̂ event corresponding to o_j^p
18          remove o_j^p from L_p
19       else
20          ConsiderAdding(j + 1, p)
            end if
```

Figure 6.2: *Part of construction of an execution E on $M_{Gharachorloo}$.*

So it remains to verify that each L_p is exhausted by the algorithm and that all events are added to E. Assume instead that at some point in the construction of E, for some m, the sequences $L_{p_1}, L_{p_2}, \ldots, L_{p_m}$ are not exhausted. For any $1 \leq i \leq m$, consider any action $o \in L_{p_i}$. Therefore, for some object $x \in J$ and some process p_j, p_k, where $1 \leq j, k \leq m$, one of the following 6 cases prevents o from being removed from L_{p_i} (and the μ_{p_i} memory event cannot be added to E).

1. $o \in O_r$ and $\exists o' \in O_r$ such that $o' \overset{r-prog}{\longrightarrow} o$ and $o' \in L_{p_i}$, or
2. $o \in O_r|x$ and $\exists o' \in O_w|x$ such that $o' <^x_{p_i} o$ and $o' \in L_{p_i}$, or
3. $o \in O_w|p_j$ and $\exists o' \in O_r|p_j$ such that $o' \overset{r-prog}{\longrightarrow} o$ and $o' \in L_{p_j}$, or
4. $o \in O_w|p_j$ and $\exists o' \in O_w|p_j$ such that $o' \overset{r-prog}{\longrightarrow} o$ and $o' \in L_{p_k}$, or
5. $o \in O_w|x$ and $\exists o' \in (O_r|p_i)|x$ such that $o' <^x_{p_i} o$ and $o' \in L_{p_i}$, or
6. $o \in O_w|x$ and $\exists o' \in O_w|x$ such that $o' <^x_{p_i} o$ and $o' \in L_{p_i}$.

Thus the actions in the sequences L_{p_1} to L_{p_m} form cycles, where each link is one of the above 6 cases. Notice that in the cases 1 to 4 and 6, $o' \overset{\widehat{pcd}}{\longmapsto} o$. Thus, if any cycle does not contain a case 5 link, this means that there is a cycle in $(O, \overset{\widehat{pcd}}{\longmapsto})$, which would imply that C is not PCGharachorloo. Hence, in every cycle, at least one of the links is due to case 5.

Examine any such cycle $\beta = o_0, o_1, \ldots o_k$. I will show that removing some actions from β results in an extended cycle of $(O, \overset{\widehat{pcd}}{\longmapsto})$. Choose any link in β that is due to case 5 and call the write action of the link w and let r be the read to the same object x of the link. Thus, for some process p_i, $r <^x_{p_i} w$ and $r \in L_{p_i}$. Let H_1 be the sequence

starting with the action w and followed by the sequence of actions following w in β, such that each action is from L_{p_i}, and such that each action is related to the next by $(O, \overset{\widehat{pcd}}{\longmapsto})$. Let H_2 start with the action o_{j+1}, where o_j is the last action of H_1, followed by the sequence of actions that follows o_{j+1} in β such that all actions of H_2 are from the same L_p sequence and each is related to its successor by $(O, \overset{\widehat{pcd}}{\longmapsto})$. Continue building these sequences, until all actions of β are in some sequence, resulting in the sequences H_1 to $H_{\hat{m}}$ for some $\hat{m} > 0$. Without loss of generality, let the sequences starting with a write that is the second action in a case 5 link be $H_1, H_2, \ldots, H_l$ for some $l \geq 1$ and let the remaining sequences be H_{l+1} to $H_{\hat{m}}$.

For any $1 \leq i \leq l$, let the first action of any sequence H_i be named $w_i \in O_w|x$ for some object x, and, for some process p_j, let $w_i \in L_{p_j}$. There must be some action $r_i \in (O_r|p_j)|x$ such that $r_i <^x_{p_j} w_i$ since w_i is part of a case 5 link. Note that r_i must be related to its predecessor in β by $(O, \overset{\widehat{pcd}}{\longmapsto})$. If H_i is of length 1, then w_i is the first action (the o' action) of a case 4 link, and thus $\exists w'_i \in O_w$ such that $w_i \overset{r-prog}{\longrightarrow} w'_i$. If $o_{\tilde{k}}$ is the action immediately following w'_i in β then $w'_i \overset{\widehat{pcd}}{\longmapsto} o_{\tilde{k}}$ since this cannot be a case 5 link. Hence, $r_i <^x_{p_i} w_i \overset{r-prog}{\longrightarrow} w'_i \overset{\widehat{pcd}}{\longmapsto} o_{\tilde{k}}$. By part 3 of $(O, \overset{\widehat{pcd}}{\longmapsto})$, $r_i \overset{\widehat{pcd}}{\longmapsto} w'_i$ and thus $o_k \overset{\widehat{pcd}}{\longmapsto} r_i \overset{\widehat{pcd}}{\longmapsto} w'_i \overset{\widehat{pcd}}{\longmapsto} o_{\tilde{k}}$ and w_i is an inserted action in an extended cycle. If w_i is immediately followed by some read action r'_i, then $w_i <^x_{p_j} r'_i$ and $r_i \in (O_r|p_j)|x$. Since r_i and r'_i are by the same process, $r_i \overset{r-prog}{\longrightarrow} r'_i$, and thus $r_i \overset{\widehat{pcd}}{\longmapsto} r'_i$. Remove all these w_i write actions from β to form the cycle β'. If the action immediately following w_i is some write action w'_i and if $w_i \overset{r-prog}{\longrightarrow} w'_i$, then $r_i \overset{\widehat{pcd}}{\longmapsto} w'_i$. Hence, w_i is an inserted action. (Note that w'_i must be related to its successor in β and β' by $(O, \overset{\widehat{pcd}}{\longmapsto})$). If w_i is immediately followed by some write action w'_i and $w_i <^x_{p_j} w'_i$, then also, $r_i <^x_{p_j} w'_i$. Such a write must be followed by a sequence of zero or more writes, all

related only because they are to the same object, but must finally be followed by either a read or another write in the relaxed program order, since the sequence is of finite length and can only be ended through a link using the relaxed program order. Thus $r_i <^x_{p_j} w_i <^x_{p_j} w_i' <^x_{p_j} w_i^1 <^x_{p_j} w_i^2 <^x_{p_j} \ldots <^x_{p_j} w_i^h$, that is, $r_i <^x_{p_j} w_i^h$ for some $h \geq 0$ and some $w_i^1, w_i^2, \ldots w_i^h \in O_w|x$. If w_i^h is followed by a read, r_i', in H_i, then, for all such cases, remove the actions $w_i, w_i', w_i^1, \ldots w_i^h$ from β' to form the cycle $\widehat{\beta}$. Note that $r_i \overset{\widehat{pcd}}{\longmapsto} r_i'$. If w_i^h is followed in β by some $w_i^{h+1} \in O_w$ and $w_i^h \overset{r-prog}{\longrightarrow} w_i^{h+1}$, then, in all such cases, remove all actions $w_i, w_i', w_i^1, \ldots w_i^{h-1}$ from $\widehat{\beta}$ to form the cycle $\bar{\beta}$. Note that $r_i <_{p_i} w_i^h \overset{r-prog}{\longrightarrow} w_i^{h+1}$ and thus $r_i \overset{\widehat{pcd}}{\longmapsto} w_i^{h+1}$. But $\bar{\beta}$ is an extended cycle in $(O, \overset{\widehat{pcd}}{\longmapsto})$ and thus $(O, \overset{\widehat{pcd}}{\longmapsto})$ contains a cycle. ■

6.3.1.3 Comparing PCGharachorloo with other memory models

Computation 6 on page 34 was shown to be a coherent computation, but is not PCGharachorloo since it is not possible to form a linearization for process q that maintains relaxed program order. Since PCGharachorloo clearly implies coherence, PCGharachorloo is strictly stronger than coherence.

Computation 2 on page 17, which is not SC, is PCGharachorloo since $<_{L_p} = w_p(x)0\ w_q(y)0\ w_p(x)1\ r_p(y)0\ w_q(y)1$ and $<_{L_q} = w_p(x)0\ w_q(y)0\ w_q(y)1\ r_q(x)0\ w_p(x)1$ are linearizations that maintain program order, agree on the ordering of writes to the same object and $(O, \overset{\widehat{pcd}}{\longmapsto})$ does not contain a cycle. Clearly, any SC computation will also be PCGharachorloo, hence SC is strictly stronger than PCGharachorloo.

Computation 10 $\begin{cases} p: w(y)2\ w(x)0\ w(x)1\ w(z)0\ r(y)0\ w(y)3\ r(z)1 \\ q: w(x)2\ w(y)0\ w(y)1\ w(z)1\ r(x)0\ r(y)3\ r(z)1 \end{cases}$

Computation 10 is PCGharachorloo, but is not PCG, nor P-RAM-A (and thus not P-RAM-R nor P-RAM-W). The sequences

$<_{L_p} = w_p(y)2\ w_q(x)2\ w_p(x)0\ w_p(x)1\ w_p(z)0\ w_q(y)0\ r_p(y)0\ w_q(y)1\ w_p(y)3\ w_q(z)1$ $r_p(z)1$ and $<_{L_q} = w_p(y)2\ w_q(x)2\ w_p(x)0\ r_q(x)0\ w_p(x)1\ w_p(z)0\ w_q(y)0\ w_q(y)1\ w_p(y)3$ $r_q(y)3\ w_q(z)1\ r_q(z)1$ are linearizations that satisfy the conditions of PCGharachorloo. Each maintains relaxed program order, and agrees on the ordering of writes to the same object. To show that $(O, \widehat{\overset{pcd}{\longmapsto}})$ does not contain a cycle, consider figures 6.3 and 6.4.

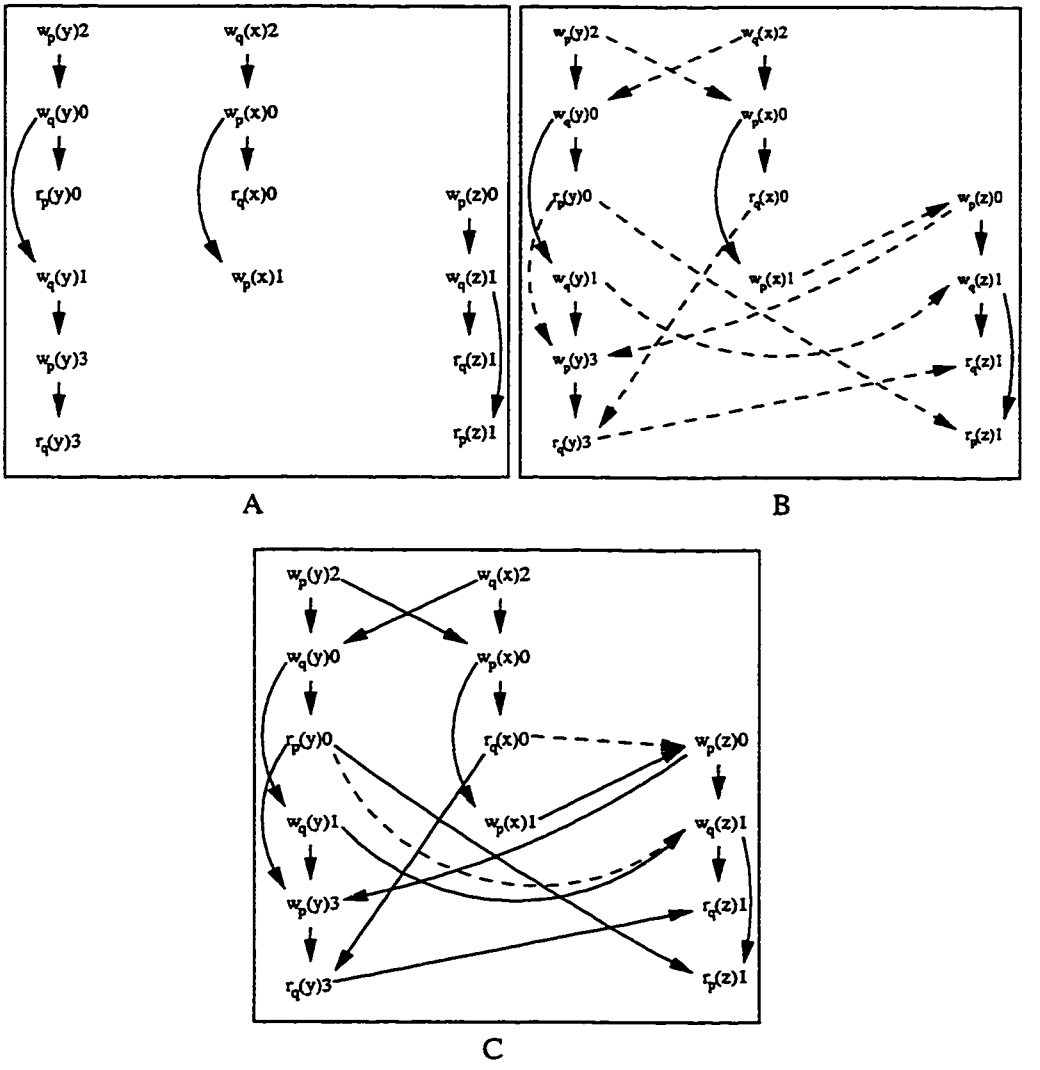


Figure 6.3: *The* $(O, \widehat{\overset{pcd}{\longmapsto}})$ *relation in Computation 10*

In figure 6.3 part A there is an arrow between two actions if they are related by part 2 of the definition of $(O, \widehat{\overset{pcd}{\longmapsto}})$ such that arrows due to ransitive closure of existing arrows are ommitted. In part B of figure 6.3, the added dashed arrows connect two actions that are related by part 1 of the definition of $(O, \widehat{\overset{pcd}{\longmapsto}})$, unless there is some path between the two actions in the graph already. In part C of figure 6.3 arrows due to part 3 of $(O, \widehat{\overset{pcd}{\longmapsto}})$ that are needed to show the final graph for the relation $(O, \widehat{\overset{pcd}{\longmapsto}})$ are added. Figure 6.4 shows a picture of the relation $(O, \widehat{\overset{pcd}{\longmapsto}})$ that is based on the linearizations $<_{L_p}$ and $<_{L_q}$. In the diagram, arrows from part C of figure 6.3 are removed if they are redundant. That is, an arrow between two actions is removed if it can be infered by transitive closure. In the remainder of this chapter, only the final graph of a relations is shown for other example computations.

Figure 6.4 shows that $(O, \widehat{\overset{pcd}{\longmapsto}})$ does not contain a cycle. However, it is impossible

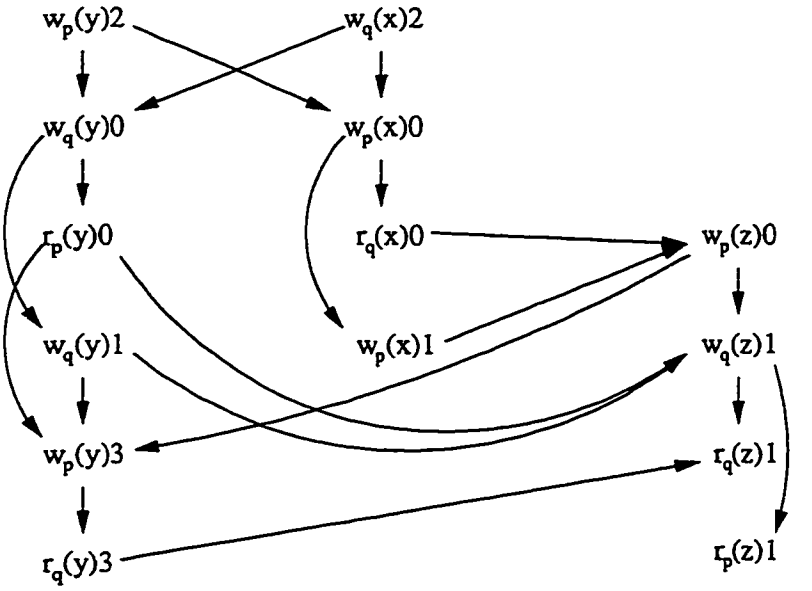


Figure 6.4: *The* $(O, \widehat{\overset{pcd}{\longmapsto}})$ *relation in Computation 10*

to build linearizations that extend program order for process q. Process q requires

that p's $w(x)1$ appears after q's read of x and that p's $w(y)3$ appears before q's read of y. Hence, $w(z)0$ must appear between $w(z)1$ and $r(z)1$ according to q's view. Hence, Computation 10 is neither PCG, nor P-RAM-A since they require that each process views all actions in program order.

Condition 2 of PCGharachorloo is exactly the definition of coherence (definition 4.2.2), thus any computation that is PCGharachorloo is also coherent. But computation 6 on page 34, which is coherent, is not PCGharachorloo since it is not possible to construct a linearization for q that maintains the relaxed program order. Hence, PCGharachorloo is stronger than coherence.

Computation 3 on page 33 is PCG and P-RAM-A but is not PCGharachorloo. In any linearization for p, $w(y)1$ must appear before $r(x)1$ and in any linearization for q $w(y)1$ must appear before $r(y)1$. Since $w(y)1 <_p^y r(y)1 \xrightarrow{r-prog} w(x)1 <_q^x r(x)1 \xrightarrow{r-prog} w(y)1$ implies that $w(y)1 \xmapsto{\widehat{pcd}} r(y)1 \xmapsto{\widehat{pcd}} w(x)1 \xmapsto{\widehat{pcd}} r(x)1 \xmapsto{\widehat{pcd}} w(y)1$, there are no linearizations that satisfy PCGharachorloo. Hence PCGharachorloo and PCG are incomparable and PCGharachorloo and P-RAM-A are incomparable.

Computation 5 on page 34 is P-RAM-W and P-RAM-R, but is not coherent and hence, not PCGharachorloo. Thus PCGharachorloo is incomparable with P-RAM-W and P-RAM-R.

Finally, Computation 8 on page 55 is not PCVax, but is PCGharachorloo since $<_{L_p} = w_r(x)0\ w_s(y)0\ w_p(x)1\ w_q(y)1$, $<_{L_q} = w_r(x)0\ w_s(y)0\ w_p(x)1\ w_q(y)1$, $<_{L_r} = w_r(x)0\ w_s(y)0\ w_p(x)1\ r_r(x)1\ r_r(y)0\ w_q(y)1$ and $<_{L_s} = w_r(x)0\ w_s(y)0\ w_q(y)1\ r_s(y)1\ r_s(x)0\ w_q(y)1$ are linearizations that satisfy PCGharachorloo. Each maintains the relaxed program order, maintains program order on a per location basis and agrees on the ordering of writes to the same location. Figure 6.5 shows the $(O, \xmapsto{\widehat{pcd}})$

relation, which clearly does not contain a cycle. Computation 9 is PCVax, but not

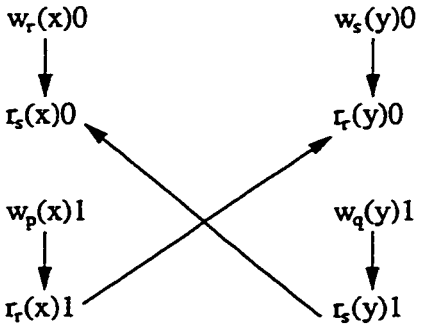


Figure 6.5: *The* $(O, \widehat{\overset{pcd}{\longmapsto}})$ *relation of Computation 8*

PCGharachorloo since it is not possible to build a linearization for p, containing all actions by p and all write actions by q, that maintains the relaxed program order. Hence, PCVax and PCGharachorloo are incomparable.

6.3.2 Two attempts to formalize at Georgia Tech

Besides developing the formal definition PCG, Ahamad et al. at Georgia Tech also developed a formal definition of processor consistency as implemented in the DASH machine based on Gharachorloo et al's 1990 paper [ABJ+93, GLL+90]. This same group of people modifies this formal definition somewhat in a later paper [KNA93]. Neither of these formal definitions is equivalent to PCGharachorloo.

6.3.2.1 PCAhamad and PCKohli

To define processor consistency, both papers use a new relation, the *partial program order* [KNA93] ($\overset{ppo}{\longrightarrow}$, also called $\overset{d}{\longrightarrow}$ in the earlier paper [ABJ+93]) which is a relaxation on the program order similar to the relaxed program order and defined as follows. For any computation C of system (P, J) containing set of actions O and

$\forall o_1, o_2 \in O$, $o_1 \xrightarrow{ppo} o_2$ iff $o_1 \xrightarrow{proc} o_2$, and either

1. o_1, o_2 are both read or both write actions, or
2. o_1 is a read and o_2 is a write action, or
3. o_1, o_2 are actions to the same object, or
4. $\exists o'$ such that $o_1 \xrightarrow{ppo} o' \xrightarrow{ppo} o_2$.

Both papers also assume that for each object x in an execution, each update of x has a unique value. This is used when defining the *semi-causality* relation. Consider any computation of system (P, J) containing the set of actions O. Let $S = \{<_p\}$ be any set of sequences over the set of actions O including exactly one sequence for each $p \in P$. Then the relation $(O, \overset{semi(S)}{\longmapsto})$ is defined as follows: $o_1 \overset{semi(S)}{\longmapsto} o_2$ iff $\exists q, r \in P$ and $\exists x \in J$ such that

1. $o_1, o_2 \in O|q$ and $o_1 \xrightarrow{ppo} o_2$, or
2. $o_1 \in O_w|r$, $o_2 \in (O_r|q)|x$ with return value v, and $\exists o' \in (O_w|r)|x$ with value v such that $o_1 \xrightarrow{ppo} o'$, or
3. $o_1 \in (O_r|q)|x$, $o_2 \in O_w|r$, and $\exists o' \in (O_w|r)|x$ such that $o_1 <_q o' \xrightarrow{ppo} o_2$, or
4. $\exists o' \in O$ such that $o_1 \overset{semi(S)}{\longmapsto} o' \overset{semi(S)}{\longmapsto} o_w$.

Even though Kohli et al.'s [KNA93] paper is the later version of this group's interpretation of processor consistency as implemented in the DASH machine, it is stated first here, because it is actually weaker than the earlier similar definition.

Definition 6.3.8 *Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCKohli if for each processor $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ such that*

1. $\forall x \in J$ *and* $\forall q \in P$ $(O_w \cap O|x, <_{L_p}) = (O_w \cap O|x, <_{L_q})$, *and*
2. $(O|p \cup O_w, \overset{semi(\{<_{L_q}|q \in P\})}{\longmapsto}) \subseteq (O|p \cup O_w, <_{L_p})$.

In the earlier paper [ABJ+93] another restriction to processor consistency is added, which makes the older definition stronger. An additional relation is needed for this alternate definition, namely the *weak-order* relation ($\overset{wo}{\longmapsto}$) which uses yet another relation: the *writes-before order* ($\overset{wb}{\longmapsto}$). The writes-before order relates a read action to the write action that is observed by the read. More formally, for any computation C of system (P, J) containing set of actions O, where each write in O is unique, and $\forall w, r \in O$, $w \overset{wb}{\longmapsto} r$ iff $\exists x \in J$ such that $w \in O_w|x$ with value v and $r \in O_r|x$ with return value v. The weak-order relation is the transitive closure[1] of the union of the partial program order and the writes-before order, or $\overset{wo}{\longmapsto} = (\overset{ppo}{\longrightarrow} \cup \overset{wb}{\longmapsto})^+$.

The definition of processor consistency as used by Ahamad et al. [ABJ+93], becomes the following, when translated to the framework.

Definition 6.3.9 *Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCAhamad if*

1. $\forall x \in J$ *there is some linearization* $(O|x, <_{L_x})$ *such that* $(O|x, \overset{proc}{\longrightarrow}) \subseteq (O|x, <_{L_x})$, *and*
2. $\forall p \in P$ *there is some linearization* $(O|p \cup O_w, <_{L_p})$ *such that*

[1] For any relation $R \subset S \times S$, the transitive closure of R is $\{(s, s') | \exists k \geq 1, \exists s_1, s_2, \ldots, s_k \in S$ *st* $sRs_1Rs_2R \ldots Rs_kRs'\}$

(a) $(O|p \cup O_w, \overset{semi(\{<_{L_q}|q \in P\})}{\longmapsto}) \subseteq (O|p \cup O_w, <_{L_p})$, *and*

(b) $\forall x \in J(O_w|x, <_{L_p}) = (O_w|x, <_{L_x})$, *and*

3. $(O, \overset{wo}{\longmapsto})$ *is cycle-free.*

In the following discussion, the notation $\overset{semi}{\longmapsto}$ is used to abbreviate $\overset{semi(S)}{\longmapsto}$, when the set S is understood.

6.3.2.2 Comparing PCAhamad and PCKohli

Claim 6.3.10 *A computation satisfies definition 6.3.8 iff it satisfies conditions 1 and 2 of definition 6.3.9.*

Proof: Let C be any computation of some system (P, J) containing actions O that satisfies definition 6.3.8 and choose any set of process linearizations, $(O|p \cup O_w, <_{L_p})$, one for each process $p \in P$, that satisfy definition 6.3.8. Use the method shown in the proof of claim 4.2.3 to find the object linearizations $(O|x, <_{L_x})$, one for each object $x \in J$, that satisfy definition 4.2.1 of coherence. Hence, condition 1 of PCAhamad is satisfied by computation C. Each $(O|p \cup O_w, <_{L_p})$ clearly also satisfies 2a of PCAhamad, and, by the construction of each $(O|x, <_{L_x})$, each process linearization also satisfies 2b of PCAhamad. Hence, C satisfies conditions 1 and 2 of definition 6.3.9.

Now choose any computation C of some system (P, J) containing actions O that satisfies conditions 1 and 2 of definition 6.3.9 and choose any set of object linearizations $(O|x, <_{L_x})$, one for each $x \in J$, and any set of process linearizations $(O|p \cup O_w)$, one for each $p \in P$, that satisfy condition 1 and 2 of definition 6.3.9. Each process linearization clearly satisfies the second condition of PCKohli. Since each process

linearization agrees on the ordering of writes to any object x, namely in the same order as they appear in $(O|x, <_{L_x})$, condition 1 of PCKohli is also satisfied by the process linearizations. Hence C satisfies definition 6.3.8. ■

Computation 3 on page 33 is not PCAhamad, since $w_q(y)1 \overset{wb}{\longmapsto} r_p(y)1 \overset{ppo}{\longrightarrow} w_p(x)1 \overset{wb}{\longmapsto} r_q(x)1 \overset{ppo}{\longrightarrow} w_q(y)1$ and hence $(O, \overset{wo}{\longmapsto})$ contains a cycle. But Computation 3 is PCKohli since the sequences $<_{L_p} = w_p(y)0\ w_q(x)0\ w_q(y)1\ r_p(y)1\ w_p(x)1$ and $<_{L_q} = w_q(x)0\ w_p(y)0\ w_p(x)1\ r_q(x)1\ w_q(y)1$ are linearizations that maintain program order and agree on the ordering of writes to the same location. Furthermore in Computation 3, the relation $(O, \overset{semi(\{<_{L_p}, <_{L_q}\})}{\longmapsto}) = (O, \overset{ppo}{\longrightarrow})$. Thus, PCAhamad is strictly stronger than PCKohli.

6.3.2.3 Comparing PCAhamad and PCKohli with other memory models

Any computation that is PCKohli is also coherent since each process must agree on the ordering of writes to the same object, and program order on a per object basis must be maintained. Thus, any linearization that satisfies definition 6.3.8 will also satisfy definition 4.2.2 of coherence on page 17. Computation 6 on page 34 was shown to be coherent but is not PCKohli since it is not possible to construct a linearization containing q's read actions and p's write actions that maintains partial program order. Hence, both PCAhamad and PCKohli are strictly stronger memory consistency models than coherence.

Computation 5 on page 34, which is P-RAM-W (and thus P-RAM-R and P-RAM-A), is not PCKohli (and hence not PCAhamad) since it is not coherent. Computation 10 on page 73, which is neither P-RAM-A (and thus not P-RAM-W nor P-RAM-R) nor PCG, is PCAhamad (and hence PCKohli). The sequences $<_{L_p} = w_p(y)2\ w_q(x)2$

$w_p(x)0\ w_p(x)1\ w_p(z)0\ w_q(y)0\ r_p(y)0\ w_q(y)1\ w_p(y)3\ w_q(z)1\ r_p(z)1$ and $<_{L_q} = w_p(y)2$ $w_q(x)2\ w_p(x)0\ r_q(x)0\ w_p(x)1\ w_p(z)0\ w_q(y)0\ w_q(y)1\ w_p(y)3\ r_q(y)3\ w_q(z)1\ r_q(z)1$ are linearizations that satisfy the conditions of PCAhamad. The two linearizations agree on the ordering of writes to the same location. Part D of figure 6.6 shows the relation $(O, \overset{semi}{\longmapsto})$ pictorially. Part A of figure 6.6 shows the partial program order, which is the first part of $(O, \overset{semi}{\longmapsto})$. Part B adds arrows for actions which are related by part 2 of $(O, \overset{semi}{\longmapsto})$, unless there already is some path between the two actions, and part C adds arrows for actions which are related by part 3 of $(O, \overset{semi}{\longmapsto})$. Part D has arrows between actions removed, if they can be derived from transitive closure. Both linearizations, $<_{L_p}$ and $<_{L_q}$, maintain the semi-causality relation as shown in figure 6.6, and $(O, \overset{wo}{\longmapsto})$ is cycle-free. Hence, Computation 10 is PCAhamad. Thus, both PCKohli and PCAhamad are incomparable with P-RAM-W, P-RAM-R and P-RAM-A.

Computation 11
$$\begin{cases} p : w(x)0\ w(x)1\ w(y)1 \\ q : r(y)1\ r(z)0 \\ r : w(z)0\ w(z)1\ w(v)1 \\ s : r(v)1\ r(x)0 \end{cases}$$

Computation 11 [ABJ+93] is PCG, since the sequences $<_{L_p} = w_p(x)0\ w_p(x)1$ $w_p(y)1\ w_r(z)0\ w_r(z)1\ w_r(v)1$, $<_{L_q} = w_p(x)0\ w_p(x)1\ w_p(y)1\ r_q(y)1\ w_r(z)0\ r_q(z)0$ $w_r(z)1\ w_r(v)1$, $<_{L_r} = <_{L_p}$ and $<_{L_s} = w_r(z)0\ w_r(z)1\ w_r(v)1\ r_s(v)1\ w_p(x)0\ r_s(x)0$ $w_p(x)1\ w_p(y)1$ are linearizations that agree on the ordering of writes to the same object and extend program order. But any process linearization for s that contains all the write actions in the computation and the two read actions by s and maintains the partial program order must order the action $r(x)0$ between the writes $w(x)0$ and $w(x)1$ by p. And, since $w(x)1 \overset{ppo}{\longrightarrow} w(y)1$, $r(x)0 <_{L_s} w(x)1$ implies that

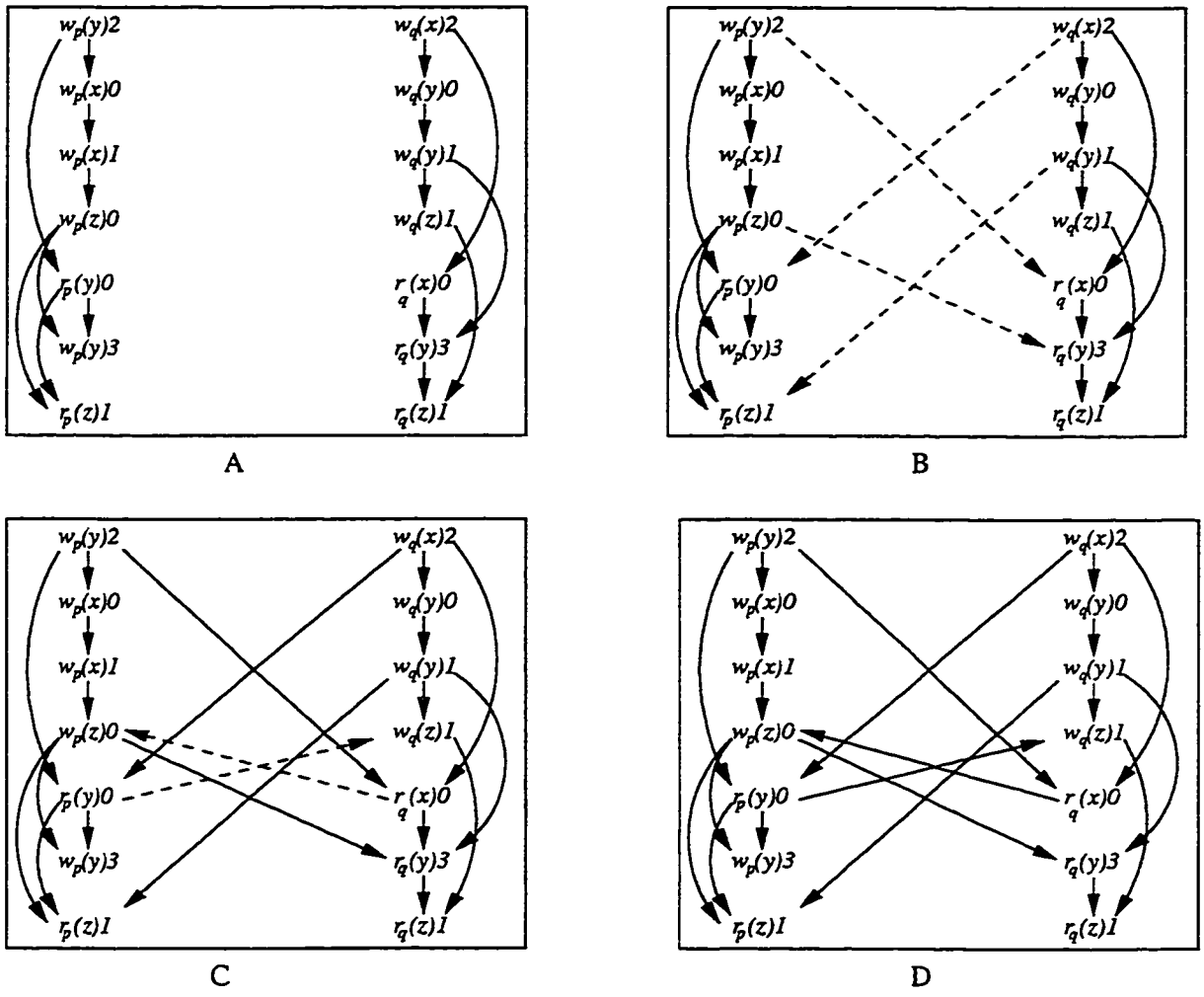


Figure 6.6: *The* $(O, \overset{semi}{\longmapsto})$ *relation in Computation 10*

$r(x)0 \overset{semi}{\longmapsto} w(y)1$. Similarly, the linearization for q necessarily orders the action $r(z)0$ between the two writes $w(z)0$ and $w(z)1$ by r, and, since $w(z)1 \overset{ppo}{\longrightarrow} w(v)1$, $r(z)0 \overset{semi}{\longmapsto} w(v)1$. Also $r(y)1 \overset{ppo}{\longrightarrow} r(z)0$ implies that $r(y)1 \overset{semi}{\longmapsto} r(z)0$. Thus, $r(x)0 \overset{semi}{\longmapsto} w(y)1 \overset{semi}{\longmapsto} r(y)1 \overset{semi}{\longmapsto} r(z)0 \overset{semi}{\longmapsto} w(v)1$. Hence, any linearizations satisfying PCKohli must order the action $r(x)0$ before the action $w(v)1$ in the process linearization for s, which is not possible if the partial program order must be maintained. Hence Compu-

tation 11 is neither PCkohli, nor PCAhamad. Thus PCAhamad and PCKohli are incomparable with PCG.

Computation 8 on page 55, which is not PCVax, is PCAhamad (and thus PCKohli). Consider the following linearizations $<_{L_p} = w_r(x)0\ w_s(y)0\ w_p(x)1\ w_q(y)1$, $<_{L_q} = w_r(x)0\ w_s(y)0\ w_p(x)1\ w_q(y)1$, $<_{L_r} = w_r(x)0\ w_s(y)0\ w_p(x)1\ r_r(x)1\ r_r(y)0\ w_q(y)1$ and $<_{L_s} = w_r(x)0\ w_s(y)0\ w_q(y)1\ r_s(y)1\ r_s(x)0\ w_q(x)1$ and note that $(O, \overset{semi}{\longmapsto}) = (O, \overset{ppo}{\longrightarrow})$ and that $(O, \overset{wo}{\longmapsto})$ does not contain a cycle. The linearizations agree on the ordering of writes to the same object and maintain the partial program order and thus Computation 8 is PCAhamad. Computation 9 on page 56, which is PCVax, is not PCKohli (and thus not PCAhamad) since it is not possible to build a linearization for process p that orders all of p's actions and all of q's actions and maintains partial program order. Hence, PCKohli and PCAhamad are incomparable with PCVax.

Computation 12
$$\begin{cases} p : r(x)2\ w(y)4 \\ q : w(y)3\ w(x)2 \\ r : r(y)4\ w(x)1 \\ s : r(x)1\ w(x)2 \end{cases}$$

In Computation 12 the linearizations $<_{L_p} = w_q(y)3\ w_r(x)1\ w_q(x)2\ r_p(x)2\ w_p(y)4$, $<_{L_q} = w_q(y)3\ w_r(x)1\ w_q(x)2\ w_p(y)4$, $<_{L_r} = w_q(y)3\ w_p(y)4\ r_r(y)4\ w_r(x)1\ w_q(x)2$, and $<_{L_s} = w_q(y)3\ w_p(y)4\ w_r(x)1\ r_s(x)1\ w_q(x)2\ r_s(x)2$ agree on the ordering of writes to the same object, and maintain partial program order. Furthermore, the semi-causality relation is equivalent to the partial program order and the weak-order relation is cycle-free. Thus Computation 12 is PCAhamad and PCKohli. But Computation 12 is not PCGharachorloo. Any linearization for process p must order $w_q(x)2$ before $r_p(x)2$, and hence, necessarily, $w_q(x)2 \overset{\widehat{pcd}}{\longmapsto} r_p(x)2$. Similarly, process r

must order the action $w_p(y)4$ before its own action $r_r(y)4$ in its linearization, and hence $w_p(y)4 \overset{\widehat{pcd}}{\longmapsto} r_r(y)4$. Since process s requires that all processes view $w_r(x)1$ before $w_q(x)2$, also $w_r(x)1 \overset{\widehat{pcd}}{\longmapsto} w_q(x)2$. That is, $w_r(x)1 \overset{\widehat{pcd}}{\longmapsto} w_q(x)2 \overset{\widehat{pcd}}{\longmapsto} r_p(x)2 \overset{r-prog}{\longrightarrow} w_p(y)4 \overset{\widehat{pcd}}{\longmapsto} r_r(y)4 \overset{r-prog}{\longrightarrow} w_r(x)1$. Hence, any linearizations would cause a cycle to appear in $(O, \overset{\widehat{pcd}}{\longmapsto})$ implying that Computation 12 is not PCGharachorloo.

Computation 13
$$\begin{cases} p : w(x)0\ w(z)0\ r(z)1\ r(x)0\ r(x)1 \\ q : w(x)1\ r(x)1\ r(y)1 \\ r : w(y)1\ w(y)2\ w(z)1 \end{cases}$$

Finally, Computation 13 is PCGharachorloo but is not PCKohli, and thus is not PCAhamad. Any linearization for process q that satisfies PCKohli must order $r(y)1$ by q between the writes $w(y)1$ and $w(y)2$ by r. The orderings $w(y)2 \overset{ppo}{\longrightarrow} w(z)1$ and $r(y)1 <_{L_q} w(y)2$ imply that $r(y)1 \overset{semi}{\longmapsto} w(z)1$, and $w(x)1 \overset{ppo}{\longrightarrow} r(x)1 \overset{ppo}{\longrightarrow} r(y)1$ implies that $w(x)1 \overset{semi}{\longmapsto} r(y)1$. Furthermore, the linearization of process p must order the write $w(x)1$ by q between the reads $r(x)0$ and $r(x)1$ by p. Hence, since the linearization for p must maintain the partial program order, any linearization for p satisfying PCAhamad must order the action $r(z)1$ before the action $r(x)0$. The action $r(x)0$ must be ordered before the action $w(x)1$ in order for the sequence to be valid. Finally, to satisfy the relation $(O, \overset{semi}{\longmapsto})$, the linearization for p must order the action $w(x)1$ before $w(z)1$. Hence, $r(z)1$ must be ordered before $w(z)1$ which is clearly invalid. Hence, Computation 13 is not PCAhamad nor PCKohli.

Now consider the following linearizations: $<_{L_p} = w_p(x)0\ w_p(z)0\ w_r(y)1\ w_r(y)2\ w_r(z)1\ r_p(z)1\ r_p(x)0\ w_q(x)1\ r_p(x)1$, and $<_{L_q} = w_p(x)0\ w_p(z)0\ w_q(x)1\ r_q(x)1\ w_r(y)1\ r_q(y)1\ w_r(y)2\ w_q(z)1$, and $<_{L_r} = w_p(x)0\ w_p(z)0\ w_q(x)1\ w_r(y)1\ w_r(y)2\ w_q(z)1$. Each maintains the relaxed program order, maintains program order on a per object basis, and agrees on the ordering of writes to the same location. Figure 6.7 shows the

relation $(O, \overset{\widehat{pcd}}{\longmapsto})$ and does not contain a cycle. Thus Computation 13 is PCGharachorloo.

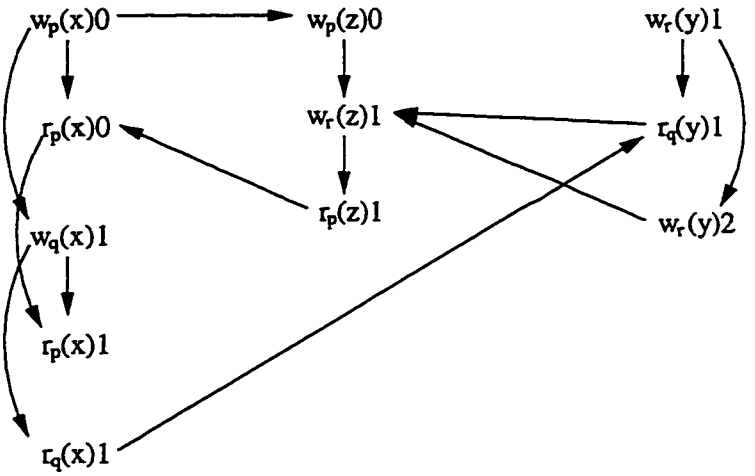


Figure 6.7: *The* $(O, \overset{\widehat{pcd}}{\longmapsto})$ *relation in Computation 13*

Hence, PCKohli and PCAhamad are incomparable with PCGharachorloo.

6.3.3 A revision of Stanford's original attempt to define Processor Consistency as implemented in the DASH machine

In 1990, Gharachorloo et al. intended to capture processor consistency as implemented in the DASH machine [GLL+90]. However, in a revision to this paper [GGH93], they correct an error in their original specifications. Computation 14 [GGH93] is an example of a computation that is allowed in the DASH system, but is not permitted according to the formal definition PCGharachorloo .

Computation 14 $\begin{cases} p : w(y)0\ w(y)1\ w(x)1\ r(x)1\ r(z)0 \\ q : w(z)0\ w(z)1\ w(x)2\ r(x)2\ r(y)0 \end{cases}$

Process p must view the action $w_q(z)1$ after its read action $r(z)0$ and $w(y)1 \overset{r-prog}{\longrightarrow} w(x)1 <_p^x r(x)1 \overset{r-prog}{\longrightarrow} r(z)0$. Similarly, process q must view the action $w_p(y)1$ after

its read action $r(y)0$ and $w(z)1 \overset{r-prog}{\longrightarrow} w(x)2 <_q^x r(x)2 \overset{r-prog}{\longrightarrow} r(y)0$. Thus, $w(y)1 \overset{r-prog}{\longrightarrow} w(x)1 <_p^x r(x)1 \overset{r-prog}{\longrightarrow} r(z)0 <_p^z w(z)1 \overset{r-prog}{\longrightarrow} w(x)2 <_q^x r(x)2 \overset{r-prog}{\longrightarrow} r(y)0 <_q^y w(y)1$ is an extended cycle in $\overset{\widehat{pcd}}{\longmapsto}$. This implies that any linearization that satisfies the first 2 conditions of PCGharachorloo will cause a cycle in $(O, \overset{\widehat{pcd}}{\longmapsto})$. Hence computation 14 is not PCGharachorloo.

The example is possible due to the use of write buffers in the DASH machine. The behaviour that is not allowed by PCGharachorloo, but is possible in the DASH system, occurs when a processor p reads a value from its write buffer, which contains only writes by p, before it has been serialized in p's copy of the memory. Thus, if in some execution and for some processor p, a processor$_p$-write-request(x, v) α_w, precedes a processor$_p$-read-request(x), α_r, the μ_p-reply(x, v) matching α_r could precede the μ_p-update$_p(x, v)$ matching α_w in the execution. In the DASH, it is therefore unnecessary that all actions by p to the same location are implemented by μ_p in program order to assure that the system is coherent.

6.3.3.1 M_{DASH} and the memory model it implements

In M_{DASH} a read action $(read, x, \lambda, v)$ is implemented by the following ordered sequence of events:

1. a processor$_p$-read-request(x) to p's copy of the memory, and
2. a matching μ_p-reply(x, v) from p's copy of the memory or from p's write buffer to p.

A write action $(write, x, v, \lambda)$ is implemented by the ordered sequence of events:

1. a processor$_p$-write-request(x, v) to μ_p, and

2. a matching μ_q-update$_p(x, v)$ at each process q's copy of the memory.

Furthermore, any execution E on M_{DASH}, which is a sequence of these 4 events, will also meet all the following constraints, for all processors p, q, r, s, locations x and values u, v, u'.

1. A read by p of x returns the value of the most recent update of μ_p at x, unless the processor$_p$-read-request follows some processor$_p$-write-request to x and the matching μ_p-reply is followed by the matching μ_p-update$_p$ event. In that case, the read returns the value of the latest such processor$_p$-write-request event (and the reply is from the write buffer). That is, for each μ_p-reply(x, v) event, m_r, with matching processor$_p$-read-request, α_r, if μ_p-update$_q(x, u)$ is the last preceding μ_p-update event of location x then $v = u$, unless processor$_p$-write-request$(x, u) \xrightarrow{E} \alpha_r \xrightarrow{E} \mu_p$-reply$(x, v) \xrightarrow{E} \mu_p$-update$_p(x, u)$ for some $(write, x, u', \lambda)$. In this case, v is the value of the last processor$_p$-write-request to x preceding α_r in E.

2. Each copy of the memory implements writes to the same location in the same order. That is, if μ_p-update$_q(x, v) \xrightarrow{E} \mu_p$-update$_r(x, u)$ then the matching μ_s events satisfy μ_s-update$_q(x, v) \xrightarrow{E} \mu_s$-update$_r(x, u)$.

3. Read actions are received by the memory in program order. That is, for any two processor$_p$-read-requests, α_1 and α_2, where r_1, r_2 are the matching μ_p-reply events respectively , if $\alpha_1 \xrightarrow{E} \alpha_2$, then $r_1 \xrightarrow{E} r_2$.

4. If an action o precedes a write in program order, then all events corresponding to o precede any update event corresponding to the write. That is, for any

processor$_p$ event α and any processor$_p$-write-request α_w, and for all μ_q event(s) m matching α and for all μ_q events m_w matching α_w, if $\alpha \xrightarrow{E} \alpha_w$ then $m \xrightarrow{E} m_w$.

Note that $M_{Gharachorloo}$ and M_{DASH} are very similar machines. Constraint 3 of $M_{Gharachorloo}$ is satisfied by M_{DASH}, but constraint 1 of M_{DASH} is different than constraint 1 of $M_{Gharachorloo}$ to capture the fact that in M_{DASH} a read might receive its value from the write buffer. The effect of the added write buffer is that any μ_p-update$_q(x, v)$ event is unnoticed by (or invisible to) processor p in some execution E if it occurs in E between a processor$_p$-write-request(x, u) event and its matching μ_p-update$_p(x, u)$ event. No μ_p-reply from location x will return that value v by constraint 1 of M_{DASH}.

To simplify the proof that M_{DASH} implements exactly the formal model presented later in this section, any execution on M_{DASH} is altered to an execution that blocks on read actions but is still an execution on M_{DASH}. For any execution E on M_{DASH}, let the *altered-execution*, E', be constructed as follows. Initially, $E' \leftarrow E$. Consider each event in E' in turn, starting with the last event in E' and moving back through E' to the first event in E'. If the event being considered is a μ_p-reply(x, v) event, m_r, let the matching processor$_p$-read-request(x) be α_r. By the ordering of events corresponding to a read in E, $E' = S_1, \alpha_r, S_2, m_r, S_3$ for some sequences S_1, S_2, and S_3. Let S_2^1 be the subsequence of S_2 containing exactly all processor$_p$-write-request events and let S_2^2 be the subsequence of S_2 containing the remaining events. Then $E' \leftarrow S_1, S_2^2, \alpha_r, m_r, S_2^1, S_3$.

Lemma 6.3.11 *If E' is the altered-execution of E, an execution of M_{DASH}, then E and E' agree on the ordering of memory events.*

Lemma 6.3.12 *If E' is the altered-execution of E, an execution of M_{DASH}, then read actions are blocking in E'. That is, each processor$_p$-read-request is immediately followed by its matching μ_p-reply event in E'.*

Lemmas 6.3.11 and 6.3.12 follow immediately from the construction of an altered-execution.

Lemma 6.3.13 *If E' is the altered-execution of E, an execution of M_{DASH}, then E and E' agree on the ordering of processor$_p$ events for each processor p in M_{DASH}.*

Proof: Let α_r and α'_r be any processor$_p$-read-request events in E matching, respectively, to μ_p-reply events m_r and m'_r. Let α_w and α'_w be any processor$_p$-write-request events in E matching, respectively, to μ_q-update$_p$ events m_w and m'_w for any processor q. Consider the following four cases:

1. If $\alpha_w \xrightarrow{E} \alpha'_w$ then $\alpha_w \xrightarrow{E'} \alpha'_w$ since the ordering of processor$_p$-write-requests is not changed during the construction of E'.

2. If $\alpha_r \xrightarrow{E} \alpha'_r$ then, by constraint 4 of M_{DASH}, $m_r \xrightarrow{E} m'_r$. By lemma 6.3.12 and 6.3.11, $\alpha_r \xrightarrow{E'} m_r \xrightarrow{E'} \alpha'_r \xrightarrow{E'} m'_r$. Thus, $\alpha_r \xrightarrow{E'} \alpha'_r$.

3. If $\alpha_w \xrightarrow{E} \alpha_r$, let m'_r be the last μ_p-reply event such that $\alpha'_r \xrightarrow{E} \alpha_w \xrightarrow{E} m'_r$. Then, $\alpha'_r \xrightarrow{E} \alpha_w \xrightarrow{E} \alpha_r$, which implies that $m'_r \xrightarrow{E} m_r$. During the construction of E', α_w is moved behind m'_r such that there are only processor$_p$-write-request events between m'_r and α_w, and by lemma 6.3.12, α_r immediately precedes m_r in E'. Hence, $m'_r \xrightarrow{E'} \alpha_w \xrightarrow{E'} \alpha_r \xrightarrow{E} m_r$. Thus, E and E' agree on the ordering of α_w and α_r. If no m'_r exists such that $\alpha'_r \xrightarrow{E} \alpha_w \xrightarrow{E} m'_r$ then α_w is not moved during the construction of E' and α_r can only move to occur later in E'. Hence, $\alpha_w \xrightarrow{E} \alpha_r$.

4. If $\alpha_r \xrightarrow{E} \alpha_w$ then there are two possible sub cases. If $\alpha_r \xrightarrow{E} m_r \xrightarrow{E} \alpha_w$ then clearly $\alpha_r \xrightarrow{E'} m_r$. During the construction of E', no processor$_p$-write-request event is moved back, that is, a processor$_p$-write-request event only precedes a μ_p-reply event in E' if it did so in E. Hence, $m_r \xrightarrow{E'} \alpha_w$ and $\alpha_r \xrightarrow{E'} \alpha_w$. If $\alpha_r \xrightarrow{E} \alpha_w \xrightarrow{E} m_r$ then α_w succeeds either m_r or some other μ_p-reply event that succeeds m_r in E and E'. Since $\alpha_r \xrightarrow{E'} m_r$, $\alpha_r \xrightarrow{E'} \alpha_w$.

Hence, E and E' agree on the ordering of processor$_p$ events for all processors p in M_{DASH}. ■

Lemma 6.3.14 *If E' is the altered-execution of E, an execution on M_{DASH}, then any processor$_p$ event in E' precedes its matching memory events in E'.*

Proof: By the construction of E', processor$_p$-read-request events clearly precede the matching μ_p-reply event. Consider any processor$_p$-write-request event, α_w. For all processor$_p$-read-request events, α_r, such that $\alpha_r \xrightarrow{E} \alpha_w$, the μ_p-reply event matching α_r precedes all μ_q-update$_p$ events matching α_w by constraint 4 of M_{DASH}. Thus, during the construction of E', α_w is not moved ahead of any of its matching μ_q-update$_p$ events. Hence, any processor$_p$-write-request event in E' precedes all its matching μ_q-update$_p$ events in E'. ■

Lemma 6.3.15 *If E' is the altered-execution of E, an execution on M_{DASH}, then for any processor$_p$-write-request event, α_w, with matching μ_p-update$_p$ event, m_w^p, and for any processor$_p$-read-request event, α_r, with matching μ_p-reply event, m_r, if $\alpha_w \xrightarrow{E} \alpha_r \xrightarrow{E} m_r \xrightarrow{E} m_w^p$ then $\alpha_w \xrightarrow{E'} \alpha_r \xrightarrow{E'} m_r \xrightarrow{E'} m_w^p$.*

Proof: By lemma 6.3.11, $m_r \xrightarrow{E'} m_w^p$. By lemma 6.3.14, $\alpha_r \xrightarrow{E'} m_r$. By lemma 6.3.13, $\alpha_w \xrightarrow{E'} \alpha_r$. ■

Claim 6.3.16 *If E' is the altered-execution of E, an execution of M_{DASH}, then E' is an execution of M_{DASH}.*

Proof: The ordering of events corresponding to an action is satisfied by E' by lemma 6.3.14. Constraint 1 of M_{DASH} is satisfied by E', by lemmas 6.3.11 and 6.3.15. Constraint 2 of M_{DASH} is satisfied by lemma 6.3.11. Constraints 3 and 4 of M_{DASH} are satisfied by E' by lemma 6.3.13 and 6.3.11. Hence, E' is an execution of M_{DASH}. ■

The formal definition of the memory model implemented by M_{DASH} requires a new relation. Consider any computation of system (P, J) containing set of actions O. Let $S_1 = \{<_p\}$ and $S_2 = \{\xrightarrow{view_p}\}$ be sets of sequences over the set of actions O. Then the relation $(O, \overset{pcd(S_1,S_2)}{\longmapsto})$ is defined as follows: $o_1 \overset{pcd(S_1,S_2)}{\longmapsto} o_2$ iff $\exists p \in P$ and $\exists x \in J$ such that

1. $o_1, o_2 \in O|p$ and $o_1 \xrightarrow{r-prog} o_2$, or
2. $\exists q \neq p \in P$ such that $o_1 \in (O_w|q)|x$, $o_2 \in (O_r|p)|x$ and $o_1 <_p o_2$, or
3. $o_1, o_2 \in O_w|x$ and $o_{1_{O_w}} \xrightarrow{view_p} o_{2_{O_w}}$, or
4. $o_1 \in (O_r|p)|x$, $o_2 \in O_w$ and $\exists o' \in O_w|x$ such that $o_1 <_p o' \xrightarrow{r-prog} o_2$.

For any cycle $\beta = o_0, o_1, \ldots o_k$ in $(O, \overset{pcd(S_1,S_2)}{\longmapsto})$, let the *extended cycle* $\widehat{\beta}$ be β augmented as follows. For every i such that $o_i \overset{pcd(S_1,S_2)}{\longmapsto} o_{i+1}$ arises from part 4 of the

definition of $\overset{pcd(S_1,S_2)}{\longmapsto}$, insert between o_i and o_{i+1} the o' action that is guaranteed by part 4. Call any such o' an *inserted* action.

Recall the definition of $(A \uplus B)$, $O_{\overrightarrow{invisible_p}}$ and $O_{memupdates_p}$ on page 42.

Definition 6.3.17 *Let O be all the actions of a computation C of the multiprocessor system (P, J). Then C is PCDash if for each processor $p \in P$ there is some total order $(O|p \uplus O_w, \overset{view_p}{\longrightarrow})$ such that*

1. $(O_w, \overset{prog}{\longrightarrow}) \subseteq (O_w, \overset{view_p}{\longrightarrow})$, *and*
2. $\forall x \in J$ *and* $\forall q \in P$ $(O_w|x, \overset{view_p}{\longrightarrow}) = (O_w|x, \overset{view_q}{\longrightarrow})$, *and*
3. $(O|p, \overset{prog}{\longrightarrow}) = (O|p, \overset{view_p}{\longrightarrow})$, *and*
4. *if* $w \in O_w|p$ *then* $w_{O|p} \overset{view_p}{\longrightarrow} w_{O_w}$, *and*
5. $<_{L_p} = ((O|p \uplus O_w) \backslash (O_{invisible_p} \cup O_{memupdates_p}), \overset{view_p}{\longrightarrow})$ *is a linearization, and*
6. $(O, \overset{pcd(\{<_{L_q}|q\in P\},\{\overset{view_q}{\longrightarrow}|q\in P\})}{\longmapsto})$ *is cycle-free.*

Abusing notation, I use $<_{L_p}$ to denote both the linearization and the total order relation that is inherent in the sequence.

6.3.3.2 M_{DASH} implements exactly PCDash

In the following discussion, for any set of actions O, any object x, any process p, and any sequence $(O|p \uplus O_w, \overset{view_p}{\longrightarrow})$, let $o_1 <_p^x o_2$ iff either $(o_1, o_2) \in (O_w, \overset{view_p}{\longrightarrow})$ or $(o_1, o_2) \in (((O|p \uplus O_w) \backslash (O_{invisible_p} \cup O_{memupdates_p}))|x, \overset{view_p}{\longrightarrow})$. When the sets S_1 and S_2 are understood, $(O, \overset{pcd}{\longmapsto})$ is used to denote $(O, \overset{pcd(S_1,S_2)}{\longmapsto})$. Furthermore, all results of calculations on indices are reduced modulo $k+1$. The proof that M_{DASH} implements exactly PCDash is very similar to the proof that $M_{Gharachorloo}$ implements

exactly PCGharachorloo. The following two lemmas are used to prove that M_{DASH} implements exactly PCDash.

Lemma 6.3.18 *Let O be some set of actions, P some set of processes, and J some set of objects. Let $\{(O|p \uplus O_w, \xrightarrow{view_p}) \mid p \in P\}$ be a set of sequences satisfying conditions 1 to 5 of definition 6.3.17, and let $\beta = o_0, o_1, \ldots, o_k$ be any extended cycle of β', a cycle in $(O, \overset{pcd}{\longmapsto})$. If $\exists p, q \in P$, and $\exists x \in J$, and $\exists i$, $0 \leq i \leq k$ such that $o_i <_p^x o_{i+1} <_q^x o_{i+2}$, then either*

1. *there exists some shorter extended cycle of $(O, \overset{pcd}{\longmapsto})$, or*
2. *$o_i <_p^x o_{i+1} \overset{r-prog}{\longrightarrow} o_{i+2}$, or*
3. *$o_i \overset{r-prog}{\longrightarrow} o_{i+1} <_q^x o_{i+2}$.*

The proof is identical to the proof of lemma 6.3.2 on page 64 with $\overset{\widehat{pcd}}{\longmapsto}$ replaced by $\overset{pcd}{\longmapsto}$.

Lemma 6.3.19 *Let O be some set of actions, P some set of processes, and J some set of objects. Let $\{(O|p \uplus O_w, \xrightarrow{view_p}) \mid p \in P\}$ be a set of sequences satisfying conditions 1 to 5 of definition 6.3.17, and let $\beta = o_0, o_1, \ldots, o_k$ be any extended cycle of β', a cycle in $(O, \overset{pcd}{\longmapsto})$. If, for some i, $0 \leq i \leq k$, $o_i \overset{r-prog}{\longrightarrow} o_{i+1} \overset{r-prog}{\longrightarrow} o_{i+2}$, then $o_0, o_1, \ldots, o_i, o_{i+2}, \ldots, o_k$ is also an extended cycle in $(O, \overset{pcd}{\longmapsto})$.*

The proof is identical to the proof of lemma 6.3.3 on page 65 with $\overset{\widehat{pcd}}{\longmapsto}$ replaced by $\overset{pcd}{\longmapsto}$.

These preceding two lemmas are now combined to conclude that, for any shortest extended cycle, the relation between consecutive actions in the cycle alternates. (Recall the definition of alternating from page 66.)

Corollary 6.3.20 *Let O be some set of actions, P some set of processes, and J some set of objects, and $\{(O|p \uplus O_w, \xrightarrow{view_p}) \mid p \in P\}$ be a set of sequences satisfying conditions 1 to 5 of definition 6.3.17, and $\beta = o_0, o_1, \ldots, o_k$ be any of the shortest extended cycle of $(O, \overset{pcd}{\longmapsto})$. Then any shortest extended cycle of $(O, \overset{pcd}{\longmapsto})$ alternates.*

The proof is identical to the proof of corollary 6.3.4 on page 66.

Theorem 6.3.21 *M_{DASH} implements exactly PCDash.*

The theorem follows directly from the following two lemmas.

Lemma 6.3.22 *Any computation arising from an execution of M_{DASH} is PCDash.*

Proof: To show that any computation C resulting from some execution E on M_{DASH} is PCDash, I use the fact that this computation also results from the altered-execution of E, E'. I first construct the total orders $(O|p \uplus O_w, \xrightarrow{view_p})$ by taking the subsequence of E' containing the processor$_p$-write-request and μ_p events. I then show that each $(O|p \uplus O_w, \xrightarrow{view_p})$ satisfies PCDash.

Let E be any execution on M_{DASH} with resulting computation C and set of actions O and let E' be the altered-execution of E. Recall that E' is also an execution on M_{DASH} by claim 6.3.16 and thus satisfies all the constraints of M_{DASH}. Furthermore, by lemmas 6.3.11, 6.3.13 and 6.3.15, C is also a computation resulting from execution E'. Let $P = \{\bar{p} \mid$ processor p is in $M_{DASH}\}$ be a set of processes such that the i^{th} action-invocation in each $\bar{p}$ corresponds to the i^{th} processor$_p$ event in E'. The set of objects J is the set of all locations in M_{DASH}. For each processor p let E_p be the subsequence of E' containing all processor$_p$-write-request and all μ_p events. For each process $\bar{p} \in P$, let $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ be the sequence of all write actions and all

actions by process $\bar{p}$ such that the i^{th} action in $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ corresponds with the i^{th} event in E_p. Note that the write actions by $\bar{p}$ appear twice, one copy in $O|\bar{p}$ and one in O_w. Specifically, if the i^{th} action of $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$ is o_i and the i^{th} event of E_p is a μ_p-update event then $o_i \in O_w$, otherwise $o_i \in O|\bar{p}$.

By constraint 4 of M_{DASH}, all update events by the same processor appear in program order in E' and, by constraint 2 of M_{DASH}, all updates at the same location appear in the same order across all copies of the memory in E'. Thus conditions 1 and 2 of PCDash are satisfied by each $(O_w, \xrightarrow{view_p})$. Since processor$_p$ events occur in program order in E', and since processor$_p$-read-request events immediately precede the matching μ_p-reply event, condition 3 is satisfied by each $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$. By lemma 6.3.14, each processor$_p$-write-request event appears before the matching μ_p-update$_p$ event in E', hence, condition 4 is also satisfied by each $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$.

Now consider each sequence $L_p = ((O|\bar{p} \uplus O_w) \setminus (O_{invisible_{\bar{p}}} \cup O_{memupdates_{\bar{p}}}), \xrightarrow{view_p})$. To show that each is a linearization, consider any $r \in (O_r|\bar{p})|x$ for any process $\bar{p}$ and any object x and let w be the last write to x to precede r in L_p. Let α_r and m_r be, respectively, the processor$_p$-read-request and μ_p-reply events corresponding to r and let α_w and m_w be, respectively, the processor$_q$-write-request and μ_p-update$_q$ events corresponding to w. If $w \in O|\bar{p}$ then $w_{O|\bar{p}} \xrightarrow{view_p} r$ implies that $w \xrightarrow{prog} r$ since it has been shown that $\xrightarrow{view_p}$ satisfies condition 3 of PCDash. Hence, $\alpha_w \xrightarrow{E'} \alpha_r$. By the choice of w, there is no processor$_p$-write-request to x event α'_w such that $\alpha_w \xrightarrow{E'} \alpha'_w \xrightarrow{E'} \alpha_r$. If $\alpha_w \xrightarrow{E'} \alpha_r \xrightarrow{E'} m_r \xrightarrow{E'} m_w$, then, by constraint 1 of M_{DASH}, m_r returns the update value of α_w and thus w and r contain the same value. If $\alpha_w \xrightarrow{E'} m_w \xrightarrow{E'} m_r$ then, by the construction of $\xrightarrow{view_p}$, $w_{O|\bar{p}} \xrightarrow{view_p} w_{O_w} \xrightarrow{view_p} r$. By the choice of w, there is no

$w' \in (O_w|\bar{p})|x$ such that $w_{O|\bar{p}} \xrightarrow{view_p} w'_{O|\bar{p}} \xrightarrow{view_p} r$. Hence, there is no $w' \in (O_w|\bar{p})|x$ such that $w_{O_w} \xrightarrow{view_p} w'_{O_w} \xrightarrow{view_p} r$. Furthermore, there is no $w' \in (O_w|\bar{q})|x$ for some $\bar{q} \neq \bar{p}$ such that $w_{O_w} \xrightarrow{view_p} w' \xrightarrow{view_p} r$ since such a w' would not be in the set $O_{invisible_{\bar{p}}}$. Hence, m_w is the last write to x preceding m_r in E'.

If $w \in O|\bar{q}$, for some $\bar{q} \neq \bar{p}$, then, by construction of $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$, $m_w \xrightarrow{E'} m_r$. $m_w \xrightarrow{E} m_r$. Since $w \notin O_{invisible_{\bar{p}}}$, there is no $w' \in (O_w|\bar{p})|x$ such that $w'_{O|\bar{p}} \xrightarrow{view_p} w \xrightarrow{view_p} w'_{O_w}$. Furthermore, by the choice of w, there is no write by $\bar{p}$ to x whose copy from $O|\bar{p}$ appears between w and r. Hence, there is no write by $\bar{p}$ to x, whose copy from O_w appears between w and r and there is no $\hat{w} \in O_{invisible_{\bar{p}}}|x$ such that $w \xrightarrow{view_p} \hat{w} \xrightarrow{view_p} r$. Hence, there is no update of x between m_w and m_r in E' and, by constraint 1 of M_{DASH}, m_r returns the update value of m_w. Thus, w and r contain the same value.

The remainder of this proof is identical in the proof of lemma 6.3.6 on page 67. It is stated here for completeness.

It remains to show that $(O, \xrightarrow{pcd})$ is cycle-free to establish that computation C is PCDash. Assume for a contradiction that it is not cycle-free. Let $\beta = o_0, o_1, \ldots, o_m$ be any of the shortest extended cycles in $(O, \xrightarrow{pcd})$. By corollary 6.3.20 (and by selectively choosing which action is named o_0 in β) the cycle can be written as $o_0 <^{x_0}_{\bar{p}_0} o_1 \xrightarrow{r-prog} o_2 <^{x_2}_{\bar{p}_2} o_3 \xrightarrow{r-prog} \ldots <^{x_{m-1}}_{\bar{p}_{m-1}} o_m \xrightarrow{r-prog} o_0$ for some processes $\bar{p}_0, \bar{p}_2, \ldots, \bar{p}_{m-1} \in P$ and some objects $x_0, x_2, \ldots x_{m-1} \in J$. For any i, j, let $\mu^j_{p_i}$ be the memory event at p_i's copy of the memory, corresponding to the action o_j. Then, by construction of each $(O|\bar{p} \uplus O_w, \xrightarrow{view_p})$, $o_i <^{x_i}_{\bar{p}_i} o_{i+1}$ implies that $\mu^i_{p_i} \xrightarrow{E} \mu^{i+1}_{p_i}$. Furthermore, if $o_i \xrightarrow{r-prog} o_{i+1}$ then, by constraints 4 and 5 of M_{DASH}, $\mu^i_{p_{i-1}} \xrightarrow{E} \mu^{i+1}_{p_{i+1}}$. Thus $o_0 <^{x_0}_{\bar{p}_0} o_1 \xrightarrow{r-prog} o_2 <^{x_2}_{\bar{p}_2} o_3 \xrightarrow{r-prog} \ldots <^{x_{m-1}}_{\bar{p}_{m-1}} o_m \xrightarrow{r-prog} o_0$ implies that

$\mu^0_{p_0} \xrightarrow{E} \mu^1_{p_0} \xrightarrow{E} \mu^2_{p_2} \xrightarrow{E} \mu^3_{p_2} \xrightarrow{E} \ldots \xrightarrow{E} \mu^m_{p_{m-1}} \xrightarrow{E} \mu^0_{p_0}$. Thus there is a cycle in the execution E and this is clearly impossible. Hence computation C is PCDash. ■

Lemma 6.3.23 *Any PCDash computation is the result of some execution of* M_{DASH}.

Proof: To show that all PCDash computations arise from some execution on $M_{Gharachorloo}$, I will first construct an execution by merging the linearizations and program orders of each process. I will then show that this execution satisfies all the constraints of an execution on M_{Dash}.

Consider any PCDash computation C of system (P, J) with resulting set of actions O. Let the processor that implements process $p \in P$ be named $\hat{p}$ and let the objects in J be the locations in M_{DASH}. Choose any set of sequences $(O|p \uplus O_w, \xrightarrow{view_p})$, one for each process p, that satisfy PCDash. For all $p \in P$, initially $L_p = (O_r|p \cup O_w, \xrightarrow{view_p}) = o^p_1, o^p_2, \ldots o^p_{k_p}$ and $P_p = (O|p, \xrightarrow{prog}) = a^p_1, a^p_2, \ldots a^p_{l_p}$ and $i_p = 1$. Also, initially $i = 1$ and $E = \lambda$. Then the algorithm

while $(\forall k_p\ i <= k_p)$ **do**
  while $(\exists p \in P$ such that $o^p_i \in L_p)$ **do**
    for $m \leftarrow 1$ **to** n **do**
      ConsiderAdding(i, p_m)
    end for
  end while
  $i \leftarrow i + 1$
end while

constructs the sequence E containing all processor and $\mu_{\hat{p}}$ events corresponding to the actions in O where the procedure ConsiderAdding is shown in figure 6.3.3.2. Note that each sequence L_p initially contains each read action by p and all writes from the set O_w and that L_p is derived from $\xrightarrow{view_p}$ rather than from $<_{L_p}$.

```
      procedure ConsiderAdding( index j, process p )
1         if (o_j^p ∈ L_p and ∃x ∈ J such that o_j^p ∈ (O_r|p)|x and
              the last write not by p to x preceding o_j^p in (O_r|p ∪ O_w, --view_p-->) is no longer
              in L_p and the last read preceding o_j^p in program order is no longer in L_p
              ) then
2             if (the processor_p̂ event corresponding to o_j^p is not in E yet) then
3                repeat
4                   append to E the processor_p̂ event corresponding to a_{i_p}^p
5                   remove a_{i_p}^p from P_p
6                   i_p ← i_p + 1
7                until (a_{i_p-1}^p = o_j^p)
              end if
8             append to E the μ_p̂-reply event corresponding to o_j^p
9             remove o_j^p from L_p
          end if
10        if (o_j^p ∈ L_p and ∃x ∈ J and ∃q ∈ P such that o_j^p ∈ (O_w|q)|x and
              all events corresponding to the last action preceding o_j^p in program order
              are all in E and the last action to x preceding o_j^p in (O_r|p ∪ O_w, --view_p-->), is
              no longer in L_p) then
11            if (the processor_q̂ event corresponding to o_j^p is not in E yet) then
12               repeat
13                  append to E the processor_q̂ event corresponding to a_{i_q}^q
14                  remove a_{i_q}^q from P_q
15                  i_q ← i_q + 1
16               until (a_{i_q-1}^q = o_j^p)
              end if
17            append to E the μ_p̂-update_q̂ event corresponding to o_j^p
18            remove o_j^p from L_p
19        else
20            ConsiderAdding(j + 1, p)
          end if
```

Figure 6.8: *Part of construction of an execution E on M_{DASH}.*

To show that E is an execution that could have occurred on M_{DASH}, assume first that this algorithm exhausts each L_p. Thus, all events corresponding to actions in O are in E since lines 2 to 9 add events corresponding to a read action (and possibly some extra processor events) and lines 12 to 18 add events corresponding to a write action (and possibly some extra processor events). Furthermore, all processor events are appended to E in program order in lines 4 and 13. It is obvious that the algorithm ensures that the processor events precede the matching memory events. Lines 1 and 10 ensure that $\mu_{\hat{p}}$ memory events to the same location are in the same order in E as the corresponding actions are ordered in $(O_w, \xrightarrow{view_p})$.

Each sequence $(O|p \uplus O_w, \xrightarrow{view_p})$ agrees on the ordering of writes from O_w to the same location, thus updates of the same location will also agree in E across all copies of the memory, and thus E satisfies constraint 2 of M_{DASH}. Line 1 of the algorithm ensures that $\mu_{\hat{p}}$-reply events are appended to E in the same order as the corresponding reads appear in $(O|p \uplus O_w, \xrightarrow{view_p})$, thus, by condition 2 of PCDash, E satisfies constraint 3. Similarly, the if-statement of line 10 ensures that if $o_1 \xrightarrow{prog} o_2$ and, for some process q, $o_1 \in O|q$ and $o_2 \in O_w|q$, then all memory events corresponding to o_1 are in E before any memory event corresponding to o_2 is appended to E, thus constraint 5 of M_{DASH} is also satisfied by E.

Finally, to show that constraint 1 of M_{DASH} is satisfied, consider any $\mu_{\hat{p}}$-reply event m_r from location x, with matching processor$_{\hat{p}}$-read-request event, α_r, and corresponding to read action r. If m_r is between some processor$_{\hat{p}}$-write-request event of x and its matching $\mu_{\hat{p}}$-update$_{\hat{p}}$ event in E, let α_w be the last processor$_{\hat{p}}$-write-request event to x preceding m_r in E. Let m_w be the $\mu_{\hat{p}}$-update$_{\hat{p}}$ event matching α_w and let w be the write action corresponding to α_w. Since $\alpha_w \xrightarrow{E} m_r$, $w \xrightarrow{prog} r$, by the

construction of E. Thus, by condition 3 of PCDash, $w_{O|p} \xrightarrow{view_p} r$ and, by the choice of α_w, there is no $w' \in (O_w|p)|x$ such that $w_{O|p} \xrightarrow{view_p} w'_{O|p} \xrightarrow{view_p} r$. If $w_{O|p} \xrightarrow{view_p} r \xrightarrow{view_p} w_{O_w}$, then, since any write by $q \neq p$ to x between $w_{O|p}$ and r is in the set $O_{invisible_p}$ and any write by p to x between w and r is in the set $O_{memupdates_p}$, w is the last write to x preceding r in $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p})$. If $w_{O|p} \xrightarrow{view_p} w_{O_w} \xrightarrow{view_p} r$, then, by line 1 of the algorithm, and since $m_r \xrightarrow{E} m_w$, for any process $q \neq p$, there is no $w' \in (O_w|q)|x$ such that $w_{O_w} \xrightarrow{view_p} w'_{O_w} \xrightarrow{view_p} r$ and w is the last write to x preceding r in $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p})$. Thus, in either case, w and r contain the same value and m_r returns the value of α_w, satisfying constraint 1 of PCDash.

If no such α_w exists, then let m_w be the last $\mu_{\tilde{p}}$-update event of x preceding m_r in E with matching processor$_{\tilde{p}}$-write-request event α_w and corresponding to write action w. By construction of E, $w_{O_w} \xrightarrow{view_p} r$ and there is no $w' \in O_w$ such that $w_{O_w} \xrightarrow{view_p} w'_{O_w} \xrightarrow{view_p} r$. If $w \in O|q$ for some process $q \neq p$, then clearly w is also the last write to x preceding r in $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p})$, which is a linearization. Thus r and w contain the same value and m_r returns the update value of m_w. If $w \in O|p$ then $w_{O|p}$ is the last write from $O|p$ that precedes w_{O_w} in $\xrightarrow{view_p}$ and since all writes not by p between $w_{O|p}$ and w_{O_w} are in $O_{invisible_p}$, w is also the last write to x preceding r in $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p})$. Hence, w and r contain the same values and m_r returns the value that m_w wrote. Hence, constraint 1 of M_{DASH} is satisfied.

The remainder of this proof is identical in the proof of lemma 6.3.7 on page 68, except case 2 below is changed slightly. It is stated here for completeness.

So it remains to verify that each L_p is exhausted by the algorithm and that all events are added to E. Assume instead that at some point in the construction of E, for some m, the sequences $L_{p_1}, L_{p_2}, \ldots, L_{p_m}$ are not exhausted, and the algorithm is in the while loop of line 22 infinitely. For any $1 \leq i \leq m$, consider any action $o \in L_{p_i}$. Then o cannot be removed from L_{p_i} (and the μ_{p_i} memory event cannot be added to E) by one of the following 6 cases for some object $x \in J$ and some process p_j, p_k, where $1 \leq j, k \leq m$:

1. $o \in O_r$ and $\exists o' \in O_r$ such that $o' \xrightarrow{r-prog} o$ and $o' \in L_{p_i}$, or
2. $o \in O_r|x$ and $\exists o' \in (O_w|p_j)|x$, $p_j \neq p_i$, such that $o' <^x_{p_i} o$ and $o' \in L_{p_i}$, or
3. $o \in O_w|p_j$ and $\exists o' \in O_r|p_j$ such that $o' \xrightarrow{r-prog} o$ and $o' \in L_{p_j}$, or
4. $o \in O_w|p_j$ and $\exists o' \in O_w|p_j$ such that $o' \xrightarrow{r-prog} o$ and $o' \in L_{p_k}$, or
5. $o \in O_w|x$ and $\exists o' \in (O_r|p_i)|x$ such that $o' <^x_{p_i} o$ and $o' \in L_{p_i}$, or
6. $o \in O_w|x$ and $\exists o' \in O_w|x$ such that $o' <^x_{p_i} o$ and $o' \in L_{p_i}$.

Thus the actions in the sequences L_{p_1} to L_{p_m} form cycles, where each link is one of the above 6 cases. Notice that in the cases 1 to 4 and 6, $o' \xrightarrow{pcd} o$. Thus, if any cycle does not contain a case 5 link, this means that there is a cycle in $(O, \xrightarrow{pcd})$, which would imply that C is not PCDash. Hence, at least one of the links is due to case 5. Examine any cycle $\beta = o_0, o_1, \ldots o_k$ with links of the above cases. I will show that removing some actions from β results in an extended cycle of $(O, \xrightarrow{pcd})$. Choose any case 5 link in β and call the write action of the link w and let r be the read to the same object x of the link. Thus, for some process p_i, $r <^x_{p_i} w$ and $r \in L_{p_i}$. Let H_1

be the sequence starting with the action w and followed by the sequence of actions following w in β, such that each action is from L_{p_i}, and such that each action is related to the next by $(O, \stackrel{pcd}{\longmapsto})$. Let H_2 start with the action o_{j+1}, where o_j is the last action of H_1, followed by the sequence of actions that follows o_{j+1} in β such that all actions of H_2 are from the same L_p sequence and each is related to its successor by $(O, \stackrel{pcd}{\longmapsto})$. Continue building these sequences, until all actions of β are in some sequence, resulting in the sequences H_1 to $H_{\hat{m}}$ for some $\hat{m} > 0$. Without loss of generality, let the sequences starting with a write that is the second action in a case 5 link be H_1 to H_l for some $l \geq 1$ and let the remaining sequences be H_{l+1} to $H_{\hat{m}}$.

For any $1 \leq i \leq l$, let the first action of any sequence H_i be named $w_i \in O_w|x$ for some object x, and for some process p_j, $w_i \in L_{p_j}$. There must be some action $r_i \in (O_r|p_j)|x$ such that $r_i <^x_{p_j} w_i$ since w_i is part of a case 5 link. Note that r_i must be related to its predecessor in β by $(O, \stackrel{pcd}{\longmapsto})$. If H_i is of length 1, then w_i is the first action (the o' action) of a case 4 link, and thus $\exists w'_i \in O_w$ such that $w_i \stackrel{r-prog}{\longrightarrow} w'_i$. If $o_{\tilde{k}}$ is the action immediately following w'_i in β then $w'_i \stackrel{pcd}{\longmapsto} o_{\tilde{k}}$ since this cannot be a case 5 link. Hence, $r_i <^x_{p_i} w_i \stackrel{r-prog}{\longrightarrow} w'_i \stackrel{pcd}{\longmapsto} o_{\tilde{k}}$. By part 4 of $(O, \stackrel{pcd}{\longmapsto})$, $r_i \stackrel{pcd}{\longmapsto} w'_i$ and thus $o_k \stackrel{pcd}{\longmapsto} r_i \stackrel{pcd}{\longmapsto} w'_i \stackrel{pcd}{\longmapsto} o_{\tilde{k}}$ and w_i is an inserted action of $(O, \stackrel{pcd}{\longmapsto})$. If w_i is immediately followed by some read action r'_i , then $w_i <^x_{p_j} r'_i$ and $r_i \in (O_r|p_j)|x$. Since r_i and r'_i are by the same process, $r_i \stackrel{r-prog}{\longrightarrow} r'_i$, and thus $r_i \stackrel{pcd}{\longmapsto} r'_i$. Remove all these w_i write actions from β to form the cycle β'. If the action immediately following w_i is some write action w'_i and if $w_i \stackrel{r-prog}{\longrightarrow} w'_i$, then $r_i \stackrel{pcd}{\longmapsto} w'_i$. Hence, w_i is an inserted action of $(O, \stackrel{pcd}{\longmapsto})$. (Note that w'_i must be related to its successor in β and β' by $(O, \stackrel{pcd}{\longmapsto})$). If w_i is immediately followed by some write action w'_i and $w_i <^x_{p_j} w'_i$, then also, $r_i <^x_{p_j} w'_i$. Such a write must be followed by a sequence of zero or more writes, all

related only because they are to the same object, but must finally be followed by either a read or another write in the relaxed program order, since the sequence is of finite length and can only be ended through a link using the relaxed program order. Thus $r_i <^x_{p_j} w_i <^x_{p_j} w_i' <^x_{p_j} w_i^1 <^x_{p_j} w_i^2 <^x_{p_j} \ldots <^x_{p_j} w_i^h$, that is, $r_i <^x_{p_j} w_i^h$ for some $h \geq 0$ and some $w_i^1, w_i^2, \ldots w_i^h \in O_w|x$. If w_i^h is followed by a read, r_i', in H_i, then, for all such cases, remove the actions $w_i, w_i', w_i^1, \ldots w_i^h$ from β' to form the cycle $\widehat{\beta}$. Note that $r_i \overset{pcd}{\longmapsto} r_i'$. If w_i^h is followed in β by some $w_i^{h+1} \in O_w$ and $w_i^h \overset{r-prog}{\longrightarrow} w_i^{h+1}$, then, in all such cases, remove all actions $w_i, w_i', w_i^1, \ldots w_i^{h-1}$ from $\widehat{\beta}$ to form the cycle $\bar{\beta}$. Note that $r_i <_{p_i} w_i^h \overset{r-prog}{\longrightarrow} w_i^{h+1}$ and thus $r_i \overset{pcd}{\longmapsto} w_i^{h+1}$. But $\bar{\beta}$ is an extended cycle in $(O, \overset{pcd}{\longmapsto})$ and thus $(O, \overset{pcd}{\longmapsto})$ contains a cycle. ■

Processor Consistency as implemented in the DASH machine is described again in Gharachorloo's PhD thesis [Gha95]. The definition is similar to the two previous papers combined [GLL+90, GGH93], but uses different terminology and is ambiguous. Gharachorloo states, for example, that read actions appear atomic. This might imply that read actions are blocking. He also uses the picture in figure 6.9 to represent a machine that implements PCDash. Each processor in the machine has its own copy of the memory and a write buffer. The picture suggests that a processor updates its own copy of the memory before broadcasting the update to other processes. This requirement is not met by the definition of PCDash.

6.3.3.3 Comparing PCGharachorloo with other memory models

Computation 6 on page 34 was shown to be a coherent computation, but is not PCDash since it is not possible to form a linearization for process q that maintains program order with respect to the two reads by q. Combining this with the following

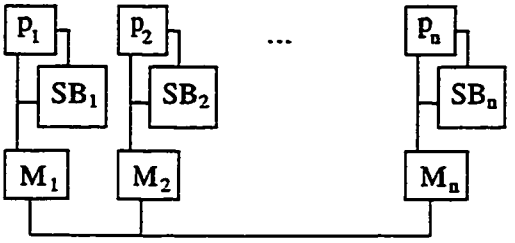


Figure 6.9: *A machine that implements the DASH's processor consistency [Gha95]*

claim, we have that PCDash is strictly stronger than coherence. The proof of the claim is very similar to the proof of claim 6.2.5 on page 57.

Claim 6.3.24 *Any computation that is PCDash is also Coherent.*

Proof: Let C be any PCDash computation on system (P, J) with set of actions O. Take any set of sequences $(O|p \uplus O_w, \overset{view_p}{\longrightarrow})$, one for each $p \in P$, that satisfy definition 6.3.17 on page 93 to build the object sequences S_x, one for each $x \in J$, that contain all the actions to x in O. Initially, $S_x = (O_w|x, \overset{view_p}{\longrightarrow})$ for any process p. (By condition 2 of PCDash, S_x will be the same sequence, independent of the process p chosen). Now, for each $x \in J$ and each $p \in P$, let $S_x^p = ((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \overset{view_p}{\longrightarrow}) = R_0^{x,p}, w_1^{x,p}, R_1^{x,p}, w_2^{x,p}, R_2^{x,p}, \ldots, w_{k_p}^{x,p}, R_{k_p}^{x,p}$, where each $w_i^{x,p}$ is a write action and each $R_i^{x,p}$ is a sequence of read actions. Note that each write action in S_x^p also appears in S_x (but not vice versa), and that these write actions appear in the same order in S_x and S_x^p. For each $0 < i \leq k_p$, insert, in order, $R_i^{x,p}$ into S_x directly behind $w_i^{x,p}$ and insert $R_0^{x,p}$ into S_x, in order, before the first action of S_x.

Clearly, each S_x is a linearization. To show that each S_x also maintains program order, consider any two actions $o_1, o_2 \in (O|p)|x$ such that $o_1 \overset{prog}{\longrightarrow} o_2$. If $o_1, o_2 \in O_w$, then by condition 1 of PCDash, o_1 precedes o_2 in S_x. If $o_1, o_2 \in O_r$, then there are 2 possible cases:

1. If $o_1, o_2 \in R_i^{x,p}$ for some index i then, by condition 3 of PCDash, o_1 precedes o_2 in S_x, otherwise

2. if $o_1 \in R_i^{x,p}$ and $o_2 \in R_j^{x,p}$, for some indices i and j, then necessarily $w_i^{x,p} \xrightarrow{view_p} w_j^{x,p}$. Thus $w_i^{x,p}$ precedes $w_j^{x,p}$ in S_x and by construction of S_x, o_1 precedes o_2 in S_x.

If $o_1 \in O_w$ and $o_2 \in O_r$, let $o_2 \in R_i^{x,p}$. If $o_1 = w_i^{x,p}$ then o_1 precedes o_2 in S_x by construction of S_x. Otherwise, if $w_i^{x,p} \in O|p$ then by condition 3 of PCDash, $w_i^{x,p} \xrightarrow{prog} o_2$. Since $w_i^{x,p}$ is the last write to x by p to precede o_2 and since $o_1 \xrightarrow{prog} o_2$, $o_1 \xrightarrow{prog} w_i^{x,p}$. Hence o_1 precedes $w_i^{x,p}$ in S_x and by construction of S_x, $w_i^{x,p}$ precedes o_2 in S_x. If $w_i^{x,p} \in O|q$ for some process $q \neq p$ then $o_{1_{O|p}} \xrightarrow{view_p} w_i^{x,p} \xrightarrow{view_p} o_2$ by condition 4 of PCDash and because $w_i^{x,p}$ is the last write preceding o_2 in $<_{L_p}$. Thus, $o_1 <_{L_p} w_i^{x,p} <_{L_p} o_2$. Since $<_{L_p}$ and S_x agree on the ordering of writes, o_1 precedes $w_i^{x,p}$ in S_x and thus o_1 precedes o_2 in S_x.

Finally, consider the case when $o_1 \in O_r$ and $o_2 \in O_w$. Since, $o_1 \xrightarrow{prog} o_2$, by condition 3 of PCDash, $o_1 \xrightarrow{view_p} o_{2_{O|p}}$ and by condition 4 of PCDash, $o_1 \xrightarrow{view_p} o_{2_{O|p}} \xrightarrow{view_p} o_{2_{O_w}}$. Let $o_1 \in R_i^{x,p}$. If $w_i^{x,p} \in O|p$ then $w_{i_{O|p}}^{x,p} \xrightarrow{view_p} o_1 \xrightarrow{view_p} o_{2_{O|p}} \xrightarrow{view_p} o_{2_{O_w}}$ and $w_i^{x,p} <_{L_p} o_1 <_{L_p} o_2$. Since $<_{L_p}$ and S_x agree on the ordering of writes to the same location, $w_i^{x,p}$ precedes o_2 in S_x, and by the construction of S_x, o_1 precedes o_2 in S_x.

Hence, each S_x is a linearization that satisfies definition 4.2.2 on page 17. ■

Computation 14 is PCDash but is not PCVax. Notice that none of the reads by p are in the set O_{cache_p} and none of the reads by q are in the set O_{cache_q}, since they are not preceded by other reads to the same object. Thus, by condition 5 of PCVax, $w(x)1_{O_w} \xrightarrow{view_p} r(x)1$ and $w(x)2_{O_w} \xrightarrow{view_q} r(x)2$. Furthermore, by condition 3 of PCVax, both $(O|p \uplus O_w, \xrightarrow{view_p})$ and $(O|q \uplus O_w, \xrightarrow{view_q})$ must agree on the ordering of all

writes from the set O_w. If $w(x)1_{O_w} \xrightarrow{view_p} w(x)2_{O_w}$ and $w(x)1_{O_w} \xrightarrow{view_q} w(x)2_{O_w}$ then, by condition 1 of PCVax, $w(y)0 \xrightarrow{view_q} w(y)1 \xrightarrow{view_q} w(x)1 \xrightarrow{view_q} w(x)2 \xrightarrow{view_q} r(x)2$. By condition 2 of PCVax, $r(x)2 \xrightarrow{view_q} r(y)0$. Hence, $w(y)0 \xrightarrow{view_q} w(y)1 \xrightarrow{view_q} r(y)0$. This clearly does not satisfy condition 6 of PCVax. If $w(x)2_{O_w} \xrightarrow{view_p} w(x)1_{O_w}$ and $w(x)2_{O_w} \xrightarrow{view_p} w(x)1_{O_w}$ then, by a similar argument, $w(z)0 \xrightarrow{view_p} w(z)1 \xrightarrow{view_p} r(z)0$, which, similarly, does not satisfy condition 6 of PCVax. Hence, Computation 14 is not PCVax.

Now consider the following sequences: $\xrightarrow{view_p} = w(y)0_{O|p}\, w(y)0_{O_w}\, w(y)1_{O|p}\, w(y)1_{O_w}$ $w(x)1_{O|p}\ r(x)1\ w(z)0\ r(z)0\ w(z)1\ w(x)2\ w(x)1_{O_w}$, and $\xrightarrow{view_q} = w(z)0_{O|q}\ w(z)0_{O_w}$ $w(z)1_{O|q}\ w(z)1_{O_w}\ w(x)2_{O|p}\ w(x)2_{O_w}\ r(x)2\ w(y)0\ r(y)0\ w(y)1\ w(x)1$. They agree on the ordering of writes to the same object from the set O_w and all writes from O_w maintain program order. Each sequence maintains the relaxed program order and maintains program order with respect to actions to the same object. For each write, the copy from the process set appears before the copy from O_w. Finally, $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p}) = w(y)0\ w(y)1\ w(x)1\ r(x)1\ w(z)0$ $r(z)0\ w(z)1\ w(x)2$ and $((O|q \uplus O_w) \setminus (O_{invisible_q} \cup O_{memupdates_q}), \xrightarrow{view_q}) = w(z)0\ w(z)1$ $w(x)2\ r(x)2\ w(y)0\ r(y)0\ w(y)1\ w(x)1$ are linearizations. The relation $(O, \xmapsto{pcd})$, which is shown in figure 6.10, does not contain a cycle. Hence, computation 14 is PCDash.

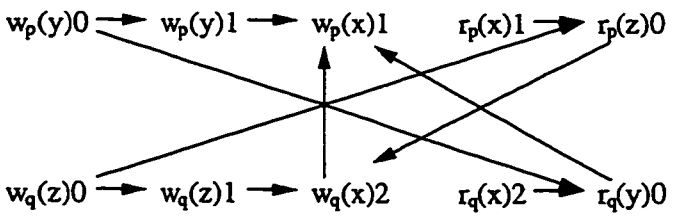


Figure 6.10: *The* $(O, \xmapsto{pcd})$ *relation of Computation 14*

Computation 14 and the following claim show that PCVax is a stronger memory consistency model than PCDash.

Claim 6.3.25 *Any PCVax computation is PCDash.*

Proof: Consider any PCVax computation C containing set of actions O of system (P, J) and choose any set of sequences $(O|p \uplus O_w, \xrightarrow{view_p})$, one for each process in P, that satisfies the definition of PCVax (page 42). It can easily be verified that each $(O|p \uplus O_w, \xrightarrow{view_p})$ satisfies conditions 1 to 5 of PCDash. To show that $(O, \xmapsto{pcd})$ does not contain a cycle, one can build one total order of all actions in the system, such that each write occurs twice, $(O \uplus O_w, \xrightarrow{view})$. The total order $(O \uplus O_w, \xrightarrow{view})$ will be a merge of all the sequences $(O|p \uplus O_w, \xrightarrow{view_p})$. Initially, $\xrightarrow{view} = (O_w, \xrightarrow{view_p}) = w_1, w_2, \ldots, w_k$ for any process p and some $k \geq 0$. By condition 3 of PCVax, $\xrightarrow{view}$ is a subsequence of each $(O|q \uplus O_w, \xrightarrow{view_q})$. Thus, for all processes q, there are some sequences $S_0^q, S_1^q, \ldots, S_k^q$, such that $(O|q \uplus O_w, \xrightarrow{view_q}) = S_0^q, w_1, S_1^q, w_2, S_2^q, \ldots, S_k^q, w_k$. Insert each S_i^q into $\xrightarrow{view}$ anywhere between w_{i-1} and w_i.

Consider any $o_1, o_2 \in O$ such that $o_1 \xmapsto{pcd} o_2$. There are four possible cases.

1. If, for some process $p \in P$, $o_1, o_2 \in O|p$ and $o_1 \xrightarrow{r-prog} o_2$, then, by condition 2 of PCVax, $(o_1, o_2) \in (O, \xrightarrow{view})$, otherwise

2. if $\exists p, q \in P$, where $p \neq q$, and $\exists x \in J$ such that $o_1 \in (O_w|q)|x$, $o_2 \in (O_r|p)|x$ and $o_1 <_{L_p} o_2$ then $o_1 \xrightarrow{view_p} o_2$, and, by construction of $(O \uplus O_w, \xrightarrow{view})$, $o_1 \xrightarrow{view} o_2$, otherwise

3. if $\exists p \in P$ and $\exists x \in J$ such that $o_1, o_2 \in O_w$ and $o_{1_{O_w}} \xrightarrow{view_p} o_{2_{O_w}}$ then, by condition 1 of PCVax, $o_{1_{O_w}} \xrightarrow{view} o_{2_{O_w}}$, otherwise

4. if for some $x \in J$ and some $p \in P$, $o_1 \in (O_r|p)|x$, $o_2 \in O_w|x$ and $\exists o' \in O_w$ such that $o_1 <_{L_p} o' \xrightarrow{r-prog} o_2$, then, by conditions 1 and 5 of PCVax, $o_1 \xrightarrow{view_p} o'_{O_w}$ and, by condition 1 of PCVax, $o'_{O_w} \xrightarrow{view_p} o_2$. Thus, by construction of $(O \uplus O_w, \xrightarrow{view})$, $o_1 \xrightarrow{view} o' \xrightarrow{view} o_2$.

Hence, in all cases, $o_1 \xrightarrow{view} o_2$. Since $(O \uplus O_w, \xrightarrow{view})$ is a total order, $(O, \xmapsto{pcd})$ does not contain a cycle. ■

Computation 9 on page 56 was shown not to be PCGharachorloo, but is PC-Dash since it is PCVax. The following claim shows that PCGharachorloo is strictly stronger than PCDash.

Claim 6.3.26 *Any PCGharachorloo computation is PCDash.*

Proof: Consider any PCGharachorloo computation C with resulting set of actions O on system (P, J). Choose any set of linearizations, containing exactly one linearization for each process in P, that satisfies the definition of PCGharachorloo (page 63). Initially, for each process p, $\xrightarrow{view_p} = (O|p \cup O_w, <_{L_p})$. To construct $(O|p \uplus O_w, \xrightarrow{view_p})$, let the writes that are already in $\xrightarrow{view_p}$ be the writes from O_w and add each write by p from $O|p$ such that it immediately follows the action from $(O|p)$ that it immediately follows in $(O|p, \xrightarrow{prog})$. Condition 1 of PCDash is satisfied by condition 1 of PCGharachorloo. Condition 2 of PCDash is satisfied by condition 2b of PCGharachorloo. In $(O|p, \xrightarrow{view_p})$ the read actions are in program order by condition 1 of PCGharachorloo. Since the write actions are inserted in program order, $(O|p, \xrightarrow{view_p})$ satisfies condition 3 of PCDash.

Assume, to reach a contradiction, that condition 4 of PCDash is not satisfied by $(O|p \uplus O_w, \xrightarrow{view_p})$ for some process p. Consider the first write by p from O_w, w, such

that $w_{O_w} \xrightarrow{view_p} w_{O|p}$. If, for some $r \in O_r|p$, w immediately follows r in $(O|p, \xrightarrow{prog})$, then $r <_{L_p} w$ by condition 1 of PCGharachorloo. Thus, by construction of $(O|p \uplus O_w, \xrightarrow{view_p})$, $r \xrightarrow{view_p} w_{O|p} \xrightarrow{view_p} w_{O_w}$, contradicting that $w_{O_w} \xrightarrow{view_p} w_{O|p}$. If, for some $w' \in O_w|p$, w immediately follows w' in $(O|p, \xrightarrow{prog})$, then $w' <_{L_p} w$ by condition 1 of PCGharachorloo. By the choice of w, $w'_{O|p} \xrightarrow{view_p} w'_{O_w}$ and by construction of $(O|p \uplus O_w, viewp)$, $w'_{O_w} \xrightarrow{view_p} w_{O_w}$ and $w_{O|p}$ immediately follows $w'_{O|p}$ in $(O|p \uplus O_w, \xrightarrow{view_p})$. Hence, $w_{O|p} \xrightarrow{view_p} w_{O_w}$, contradicting that $w_{O_w} \xrightarrow{view_p} w_{O|p}$. Hence, each $(O|p \uplus O_w, \xrightarrow{view_p})$ satisfies condition 4 of PCDash.

To show that condition 5 of PCDash is satisfied, for each $p \in P$, let $L_p = ((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memupdates_p}), \xrightarrow{view_p})$. Consider any $r \in (O_r|p)|x$ for some object x and some process p and let $w \in (O_w|x)$ be the last write to x preceding r in L_p. If $w \in O|p$ then $w_{O|p} \xrightarrow{view_p} r$ implies that $w \xrightarrow{prog} r$. Hence, by condition 2 of PCGharachorloo, $w <_{L_p} r$ and thus $w_{O|p} \xrightarrow{view_p} w_{O_w} \xrightarrow{view_p} r$. By the choice of w, there is no $w' \in (O_w|p)|x$ such that $w_{O|p} \xrightarrow{view_p} w'_{O|p} \xrightarrow{view_p} r$ and hence, no $w' \in (O_w|p)|x$ such that $w_{O_w} \xrightarrow{view_p} w'_{O_w} \xrightarrow{view_p} r$. This also implies that there is no $w' \in (O_w|q)|x$ such that $w_{O_w} \xrightarrow{view_p} w' \xrightarrow{view_p} r$, since such a $w' \notin O_{invisible_p}$. Hence, there is no $w' \in O_w|x$ such that $w <_{L_p} w' <_{L_p} r$ and w is the last write to x preceding r in $<_{L_p}$. Hence, w and r contain the same value.

If $w \in O|q$ for some $q \neq p$, then there is no $w' \in (O_w|p)|x$ such that $w \xrightarrow{view_p} w'_{O|p} \xrightarrow{view_p} r$ by the choice of w. Furthermore, there is no w' such that $w \xrightarrow{view_p} w'_{O_w} \xrightarrow{view_p} r$, since this would imply that $w'_{O|p} \xrightarrow{view_p} w \xrightarrow{view_p} w'_{O_w}$ and $w \in O_{invisible_p}$ and w could not be the last write to x to precede r in L_p. Furthermore, by a similar argument, there is no write, w', to x by some process $s \neq p$ such that $w \xrightarrow{view_p} w' \xrightarrow{view_p} r$, since $w' \notin O_{invisible_p}$. Hence,

w is the last write to x preceding r in $<_{L_p}$ and w and r contain the same value.

Finally, since $(O_w, \overset{view_p}{\longrightarrow}) = (O_w, <_{L_p})$, $(O, \overset{pcd}{\longmapsto}) \subseteq (O, \overset{\widehat{pcd}}{\longmapsto})$. Thus, $(O, \overset{pcd}{\longmapsto})$ is cycle-free by condition 3 of PCGharachorloo. ■

Since SC is stronger than PCGharachorloo, SC is also strictly stronger than PCDash.

Computation 10 on page 73 is PCDash since it was shown to be PCGharachorloo, but is not PCG, nor P-RAM-A (and thus not P-RAM-R nor P-RAM-W).

Computation 3 on page 33 is PCG and P-RAM-A but is not PCDash. In any sequence for p, $w(y)1$ must appear before $r(x)1$ and in any sequence for q $w(y)1$ must appear before $r(y)1$. Since $w(y)1 <_p^y r(y)1 \overset{r-prog}{\longrightarrow} w(x)1 <_q^x r(x)1 \overset{r-prog}{\longrightarrow} w(y)1$ implies that $w(y)1 \overset{pcd}{\longmapsto} r(y)1 \overset{pcd}{\longmapsto} w(x)1 \overset{pcd}{\longmapsto} r(x)1 \overset{pcd}{\longmapsto} w(y)1$, there are no sequences that satisfy PCDash. Hence PCDash and PCG are incomparable and PCDash and P-RAM-A are incomparable.

Computation 5 is P-RAM-W and P-RAM-R, but is not coherent and hence, not PCDash. Thus, PCDash is incomparable with P-RAM-W and P-RAM-R.

Computation 12 on page 84 was shown to be PCAhamad, but is not PCDash. Any sequence for process p must order $w_q(x)2$ before $r_p(x)2$, and hence, necessarily, $w_q(x)2 \overset{pcd}{\longmapsto} r_p(x)2$. Similarly, process r must order the action $w_p(y)4$ before its own action $r_r(y)4$ in its sequence, and hence $w_p(y)4 \overset{pcd}{\longmapsto} r_r(y)4$. Since process s requires that all processes view $w_r(x)1$ before $w_q(x)2$, also $w_r(x)1 \overset{pcd}{\longmapsto} w_q(x)2$ and $w_r(x)1 \overset{pcd}{\longmapsto} w_q(x)2 \overset{pcd}{\longmapsto} r_p(x)2 \overset{r-prog}{\longrightarrow} w_p(y)4 \overset{pcd}{\longmapsto} r_r(y)4 \overset{r-prog}{\longrightarrow} w_r(x)1$. Hence, any sequences satisfying conditions 1 to 6 of PCDash would cause a cycle to appear in $(O, \overset{pcd}{\longmapsto})$ implying that Computation 12 is not PCDash.

Finally, Computation 13 on page 85 was shown not to be PCKohli (and hence not PCAhamad), but is PCDash since it is PCGharachorloo. Hence, PCKohli and PCAhamad are incomparable with PCDash.

Chapter 7

Summary and Concluding Remarks

This thesis provides a framework that facilitates the description, analysis and comparison of memory consistency models; gives an overview of two standard memory consistency models, sequential consistency and coherence; and focuses on two commonly used terms that have various interpretations, pipelined RAM and processor consistency.

The pipelined RAM machine is defined ambiguously by Lipton and Sandberg [LS88]. Ahamad et al. give a formal definition of the memory model that one possible interpretation of the machine implements [ABJ+93]. This thesis presents a precise description of this interpretation of the machine and proves that Ahamad et al.'s definition does indeed capture that machine. Mosberger describes a second possible interpretation of the machine [Mos93]. The precise description of this machine is presented together with the formal definition of the memory model implemented. Furthermore, a third possible interpretation is presented, with the precise description of the machine and the formal definition of the memory consistency model that it implements.

The other memory consistency model, processor consistency, has many more interpretations, and some of them are presented in this thesis. There is one formal interpretation of Goodman's original definition [Goo89] (PCG) [ABJ+93, KNA93] and there are two machines that implement a memory model called processor consistency. Both machines are defined ambiguously [FKH87, GLL+90, GGH93, Gha95].

For the VAX 8800, one possible interpretation is presented in this thesis, stating the assumption made, with the precise machine description (M_{VAX}) and the formal definition of the memory model that is implemented (PCVax).

Processor consistency as implemented in Stanford's DASH machine was originally presented and described in 1990 [GLL+90] by Gharachorloo et al. ($M_{Gharachorloo}$). This thesis shows that Ahamad et al.'s formal definitions of the memory model of processor consistency as implemented in the DASH, based on this early work, (PC-Ahamad and PCKohli) [ABJ+93, KNA93] do not accurately capture $M_{Gharachorloo}$. Furthermore, this thesis supplies a precise definition of $M_{Gharachorloo}$ and a formal definition of the memory model implemented (PCGharachorloo).

In 1993, Gharachorloo et al. presented a correction to their original description of processor consistency as implemented in the DASH machine [GGH93]. A precise description of this machine (M_{DASH}) is presented in this thesis with the formal definition of the memory model that it implements (PCDash). This machine is described again later using different terminology [Gha95]. Some possibly different interpretation of this new description of processor consistency as implemented in the DASH is discussed.

Figure 7.1 summarizes the models described in this thesis and their relationships. An arrow from memory consistency model A to another memory model B, indicates that any system satisfying the constraints of model A will also satisfy the constraints of model B; that is, A is a stronger memory model than B.

Table 7 summarizes the relationships between the computations and models used throughout this thesis.

This thesis illustrates clearly the need for a unifying formal framework to describe

memory consistency models, such as the framework used in this thesis.

However, the framework still has a notable weakness. Each machine in this thesis has a set of constraints that describes the possible executions on the machine. From the machine description, it is generally clear that the machine does indeed satisfy these constraints. However, it is possible that some execution satisfies the constraints of the precise description of the machine, but is not actually possible on the machine itself. I know of no way to verify that the constraints are sufficient to describe any execution on a machine.

Ongoing research, which uses the formal definitions of memory consistency models such as presented in this thesis, is developing basic tools such as mutual exclusion, using a minimum of synchronization constructs for various weak memory consistency models. Future research that could benefit from the work in this thesis includes (semi) automated verification that a machine implements a particular memory consistency model, that two memory consistency models are equivalent or that an algorithm behaves as required when implemented on a multiprocessor machine with a particular memory consistency model. CCS's workbench, which is a verification tool, could be considered as a possible tool for such automated verification.

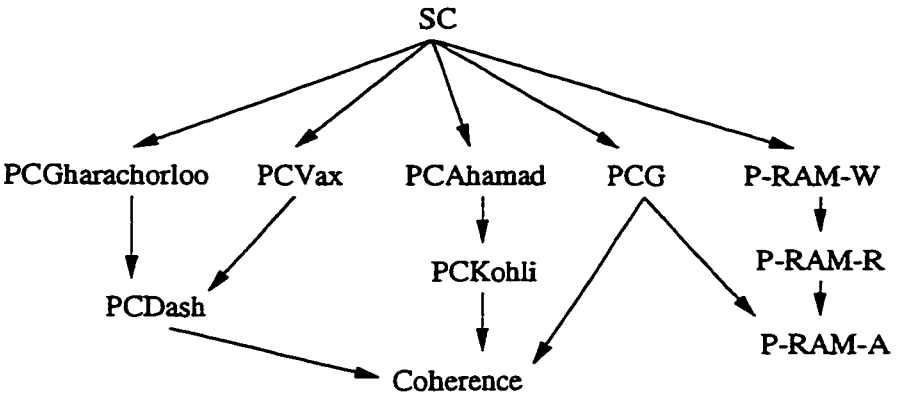


Figure 7.1: *Relationships between memory consistency models*

	SC	coherence	P-RAM-			PC					
			A	R	W	G	Vax	Gharachorloo	Ahamad	Kohli	Dash
Computation 1	√	√	√	√	√	√	√	√	√	√	√
Computation 2		√	√	√	√	√	√	√	√	√	√
Computation 3		√	√			√				√	
Computation 4			√	√							
Computation 5			√	√	√						
Computation 6		√									
Computation 7		√	√	√	√						
Computation 8		√	√	√	√	√		√	√	√	√
Computation 9		√					√				√
Computation 10		√					√	√	√	√	√
Computation 11		√	√	√	√	√					
Computation 12		√	√	√	√	√	√		√	√	
Computation 13		√	√	√	√	√		√			√
Computation 14		√	√	√	√						√

Table 7.1: *Model-computation relationships*

References

[ABJ+93] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Also available as College of Computing, Georgia Institute of Technology technical report GIT-CC-92/34.

[ACFW93] Hagit Attiya, Soma Chaudhuri, Rob Friedman, and Jennifer Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 241–250, 1993.

[Adv96] Sarita V. Adve. Using information from the programmer to implement system optimizations without violating sequential consistency. Technical Report ECE 9603, Department of Electrical and Computer Engineering, Rice University, March 1996.

[AH90a] Sarita V. Adve and Mark D. Hill. Implementing sequential consistency in cache-based systems. *1990 Int'l Conf. on Parallel Processing*, pages I47–I50, August 1990.

[AH90b] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture*, pages 2–14, May 1990.

[AH93] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-

memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, 1993.

[ANB+95] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto. Causal memory: Definitions, implementations, and programming. *Distributed Computing*, 9:37–49, 1995.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. *Proc. of the 13th Annual Int'l Symp. on Computer Architecture*, pages 434–442, June 1986.

[FKH87] J. Fu, J.B. Keller, and K.J. Haduch. Aspects of the VAX 8800 C box design. *Digital Technical Journal*, (4):41–51, February 1987.

[GAG+92] Kourosh Gharachorloo, Sarita Adve, Anoop Gupta, John Hennessy, and Mark Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[GGH93] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Revision to memory consistency and event ordering in scalable shared-memory multiprocessors. Technical Report CSL-TR-93-568, Computer Systems Laboratory, Stanford University, April 1993.

[Gha95] Kourosh Gharachorloo. *Memory consistency models for shared-memory multiprocessors.* PhD thesis, Stanford University, 1995.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event or-

dering in scalable shared-memory multiprocessors. In *Proc. of the 17th Int'l Symp. on Computer Architecture*, pages 15–26, May 1990.

[GM92] Phillip B. Gibbons and Michel Merritt. Specifying nonblocking shared memories. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 306–315, 1992.

[GMG91] Phillip B. Gibbons, Michel Merritt, and Kourish Gharachorloo. Proving sequential consistency of high-performance shared memories. *Symp. on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[Goo89] James Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[HKV97] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems*, pages 349–356, October 1997.

[HKV98] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Weak memory consistency models part I: Definitions and comparisons. Technical Report 98/612/03, Department of Computer Science, The University of Calgary, January 1998. Submitted to IEEE Trans. on Computers.

[HW90] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[JS96] Jerry James and Ambuji Singh. The impact of hardware models on shared memory consistency conditions. *Lecture Notes in Computer Science*, 1119:719–734, 1996.

[KNA93] Prince Kohli, Gil Neiger, and Mustaque Ahamad. A characterization of scalable shared memories. In *Proc. of the 1993 Int'l Conf. on Parallel Processing*, August 1993.

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

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

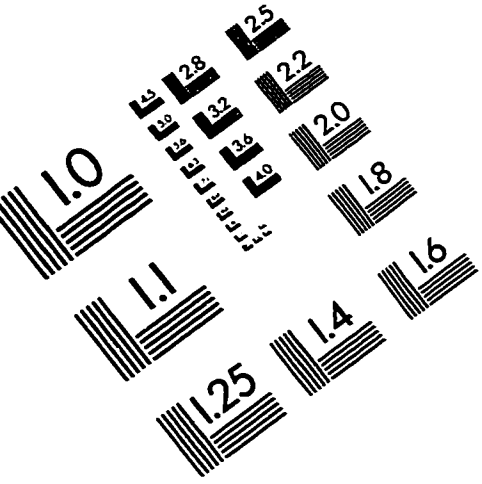
[Lam86] Leslie Lamport. On interprocess communication (parts I and II). *Distributed Computing*, 1(2):77–85 and 86–101, 1986.

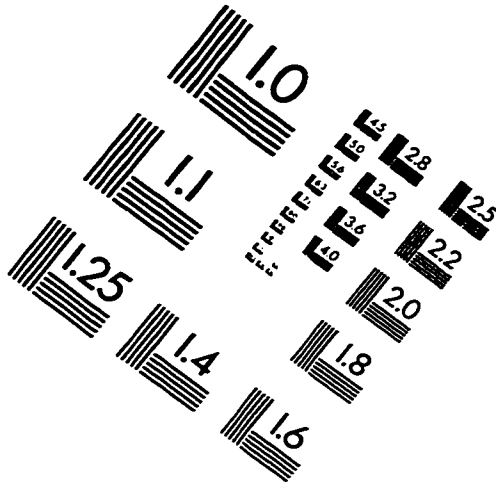
[LS88] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.

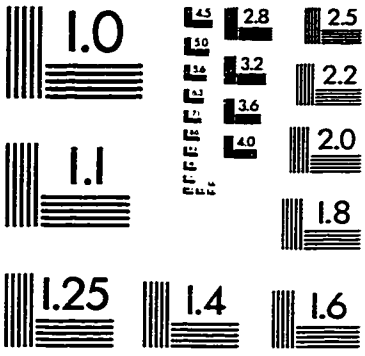
[Mos93] David Mosberger. Memory consistency models. *ACM Operating Systems Review*, 27(1):18–26, January 1993.

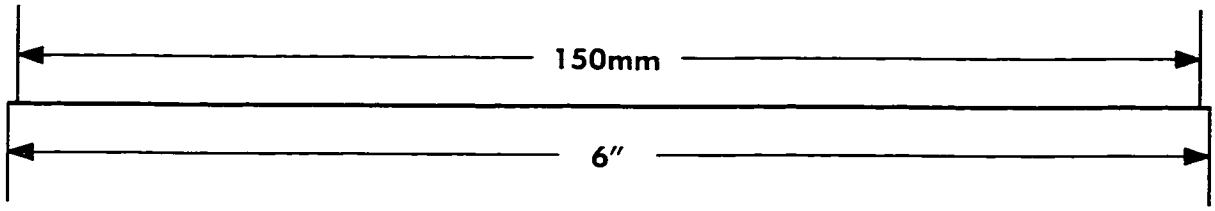
[SFC91] Pradeep Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. Technical Report CSL-91-11, XEROX Corporation Palo Alto Research Center, December 1991.

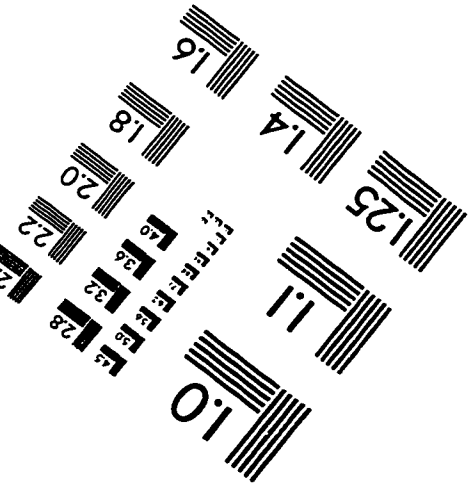
IMAGE EVALUATION TEST TARGET (QA-3)

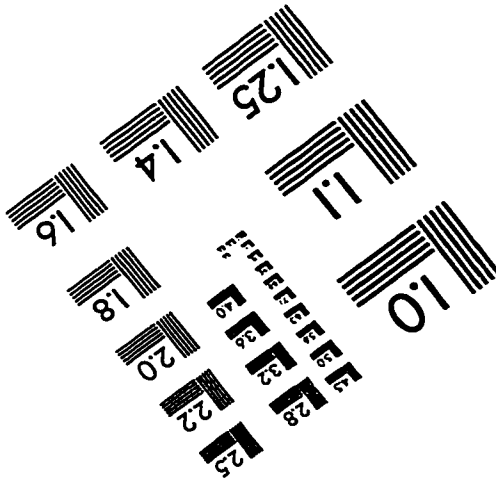












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989