The weakest reasonable memory model

by

Matteo Frigo

Laurea, Università di Padova (1992)
Dottorato di Ricerca, Università di Padova (1996)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

October 1997

Author . . . .
Department of Electrical Engineering and Computer Science
January 28, 1998

Certified by . . . .
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
PUT NAME HERE
Chairman, Departmental Committee on Graduate Students

Abstract

A memory model is some description of how memory behaves in a parallel computer system. While there is consensus that sequential consistency [Lamport 1979] is the strongest memory model, nobody seems to have tried to identify the weakest memory model. This thesis concerns itself with precisely this problem.

We cannot hope to identify the weakest memory model unless we specify a minimal set of properties we want it to obey. In this thesis, we identify five such properties: completeness, monotonicity, constructibility, nondeterminism confinement, and classicality. Constructibility is especially interesting, because a nonconstructible model cannot be implemented exactly, and hence every implementation necessarily supports a stronger model. One nonconstructible model is, for example, dag consistency [Blumofe et al. 1996a].

We argue (with some caveats) that if one wants the five properties, then location consistency is the weakest reasonable memory model. In location consistency, every memory location is serialized, but different locations may be serialized independently. (Location consistency is sometimes called coherence [Hennessy and Patterson 1996], and our location consistency is *not* the model with the same name proposed by Gao and Sarkar [1994].)

We obtain these results within a computation-centric theory of memory models, where memory models are defined independently of scheduling issues and language semantics.

Chapter 1

Introduction

All poets and writers who are
in love with the superlative want
more than they are capable of.

F. W. Nietzsche, *Mixed opinions and maxims*, n. 141

A memory model is some description of how memory behaves in a computer system. On a sequential computer, the memory model is so natural and obvious that many people do not even realize there is one: if you write some value to a memory location, you expect to receive that value if you read that location afterwards. Moreover, by writing, you destroy whatever value was already stored in the location. If you had a computer where writing to a memory location does not destroy the previous value of the location, you would deem the computer buggy and not use it—after all, how could you ever program such a machine?

The reason why the sequential memory model is so simple is that in a sequential computer there is a clear notion of *before* and *after*. The sequential computer executes instructions in the order specified by its program, and, for every pair of instructions, one instruction comes either before or after the other. In other words, instructions are totally ordered. A computer architect may choose to design the machine so that it performs certain instructions in parallel, yet, as long as the computer behaves as if it were sequential and obeys the memory model, the programmer will never notice it.

As soon as we remove the restriction that instructions be totally ordered, we open a huge can of worms. It is no longer clear what occurs before what, and in fact, the very meaning of "before" is nebulous. This situation is precisely what happens in a parallel computer. For example, consider a machine with a single memory location l and two processors (Figure 1-1). Initially, l contains the value 0. The two processors operate independently. The first processor writes 1 to l, and the second processor reads l and then writes 2 to it. Call x the value that the second processor receives when it reads. Nobody would argue that x could be 3, since no processor ever writes 3 to l. In all likelihood, nobody would argue that x

Processor 1	Processor 2
$l \leftarrow 1$	$x \leftarrow l$ $l \leftarrow 2$

Figure 1-1: Simple example of program for two processors. The first processor writes the memory location l, and the second processor first reads and writes the location. Initially, l is 0.

could be 2, either, since 2 is written after the location is read. Yet, it is arguable that x can be either 0 or 1.

The previous example illustrates two important points. First, we must live with the fact that behavior of memory is not fully specified, and if you read a location, more than one value may be legal. Second, we had better be precise in how we design and define the memory model. If the memory model is not designed properly, strange things may happen (for instance, a processor reads a value that nobody writes).

Unlike the sequential case, more than one sensible memory model for a parallel computer exists, and more than one formalism exists to define them. We shall encounter many memory models in the rest of the thesis.

Strong and weak memory models Certain memory models are more restrictive than others, in the sense that they permit a subset of memory behaviors. We say that the model that allows a subset of the behaviors is *stronger*. Intuitively, a memory model in which you can receive anything when you read a memory location is really weak, and probably completely useless (it is a kind of write-only memory). Accept for now this intuitive notion of strength, even though it is inaccurate. (For example, according to our notion of "strong", a model where reads always return 0 would be really strong, since it has only one permissible behavior.) A formal definition of "strong" is provided in Section 2.3.

Traditionally, people have agreed that the strongest reasonable memory model is the ***sequential consistency*** model defined by Lamport [1979]. Curiously, researchers have apparently not asked the question of what *the weakest* reasonable memory model is. This thesis concerns itself with precisely this question. I argue that ***location consistency*** is the weakest reasonable memory model. In location consistency every memory location is serialized, but different locations may be serialized independently.

Why should we care at all about weak memory models, once we have sequential consistency? Unfortunately, strong models have a price. It is generally believed [Hennessy and Patterson 1996] that a sequential consistency model imposes major inefficiencies in an implementation. Consequently, many researchers have tried to relax the requirements of sequential consistency in exchange for better performance and ease of implementation. For example, ***processor consistency*** by Goodman [1989] is a model where every processor can have an independent view of memory, and ***release consistency*** by Gharachorloo et al. [1990] is a model where the memory becomes consistent only when certain synchronizing operations are performed. See [Adve and Gharachorloo 1995] for a good tutorial on this subject. In this thesis, we try to establish limits to this process of "relaxation" of sequential consistency.

Properties of memory models We cannot hope to identify the weakest memory model unless we specify the properties we want it to obey. The absolutely weakest memory model allows memory to return arbitrary values, and is therefore completely useless. This is the sense in which I am trying to identify the weakest *reasonable* memory model, not just the weakest *per se*.

Capturing reasonableness is a tough problem. In this thesis I identify five properties that every reasonable memory model should possess. The properties are discussed in detail in Chapter 3. Three of them have a precise mathematical definition, and the other two are more subjective.

Completeness says that a memory model must define at least one behavior, no matter what the program does. The memory cannot say "oops, I don't know" in response to certain programs.

Monotonicity has to do with the partial order of instructions. Suppose that we have a partially ordered instruction stream, and the model allows a certain behavior. Monotonicity demands that the same behavior be still valid for a subset of the partial order.

Constructibility is a necessary condition for the existence of an online algorithm that maintains a model. In real life, it often happens that an implementation of a model actually maintains a stronger model. It may come as a surprise that, for some models, this situation is unavoidable. There exist models Δ that cannot be implemented exactly, and if one wants Δ, then one must necessarily implement a model that is strictly stronger. In fact, we prove that the weakest constructible model Δ^* that is stronger than Δ exists and is unique. We call Δ^* the ***constructible version*** of Δ. There is no point in adopting a model if we must necessarily implement its constructible version; we should simply adopt the stronger model.

Nondeterminism confinement, unlike the previous properties, is not formally defined, and yet I regard this property as necessary. The basic idea is that the memory model should allow the programmer to confine nondeterminism within certain regions of the program (say, a subroutine). We shall clarify this concept in Chapter 3.

Classicality says that reading the memory should not alter its state, as opposed to *quantum* memory models where observing the memory forces it to behave differently. We shall discuss some curious quantum phenomena in Chapter 3.

The weakest reasonable memory model In the past, there have been at least two proposals of very weak memory models. The first, proposed by Gao and Sarkar [1994], is a model that they called "location consistency" and we shall call "GS-location consistency". The other model is ***dag consistency***, which was introduced by the Cilk group of the MIT Laboratory for Computer Science (including myself) [Blumofe et al. 1996b; Blumofe et al. 1996a]. In this thesis, we show that neither model obeys the five properties I regard as necessary. GS-location consistency does not confine nondeterminism, and dag consistency is not constructible.

Discovering that dag consistency is not constructible was surprising, because dag consistency has been quite a useful model in practice. Students and other users of the Cilk system [Blumofe et al. 1995] have written correct dag-consistent programs *without even knowing* they were using dag consistency. Dag consistency seems to capture a lot of the intuition that Cilk programmers have about the shared memory. In the dag consistency papers, we introduced the BACKER algorithm for maintaining dag consistency [Blumofe et al. 1996b]. (Another algorithm, DAGGER, appears in [Joerg 1996].) We investigated the performance of BACKER both empirically and theoretically [Blumofe et al. 1996a]. Indeed, to the best of my knowledge, BACKER is the only coherence algorithm for which there is any kind of theoretical performance guarantee. The fact that dag consistency is not constructible in no way implies that these results are wrong. Instead, it suggests that there is a model stronger than dag consistency for which these results are still valid. In other words, the BACKER algorithm must do something more than what it was designed for.

In an attempt to better understand dag consistency, in this thesis we define a whole class of dag-consistent models and study their properties. Roughly speaking, the situation of these models is as follows. The weaker models have the same anomaly as GS-location consistency. The stronger models are not constructible. One such model, called NN-dag consistency, is the strongest model in the class. NN-dag consistency is not constructible, but remarkably, this thesis proves that its constructible version is exactly location consistency. Consequently, if you want to implement all the properties of NN-dag consistency, you automatically get location consistency. The proof of this equivalence is a major result of this thesis.

What should we conclude about the weakest reasonable memory model? GS-location consistency and the dag-consistent models are faulty, in one way or another. If one wants all the properties of NN-dag consistency, location consistency is implied in every implementation. Furthermore, as shown by Luchangco [1997], BACKER indeed maintains location consistency, and all the results from [Blumofe et al. 1996b; Blumofe et al. 1996a] apply to location consistency directly. This evidence provides a strong rationale for concluding that location consistency is the weakest reasonable memory model.

There is one caveat to the previous argument. I know of another dag-consistent memory model that is not constructible, and whose constructible version I have not been able to identify. My understanding at this point is that this constructible version is indeed strictly weaker than location consistency, and it obeys all the five properties. Unfortunately, I have not been able to find a simple definition of this model; all I know is that it exists. We shall discuss this situation in Section 4.6.

Computation-centric framework In order to talk about properties of memory models and compare them, we introduce a ***computation-centric*** theory of memory models. We start from a ***computation***, which is an abstract representation of a (parallel) instruction stream, and define memory models in terms of the computation alone. Roughly speaking, a computation consists of all the actions that a program does in response to an input. For example, the program `a = b + c` specifies the actions "read `b`", "read `c`", "compute the sum" and "store the sum into `a`". Implicitly, the program also says that the sum cannot be computed before `b` and `c` have been read, and it does not specify which read operation occurs first. We consider the dependencies specified by a program as part of the computation. In this thesis, I deliberately ignore the problem of how programs map into computations. In general, this mapping can be very complex, depending on the program semantics and on the memory model itself. Programs and languages disappear from the universe of this thesis, and only the computation is left. The computation alone constitutes our starting point.

In contrast, most of the literature about memory models defines them in processor-centric terms [Lamport 1979; Dubois et al. 1986; Adve and Hill 1990; Goodman 1989; Hennessy and Patterson 1996; Adve and Gharachorloo 1995]. These models define how a processor (not the computation) sees the memory. Consequently, a programmer that specifies a computation must also worry about how the computation is scheduled on many processors, since memory semantics may change depending on the schedule. Computation-centric models do not have this problem.

The computation-centric framework is not the ultimate way to define memory models. It is true that we can ignore language semantics and scheduling, but for this very reason, there are subtle effects that the framework does not capture. (See Section 2.1.3 for a discussion of this topic.) Nevertheless, the computation-centric theory has proven to be sufficient to derive all the results we have discussed so far.

Structure of this thesis The rest of the thesis is organized as follows. In Chapter 2 we develop the foundations of the computation-centric framework. We define what a computation and a memory model are, as well as what it means for one memory model to be stronger than another. In the same chapter, we reinterpret sequential consistency, processor consistency, and dag consistency within the computation-centric framework. We also give a definition of location consistency. In Chapter 3, we discuss what properties a memory model should have. We identify five such properties. As we said earlier, some are mathematically well defined, and others try to capture an intuitive notion of reasonableness. In Chapter 4, we define the dag-consistent memory models and investigate their properties in details. We prove the constructibility results, and argue that location consistency is the weakest reasonable memory model. Finally, in Chapter 5, we give some concluding remarks.

Chapter 2

Computation-centric memory models

In this chapter we develop the computation-centric theory of memory models. In this theory, two concepts are especially important: the notion of a ***computation*** and the notion of an ***observer function***.

An execution of a program can be abstracted as a ***computation***, which is just a directed acyclic graph whose concrete meaning is immaterial for our purposes. In Section 2.1 we define computations, and give some example of how real-world programs can be mapped onto computations. I then explain that I believe computation-centric memory models are a good idea, because with them, we can forget about the existence of processors, schedulers, and languages, and because they allow us to discuss memory semantics in a simple abstract way. In the rest of the thesis, I am concerned with how a computation, not a processor, sees memory.

In Section 2.2 we define ***observer functions***. For any computation node and memory location, an observer function points to some node that writes to that location. On one side, observer functions are just a technical device that simplifies our notations and allows us to ignore the concrete values of memory locations: we just say that reading a location returns whatever the node specified by the observer function for that location writes. In this way, we forget about read operations altogether. On the other side, observer functions also give memory semantics to computation nodes that *do not* read or write to memory. This property will turn out to be important in the rest of the discussion, when we discuss nondeterminism in Section 3.4.

Since an observer function specifies the memory behavior uniquely, it follows that a memory model is fully characterized by the set of observer functions it allows. In other words, a memory model *is* a set of observer functions. Modulo technicalities, this is the way in which we define memory models in Section 2.3.

We next show how our framework can be applied to the definition of models that previously appeared in the literature, such as sequential consistency, dag consistency, and processor consistency. We also define ***location consistency***, a memory model that behaves as if each memory location were serialized. Location consistency is the main focus of this thesis, since I am proposing it as the weakest reasonable memory model.

2.1 Computations

In this section, we define computations. We first state the definition (Definition 1), and give some examples (Section 2.1.1). We then define the memory operations that a computation is allowed to perform (Section 2.1.2). Finally, I discuss the advantages and the limits of the computation-centric approach.

We start with the definition of a computation, and a few related notions.

Definition 1 *A **computation** is a (finite) directed acyclic graph (**dag**) $G = (V, E)$.*

If there is a path of nonzero length in the dag from node u to node v, we say that u ***precedes*** v, and we write $u \prec v$. The notation $u \prec_G v$ is used whenever we need to be precise about which dag G we are talking about. Observe that this definition of precedence is strict: a node does not precede itself. Whenever we want a non strict definition of precedence, we explicitly use the notation $u \preceq v$, meaning that $u \prec v$ or $u = v$. For the rest of the thesis, remember that precedence is the only order relation that is strict by default. Everything else (e.g., set inclusion, relative strength of memory models, etc.) is not strict.

2.1.1 What is a computation?

The reader might now wonder what Definition 1 has to do real computers that execute real programs (or idealized computers, for example a Turing machine, executing idealized programs).

In order to understand this notion of computation, imagine having a sequential computer executing a program. To fix our ideas, let's say that the computer executes a quicksort program. Recall that quicksort is a divide-and-conquer algorithm that partitions an array into two "halves" and calls itself recursively on each half.

The execution of the program generates a stream of instructions, which may be much longer than the program itself. In the example, since quicksort is a recursive algorithm, the same line of code can be instantiated (i.e., executed) many times during the execution. We say that each of these instantiations is a node of the computation. I remark again that there is a node for every *instantiation* of an instruction, and not for every instruction in the executable program. If an instruction is executed twice (for instance, because belongs to a loop), we put two nodes in the computation.

Without too much effort, we can identify dependencies among computation nodes. In the quicksort example, the partition of the input array must be performed before quicksort recurses on the two halves. Therefore, we can say that the partition *precedes* the recursive calls. Can we also say that the first recursive call precedes the other? In this case, the computation is in the eye of the beholder. You can either argue that the program is sequential and therefore the second recursive call follows the first, or you can say that there is no logical dependency, and that the first call is first only because programs are one-dimensional strings.[1] It is up to you to decide what computation your program is performing. In either case, when the recursion is completely unfolded, we end up with a set of nodes representing all the instructions that the machine executed, and a set of dependencies among those nodes.

We now show another example of computation. Cilk [Blumofe et al. 1995] is a multithreaded language and runtime system. The execution of a Cilk program generates many *threads* that obey certain control dependencies. The graph of threads and of their dependencies is called a *multithreaded computation*. A multithreaded computation is a computation, according to our definition. Unlike the previous example, however, in Cilk there is no subjective interpretation of a computation, because Cilk provides keywords that specify whether there is a dependence or not. More precisely, two threads depend on each other unless the keyword `spawn` is used in the program. (In this sense, Cilk is a computation-friendly version of the C language.)

A third case of computation can be identified in a machine with P processors and a shared memory. Each processor executes its own program, and we assume that there are no synchronization operations in the machine: processors can only communicate by reading and writing into memory. We can construct a computation that models the behavior of the machine. The computation nodes are the machine instructions. The edges are the dependencies imposed by the program order. If instruction A comes after instruction B in the program, then there is an edge from B to A.

The situation becomes much more complicated if we allow the processors to perform explicit synchronization operations (such as barriers and/or mutual exclusion). In this case, it is not clear what the computation is. Specifying the behavior of this system is a tough problem, and I am not trying to solve it in this thesis. We simply assume that the computation is given.

2.1.2 Memory operations

In this section, we state precisely what a computation can do with the shared memory.

The only memory operations that we allow a computation to perform are ***read*** and ***write***. We do not care about the other operations that the dag performs (e.g., additions, etc.), and regard all these things as no-op. We do not allow a computation to perform synchronizing memory operations (for instance, atomic test-and-set or read-modify-write operations,

[1] For all "reasonable" programming languages. In fact, there exists a toy two-dimensional programming language called `orthagonal` (sic). An `orthagonal` program is laid out on a grid, and the `orthagonal` computer is a stack-based machine with a two dimensional program counter. The program counter can move in four different directions, and the machine has special instructions to turn left, right, or backwards, as well as to specify absolute directions. As you would imagine, looping constructs in this language are *really* elegant; you can literally see the loop on the grid. `orthagonal` was written by Jeff Epler, and is part of the Computer Retromuseum maintained by Eric S. Raymond at `ftp://ftp.ccil.org/pub/retro/`. The author of the program does not care about being given credit: "In fact, don't even bother keeping my name on this. It'll help free me from blame."

memory fences, and so on).

> *Rationale:* The lack of synchronizing memory operations is indeed a key aspect of our model. My point of view is that a computation already defines some synchronization, in the form of edges. I want to be able to use that synchronization, and no more, for defining memory semantics. The technicalities that allow us to accomplish this goal are explained in Section 2.2.

Before continuing, we need a couple of technical assumptions.

Assumption 1 *A computation node can perform at most one memory operation, that is, a **read** or a **write**. If a node does not perform any memory operation, we say it performs a **no-op**.*

> *Rationale:* By assuming that every node performs at most one memory operation, we can unambiguously speak of *the* memory operation performed by that node. The assumption simplifies the notation, without loss of generality.

Assumption 2 *Every computation G contains a special **initial node** $initial(G)$ that writes some initial value to all memory locations. The initial node is a predecessor of all nodes in G.*

> *Rationale:* We must be able to specify a value received by a read operation even if there are no writes in the computation. One way to achieve this effect is to say that the read receives the bottom value $\perp$ ("undefined"). This solution would force us to consider values, however, while our theory does not otherwise deal with memory values.
>
> It is not really important whether there is a set of initial nodes writing to all locations, or just one node. In the latter case, as we assume here, the initial node is exempt from Assumption 1.

The smallest computation is the ***empty computation*** ε that consists of the initial node alone.

Notation for memory operations We now introduce the notation for read and write operations. First, we define memory.

Definition 2 *A **memory** $\mathcal{M} = \{l\}$ is a set of **locations**.*

In this thesis I do not care about the set of values that a memory location can contain. Moreover, memory is never explicitly mentioned in the definitions and theorems that follow; it is a kind of parameter to the whole thesis. (Pretend there is a big "for any memory $\mathcal{M}$" on the cover page of this thesis.)

The notation for read and write operations is given in the form of a ***read predicate*** and a ***write predicate***.

$$\begin{aligned} \mathrm{R}(l, u, x) &\triangleq \text{“node } u \text{ reads location } l \text{ and receives the value } x\text{”} \\ \mathrm{W}(l, u, x) &\triangleq \text{“node } u \text{ writes the value } x \text{ to location } l\text{”} \end{aligned}$$

Observe that, because of Assumption 1, for a given node u, the write predicate $\mathrm{W}(l, u, x)$ can be true for at most one location l and value x.

Whenever the value x is immaterial (that is, almost always in this thesis), we use an abbreviated notation.

$$\begin{aligned} \mathrm{R}(l, u) &\triangleq \text{“node } u \text{ reads location } l\text{”} \\ \mathrm{W}(l, u) &\triangleq \text{“node } u \text{ writes to location } l\text{”} \end{aligned}$$

2.1.3 Why computations?

Most of the literature defines memory semantics in processor-centric terms. Instead, I use a computation-centric framework, because I think it is much simpler. In this section, I describe why I think the processor-centric approach is too complicated. We also point out some of the limits of the computation-centric approach. (Simplicity does not come for free, unfortunately.)

Most memory models are processor-centric [Lamport 1979; Dubois et al. 1986; Adve and Hill 1990; Goodman 1989; Hennessy and Patterson 1996; Adve and Gharachorloo 1995]. On the contrary, I believe that processor-centric memory models are too difficult to reason about. They are too difficult even to *define* properly. Consider, for example, the complicated definition of “performing a memory request” in [Gharachorloo et al. 1990] quoting [Dubois et al. 1986; Scheurich and Dubois 1987].

> A LOAD by P_i is considered *performed with respect to* P_k at a point in time when the issuing of a STORE to the same address by P_k cannot affect the value returned by the LOAD. A STORE by P_i is considered *performed with respect to* P_k at a point in time when an issued LOAD to the same address by P_k returns the value defined by this store (or a subsequent STORE to the same location).

The problem with this definition is that the very notion of “subsequent” is what the memory model is supposed to define. It is not clear at all whether “subsequent” means “issued by a processor at a later point in time” or “served by a memory module at a later point in time”. In any case, there is a hidden assumption of some universal clock by which events can be globally ordered.

As another example, Hennessy and Patterson [1996, Page 656] state the following nebulous condition as part of their “definition” of coherence.

> A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated and no other writes to X occur between the two accesses.

Even ignoring the semantics of "sufficiently separated", I still have no idea of the meaning of "follows" and "occurs between". Again, this is what the memory model was supposed to tell us.

The problem with these definitions, in my eyes, is that an operational model of a machine is needed in order to give meaning to the definition. The operational behavior of the machine, however, is precisely what the the definition is supposed to specify. It comes as no surprise, if this is the case, that things quickly become very complicated.

On the contrary, Lamport's definition of sequential consistency [Lamport 1979] is often cited, with good reasons, for its clarity and precision:

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

Lamport's criterion does not assume any global time. The system is sequentially consistent if one can get the same result on one processor, in the way stated by the definition. Observe that Lamport's definition talks about processors only incidentally; the real point is that there are P threads of control. It does not matter whether there are P, $2P$ or $P/2$ processors. In this thesis, instead of extending Lamport's model by giving more capabilities to processors, we give more capabilities to the "threads of control". We allow an arbitrary number of these threads, and we allow arbitrary dependencies (not just P sequential chains).

There are two things that I do not like in processor-centric models. (The following criticisms do not apply to Lamport's definition.) First, some notion of *time* is implicitly assumed, so that it can be said whether one event happened before another. Programmers do not have control over time, however, since a processor can employ a variable amount of time to execute a certain operation. The second thing I do not like is that a computation executed within one processor has certain semantics, and a computation executed by more than one processor has *different* semantics. When I write a program, however, I would rather ignore how many processors my program will run on. Consequently, I must ignore how my program is mapped onto processors, since there is now way to know it if the number of processors is not specified. In other words, I claim that a computation should have memory semantics that do not depend on the schedule. Whether the computation runs on one or more processors, and how it is mapped to processors are not things of which programmers should be aware. (Programmers might want to deal with these issues for performance reasons, but this is another story.)

Computation-centric memory models are a way to deal with these two issues, as will become clear in the rest of the chapter.

Processor 1	Processor 2
$x \leftarrow 1$ $y \leftarrow 1$	$x_1 \leftarrow x$ $y_1 \leftarrow y$ if $x_1 = 0$ and $y_1 = 1$ then $\quad z \leftarrow 1$

Figure 2-1: Example illustrating that the mapping of programs into computations depends on the memory model. In the example, there are three memory locations, x, y, and z, and all three locations are initially 0.

It should be remarked that the framework I propose is not the only alternative to processor-centric models. Memory semantics can (and should) be incorporated into language semantics (as in the λ_S-calculus by Arvind et al. [1996]). In this thesis I am completely ignoring the fundamental issue of how a computation is generated by a program. I acknowledge that this point of my theory is weak. The computation generated by a program indeed may depend on memory semantics, and separation of the two issues is not justified. While ultimately a unified semantics of languages and memory is desirable, I believe that defining the behavior of memory is already complicated enough by itself, that for now linguistic issues should be kept orthogonal as much as possible.

In conclusion, here is the situation as I see it. Processor-centric models are too complex. Language-centric models must take into account the language semantics, which is already a vast subject in itself, most of which has nothing to do with memory. Computation-centric models are simple, and they provide the abstraction of language-centric models without the linguistic issues. Computation-centric models can be used to understand abstract properties of memory models, as we will see in Chapter 3. The computation-centric view is also supported by Gao and Sarkar, who, in a recent paper [Gao and Sarkar 1997], use an end-to-end argument to suggest that memory models should be defined in terms of a partial order.

Limits of the computation-centric approach Despite the above discussion, I do not want the reader to get the idea that computation-centric memory models are the ultimate solution to all problems. The major strength of the computation-centric framework is that it abstracts away from processors and linguistic issues. This is also its major weakness, because the computation may depend on the memory model itself.

Consider Figure 2-1. Suppose that the memory model dictates that writes issued by Processor 1 be observed in the same order by Processor 2. In this case, the instruction $z \leftarrow 1$ is never executed, and is therefore not part of the computation. Suppose now that Processor 2 can observe writes in a different order than they are issued. In this case, the instruction $z \leftarrow 1$ might be executed. The point is that we cannot tell *a priori* what the computation is.

Of course, one can construct many examples like this. A particularly beautiful one was shown to me by Bert Halstead [Halstead 1997]. I leave it (Figure 2-2) as a puzzle for the

Processor 1	Processor 2
if $y = 1$ then $x \leftarrow 1$	if $x = 1$ then $y \leftarrow 1$

Figure 2-2: A puzzle illustrating how memory and language semantics are intimately linked. The two memory locations x and y are initially 0, and there are two processors executing the program shown in the figure. Can x and y both be 1 at the end of the execution? How do you define a memory model in such a way that this situation does not occur?

reader.

2.2 Observer functions

In this section we define the notion of an ***observer function***, which is the second key concept in the computation-centric theory. We first motivate observer functions, then give the definition, and finally explain what the definition means.

In trying to define memory semantics, I have two goals in mind. First, I want to ignore memory values and instead specify where a given value comes from. In other words, our answer to the question, "What does a certain read return?" is of the form, "It returns what a certain node writes, it does not matter what" (where the focus is on the writer rather than the value). Second, I want to specify the memory semantics of nodes that do not perform any memory operation, because these nodes might represent some synchronization in the computation. We have no explicit synchronization operations in our model, and therefore it is important to be able to exploit the synchronization given by the dag for defining memory semantics. These two goals are both accomplished by using an ***observer function***.

Definition 3 *Let $G = (V, E)$ be a computation. An* ***observer function*** *is a function $\Phi : \mathcal{M} \times V \mapsto V$ such that, for all locations $l \in \mathcal{M}$ and all nodes $u \in V$, we have that*

3.1. $W(l, \Phi(l, u))$ *;*

3.2. $u \not\prec \Phi(l, u)$ *.*

The first property says that an observer function must point to a node that writes to memory, that is, $\Phi(l, u)$ writes to l. The second property says that an observer function cannot point to a successor node, that is, $\Phi(l, u)$ does not follow u.

In order to understand why we require these two properties, we must first explain what an observer function means. The main purpose of an observer function is to specify the semantics of read operations. Suppose node u reads location l. Then, the read operation returns whatever value is written to l by node $\Phi(l, u)$. We formalize this idea in the next postulate.

Postulate 4 (Standard semantics of read operations) *Let G be a computation, and let Φ be an observer function for G. The read predicate satisfies this property: If $R(l,u,x)$, then $W(l,\Phi(l,u),x)$.*

Postulate 4 says that if node u reads location l, receiving the value x, then x is indeed the value written to l by node $\Phi(l,u)$. The reader can now happily forget about reads, memory values, and Postulate 4, since observer functions suffice for the rest of the thesis. We only use Postulate 4 when arguing the equivalence between models that are expressed in terms of observer functions and models that are not.

We can now justify the two properties in Definition 3. The first property says that an observer function must point to a node that writes to memory. This constraint is necessary, since otherwise the semantics of read operations would be ill defined. The second property says that an observer function cannot point to a successor node, that is, a node cannot read something written in the future. With current technology, this assumption carries no loss of generality.

Empty computation We remark that the only observer function for the empty computation ε is, trivially, $\Phi_\varepsilon(l, \mathit{initial}(\varepsilon)) = \mathit{initial}(\varepsilon)$, for all locations l.

2.3 Memory models

In this section we define a ***memory model*** as a set of pairs, each consisting of a computation and an observer function defined over that computation. We then define what it means for one model to be stronger than another.

Recall that an observer function completely defines the semantics of read operations. Thus, a set of observer functions defines a memory semantics. We would like just to say: "A memory model is a set of observer functions." Unfortunately, an observer function is defined only for one computation. In order to consider observer functions of many computations, we keep track of the computation by encapsulating both the observer function and the computation in a pair. (We also require that the empty dag ε and its observer function Φ_ε belong to all memory models in order to make boundary cases in proofs easier.)

Definition 5 (Memory model) *A* ***memory model*** *Δ is a set*

$$\Delta = \{(G,\Phi) : G \textit{ is a computation and } \Phi \textit{ is an observer function for } G\} \cup (\varepsilon, \Phi_\varepsilon) .$$

Before we see some examples of memory models in Section 2.4, we first discuss briefly what it means for one model to be stronger than another.

Definition 6 *A memory model Δ is* ***stronger*** *than a model Δ' if $\Delta \subseteq \Delta'$. A memory model Δ is* ***weaker*** *than a model Δ' if Δ' is stronger than Δ.*

Notice that we say that the *subset* is stronger, not the superset, because the subset enjoys more properties. Consequently, the empty set is the strongest memory model, and the set of all (dags, observer functions) is the weakest. (The empty set vacuously enjoys all properties.) One model is not necessarily stronger or weaker than another; if this is the case, we say that the two models are ***incomparable***. If two models are the same set, we also say they are ***equivalent***.

We remark that the usual set operations of union and intersection can be applied to memory models. In this way, new models can be constructed by combining old models. For example, the intersection of two models is a model stronger than both. In this thesis, we do not play with models in this way, with one important exception: we consider infinite unions of memory models in Section 3.3 in order to define the constructible version of a model.

2.4 Example memory models

In this section, we give some examples of memory models. For now, we do not introduce any new models that do not already appear in the literature. We start with ***sequential consistency***, which is usually agreed to be the strongest reasonable memory model. We then consider ***location consistency*** and ***dag consistency***. Finally, for completeness, we also suggest how other models could be defined, although they are not relevant to the later results of this thesis.

2.4.1 Sequential consistency

Sequential consistency [Lamport 1979] is generally considered to be the strongest memory model. It is not the strongest in an absolute sense, just the "strongest reasonable". There exist in principle stronger models, but they are too strong for practical use.[2] We now give a computation-centric definition of sequential consistency. The definition we give is equivalent to Lamport's definition (quoted in Section 2.1.3) for the case where the computation consists of P chains of nodes, but our new definition is more general in that it applies to all computations.

Lamport's criterion for sequential consistency says that one should look at the parallel (processor-centric) execution of a program. If one can find a *sequential* execution of the parallel program that respects program order and gives the same results for all memory accesses, then the memory is sequentially consistent. There are, however, two distinct issues in Lamport's definition. The first one is clearly stated by Lamport: it must be possible to identify a global interleaving of the parallel program (the sequential execution). The

[2]For example, suppose the system behaves as if Processor 1 alone executes its program, then Processor 2 executes its own program, etc. This processor-centric memory model is stronger than sequential consistency, but there seems to be no way to implement this model without using only one processor at a time, which limits its interest.

second issue is not explicitly addressed by Lamport's definition: what is the the memory semantics of a sequential program? In order to define sequential consistency within our framework, we first need to address this problem.

A sequential program, in the processor-centric world, has a very natural memory semantics: reading a location returns the last value written to that location. The notion of "last value" is well defined because the program is sequential. Consider now our computation-centric framework. By superimposing a total order (topological sort) to a computation, the notion of the last writer preceding a given node is well defined in our framework, too. (We define the last writer and not the last value because our observer function framework frees us from dealing with values.)

Definition 7 *Let $G = (V, E)$ be a computation, and let $\mathcal{T}$ be a topological sort of G. The **last writer** according to $\mathcal{T}$ is the function $\mathcal{L}_{\mathcal{T}} : \mathcal{M} \times V \mapsto V$ such that, for all nodes $w \in V$ and all locations $l \in \mathcal{M}$, we have*

1. *$W(l, \mathcal{L}_{\mathcal{T}}(l, w))$;*
2. *$\mathcal{L}_{\mathcal{T}}(l, w) \preceq_{\mathcal{T}} w$;*
3. *for all nodes $v \in V$ such that $\mathcal{L}_{\mathcal{T}}(l, w) \prec_{\mathcal{T}} v \preceq_{\mathcal{T}} w$, the predicate $W(l, v)$ is false.*

The interpretation of the last writer function is that $\mathcal{L}_{\mathcal{T}}(l, w)$ is a node u that writes to l and comes before w in the given topological sort $\mathcal{T}$, and no writes to l occur between u and w in the topological sort. Since we assume that the initial node writes to all locations, it follows that $\mathcal{L}_{\mathcal{T}}(l, w)$ is indeed well defined. Observe also that, if node w writes to l, then $\mathcal{L}_{\mathcal{T}}(l, w) = w$.

The last writer function turns out to be an observer function, because of the way it is constructed.

Lemma 8 *Let G be a computation, and let $\mathcal{T}$ be a topological sort of G. Then $\mathcal{L}_{\mathcal{T}}$ is an observer function.*

Proof: The first property of Definition 3 is the same as the first property of Definition 7. The second property of Definition 3 is true because $\mathcal{T}$ is a topological sort of G, which implies that if $\mathcal{L}_{\mathcal{T}}(l, w) \preceq_{\mathcal{T}} w$, then $w \not\prec_G \mathcal{L}_{\mathcal{T}}(l, w)$. ■

Sequential consistency is therefore that model in which one can find a global topological sort of the computation such that all reads return the last value written to the location being read. Equivalently, a sequentially consistent observer function is a "last writer".

Definition 9 ***Sequential consistency** is the memory model*

$$SC = \{(G, \mathcal{L}_{\mathcal{T}}) : \mathcal{T} \text{ is a topological sort of } G\} \ .$$

2.4.2 Location consistency

Instead of insisting that there exist a single global order for all memory operations (as in SC), we could allow a different topological sort for every location. We call ***location consistency*** the memory model that behaves in this way. In this section, we first give the definition, and then explain why we call it "location consistency". The naming issue is intricate because I) location consistency is sometimes called "coherence", and II) there already exists a completely different memory model called "location consistency".

Definition 10 ***Location consistency*** *is the memory model*

$$LC = \left\{(G, \Phi) : \Phi(l, u) = \mathcal{L}_{\mathcal{T}(l)}(l, u) \text{ and } \mathcal{T}(l) \text{ is a topological sort of } G\right\} .$$

In words, here is what the definition means. An observer function is location consistent if, for every location l, one can find a topological sort $\mathcal{T}(l)$ of the dag, and the observer function for l coincides with the last writer according to $\mathcal{T}(l)$.

Location consistency is the model I propose as the weakest reasonable memory model, for reasons that will be clear after Section 4.5.

On the name "location consistency" We now proceed to the issue of names, which is a tough one. I want to name this model "location consistency", although the model has been called "coherence", and although "location consistency" is already the name of a different model. The reader not interested in academic arguments can safely skip to the next section.

Researchers are usually careful in how they name their intellectual creatures. Names are not just accidental properties of objects. Well-chosen names will help generations of future students to create a mental framework in which to understand things. An example of the bad things that happen when concepts are misnamed comes from the uncountable number of books that have been written to clarify the confusion between Aristotle's first and second kind of substance. (See, for example, [Boethius 512]. The issue is now out of fashion, which is another way to solve it.)

Unfortunately, it is not always possible to choose names that reflect both current and future understanding. For example, the name "dynamic programming" was a reasonable name when Bellman [1957] introduced it, but it makes little sense in our days. In this case, posterior researchers, having a better understanding, have the duty to change the name before it is too late. (For dynamic programming it is probably already too late.) For the small piece of world that this thesis is concerned with, I will try to be a conscious researcher and come up with good names, at the cost of overriding previous choices by other people.

Hennessy and Patterson [1996], when they discuss memory models, distinguish between ***coherence*** and ***consistency***. Coherence means that each location is serialized, while consistency refers to additional constraints that are preserved across different locations. Coherence is usually assumed when memory models are defined in processor-centric terms. On the other hand, "X-consistency" is the canonical form of the names of memory models (e.g., sequential consistency [Lamport 1979], processor consistency [Goodman 1989], release consistency [Gharachorloo et al. 1990], entry consistency [Bershad et al. 1993], scope consistency [Iftode et al. 1996]).

I see no reason to use two separate names. "Coherence" is a legitimate memory model, and should be called X-consistency, for some X. Since its distinguishing feature is that consistency occurs on a per-location basis, we call it location consistency.

The very name "location consistency" has been used by Gao and Sarkar [1994] to describe a memory model different from LC. It is our obligation to explain why we are reusing the same name.

First, I believe that "location consistency" is misnamed for the model of Gao and Sarkar. As explicitly stated by the authors, their model does not require that a single location be serialized. In other words, their model allows "location inconsistency". Second, Gao and Sarkar's model appears to have anomalies that are undesirable for a programmer, and I think it should not be adopted. (See Section 3.4.)

I want to point out that the above remarks do not alter the significance of the technical content of [Gao and Sarkar 1994]. Indeed, that paper contains many ideas that I strongly support. Gao and Sarkar are also looking for a very relaxed memory model, and define the model in terms of a graph of dependencies (although with a processor-centric flavor). Indeed, Gao and Sarkar's model appears to be a kind of dag consistency, as we shall see in the Section 2.4.3 below.

2.4.3 Dag consistency

In the past, with other people from the MIT Laboratory for Computer science, I proposed ***dag consistency*** [Blumofe et al. 1996b; Blumofe et al. 1996a] as a very relaxed memory model for the Cilk [Blumofe et al. 1995] system of parallel computing. Indeed, we published two different models, both called dag consistency. Unfortunately, both turned out to be "unreasonable". We now discuss these two models. First, we give the two definitions, and explain what they mean. Then, we restate both definitions in terms of computations and observer functions.

The first definition of dag consistency appeared in Joerg's thesis [Joerg 1996] and in [Blumofe et al. 1996b].[3]

Definition 11 *The shared memory $\mathcal{M}$ of a multithreaded computation $G = (V, E)$ is **dag consistent** if the following two conditions hold.*

11.1. Whenever any node $u \in V$ reads any location $l \in \mathcal{M}$, it receives a value x written by some node $v \in V$ such that $u \not\prec v$.

[3] Modulo alpha-conversion. I changed the definition a little to make it consistent with our notations, and to avoid introducing new notation.

11.2. For any three nodes $u, v, w \in V$, satisfying $u \prec v \prec w$, if v writes some location $l \in \mathcal{M}$ and w reads l, then w does not receive a value written by u.

The second definition of dag consistency appeared in [Blumofe et al. 1996a].[4]

Definition 12 *The shared memory $\mathcal{M}$ of a multithreaded computation $G = (V, E)$ is* ***dag consistent*** *if there exists a function $\Phi : \mathcal{M} \times V \mapsto V$ such that the following conditions hold.*

12.1. For all nodes $u \in V$, the node $\Phi(l, u)$ writes to l.

12.2. If a node u writes to l, then we have $\Phi(l, u) = u$.

12.3. If a node u reads l, it receives the value written by $\Phi(l, u)$.

12.4. For all nodes $u \in V$, we have that $u \not\prec \Phi(l, u)$.

12.5. For each triple u, v, and w of nodes such that $u \prec v \prec w$, if $\Phi(l, v) \neq u$ holds, then we have $\Phi(l, w) \neq u$.

We now explain what the two definitions mean, and why there are two definitions. (The two definitions indeed define two different memory models. The second model is strictly stronger than the first, as we shall see in Section 4.2).

Explanation of Definition 11 The first definition of dag consistency is trying to characterize a sort of "per-node sequential consistency". Sequential consistency demands a single topological sort valid for all nodes and locations. Location consistency demands a topological sort valid for all nodes, but each location can have a different topological sort. Dag consistency, instead, allows *each node* to "see" a different topological sort of the computation. The only requirement of dag consistency, therefore, is that a node w not see a write by u if there is another write by a node v that lies in the path from u to w (since u cannot otherwise be the last writer before w in any topological sort of the computation). This requirement is precisely what the definition mandates. Notice that Definition 11 is computation-centric in that it gives semantics to a computation, but it does not use any observer function.

Definition 11 is the model by Gao and Sarkar [1994] Albeit defined in different contexts, Definition 11 defines the same model as GS-location consistency. This model says that a read operation can receive any element of a set of "most recent writes". This set is maintained during the execution. It is initially empty, and a write operation added to the set removes all predecessor writes, where "predecessor" is defined by certain synchronizing operations that the model allows. As it can be seen, the set of "most recent writes"

[4] Again, we adapted the definition to our notations, for consistency.

contains precisely those elements that could be seen by a node according to Definition 11. Historically, we did not realize this equivalence when we published [Blumofe et al. 1996b], although we were aware of Gao and Sarkar's work. The equivalence of Definition 11 and GS-location consistency is also pointed out in [Gao and Sarkar 1997].

Unfortunately, there are certain anomalies in Definition 11, and thus in GS-location consistency. We shall explain these anomalies in detail in Chapter 3. For now, we just say that Definition 11 does not confine nondeterminism. Historically, we proposed Definition 12 [Blumofe et al. 1996a] to solve these anomalies.

Explanation of Definition 12 Unlike Definition 11, Definition 12 in not readily expressible in terms of the last writer function. It introduces an observer function explicitly, however. Properties 12.1 and 12.4 are indeed the defining properties of observer functions. Property 12.3 corresponds to the semantics of reads (Postulate 4). The other two properties of the observer function are specific to this form of dag consistency. Property 12.2 says that the observer function of a node that writes must point to the node itself. Property 12.5 can be understood in two different senses.

In the first sense, Property 12.5 is Property 11.2 applied to the observer function instead of a read operation.

The other interpretation of Property 12.5 arises when we consider its contrapositive. We now explore this interpretation. The explanation is a bit long, but it will be useful for generalizing dag consistency to a wider class of models.

We first recall some notation from propositional logic. The symbol $\wedge$ denotes the logical *and* operator, and $\rightarrow$ denotes the logical implication. In order not to clutter the formulas with many parentheses, we assume that $=$ binds more strongly than $\wedge$, which in turn binds more strongly than $\rightarrow$. The expression $\Phi(l, w) = \Phi(l, u) \wedge \mathrm{W}(l, u) \rightarrow \Phi(l, v) = \Phi(l, u)$, therefore, is the same as $((\Phi(l, w) = \Phi(l, u)) \wedge \mathrm{W}(l, u)) \rightarrow (\Phi(l, v) = \Phi(l, u))$. (This formula appears in Definition 13 below.)

Consider now the contrapositive of Property 12.5, that is,

12.5$'$. For each triple u, v, and w of nodes such that $u \prec v \prec w$, we have

$$\Phi(l, w) = u \rightarrow \Phi(l, v) = u \ .$$

We have just manipulated symbols formally, for now. The next step is to substitute the condition "$\Phi(l, w) = u$" with another condition of the form "$\Phi(l, w) = \Phi(l, u)$". The substitution arises as follows. We claim that

$$\Phi(l, w) = u \qquad \Longleftrightarrow \qquad \Phi(l, w) = \Phi(l, u) \wedge \mathrm{W}(l, u) \ .$$

To prove the claim, observe that if $\Phi(l, w) = u$, then $\mathrm{W}(l, u)$, as stated by Property 12.1. By Property 12.2, we also have $\Phi(l, u) = u$, and the "$\Rightarrow$" follows. Conversely, if $\mathrm{W}(l, u)$, again by Property 12.2, we have $\Phi(l, u) = u$ and the "$\Leftarrow$" follows, proving the claim.

We now substitute the equivalence just proven into Property 12.5$'$, yielding the equivalent statement

12.5$''$. For each triple u, v, and w of nodes such that $u \prec v \prec w$, we have

$$\Phi(l, w) = \Phi(l, u) \wedge \mathrm{W}(l, u) \rightarrow \Phi(l, v) = \Phi(l, u) .$$

Therefore, we see that Property 12.5 is a form of ***convexity*** of the observer function: if $u \prec v \prec w$ and the observer function (for l) assumes the same value at u and w, it must assume that value also at v. This particular definition of dag consistency does not demand such convexity in all cases, but only when u writes to l.

Why do we say that Property 12.5 is a "convexity" property? We say that a set of nodes in a dag is ***convex*** if, whenever two nodes u and w are in the set, and $u \prec w$, then any node v that is in a path from u to w also belongs to the set. The definition is inspired by the usual definition of convexity in geometry, where a set is convex if, whenever two point are in the set the whole segment joining the two points is in the set. Property 12.5 says that, for each location, the set of nodes where the observer function is constant is convex. The terminology "convexity" is not that important for the purposes of this thesis, but the concept is. Convexity is the characteristic property of dag consistency.

New definitions of dag consistency We now redefine dag consistency so that it matches the technical definition of memory model (Definition 5). By the above discussion, we can reorganize Definition 12 in the following way.

Definition 13 ***WN-dag consistency*** *is the set WN* $= \{(G, \Phi)\}$, *where* $G = (V, E)$ *is a computation, and* Φ *is an observer function for* G *such that the following properties hold.*

13.1. For all $l \in \mathcal{M}$ *and for all* $u \in V$, *if* $W(l, u)$ *then* $\Phi(l, u) = u$.

13.2. For all locations l, *and for each triple* u, v, *and* w *of nodes such that* $u \prec v \prec w$, *we have*

$$\Phi(l, w) = \Phi(l, u) \wedge W(l, u) \rightarrow \Phi(l, v) = \Phi(l, u) .$$

The logical implication in Property 13.2 means that if $\Phi(l, w) = \Phi(l, u)$ and $\mathrm{W}(l, u)$, then $\Phi(l, v) = \Phi(l, u)$.

The model is called WN because, of the nodes u and v in the definition, we require that the first be a write ("W"), and we do not care about the second ("N" for "don't care"). We now define the model WW that requires that both nodes write. (In Section 4.1 we complete the picture by considering all combinations of writes and "don't care".) We then argue that WW is equivalent to the model from Definition 11.

Definition 14 ***WW-dag consistency*** *is the set WW* $= \{(G, \Phi)\}$, *where* $G = (V, E)$ *is a computation, and* Φ *is an observer function for* G *such that the following properties hold.*

14.1. For all $l \in \mathcal{M}$ and for all $u \in V$, if $W(l, u)$ then $\Phi(l, u) = u$.

14.2. For all locations l, and for each triple u, v, and w of nodes such that $u \prec v \prec w$, we have that

$$\Phi(l, w) = \Phi(l, u) \wedge W(l, u) \wedge W(l, v) \rightarrow \Phi(l, v) = \Phi(l, u) \; .$$

Observe that the only difference between the definitions of WN and WW is the antecedent "$\mathrm{W}(l, v)$" that appears in Property 14.2 but not in Property 13.2.

Observe also that Property 14.2 makes little sense if read literally, because, if the three antecedents are true, then Property 14.1 implies that the consequent is false! Indeed, if both $\mathrm{W}(l, u)$ and $\mathrm{W}(l, v)$ hold, then the consequent $\Phi(l, v) = \Phi(l, u)$ is equivalent to the proposition $v = u$, which is false because $u \prec v$. The "right" way to interpret Property 14.2 is to consider the contrapositive. We stated the definition in this way to show the similarity with Definition 13.

Equivalence of Definition 14 and Definition 11. Definition 14 and Definition 11 are equivalent for a simple reason, but it is difficult to state it formally. The simple reason is that if we have a WW observer function, it satisfies the properties of Definition 11 automatically. Conversely, if we have a shared memory satisfying Definition 11, we can build a WW observer functions by first assigning a value of the observer function to nodes that read and write memory, as imposed by the shared memory satisfying Definition 11, and then filling in all the other nodes in an almost arbitrary way. The point is that neither Definition 11 nor Definition 14 care about these nodes.

The formal difficulty lies in the fact that Definition 11 is expressed in terms of an *execution* of a computation and not in terms of observer functions. Therefore, we must invoke the standard semantics of reads (Postulate 4) to map observer functions into executions and vice versa.

We now informally argue the equivalence of Definition 14 with Definition 11. More precisely, we argue that a WW observer function, under the standard semantics of reads, yields to an execution that satisfies Definition 11. Conversely, from an execution that satisfies Definition 11, we can build a WW observer function that, under the standard semantics of reads, yields to the same execution. I apologize for the confusion, but I found no better way to explain this.

We first state the contrapositive of Property 14.2, in order to show the similarity with Definition 11.

14.2$'$. For all locations l, and for each triple u, v, and w of nodes such that $u \prec v \prec w$, we have that

$$\Phi(l, v) \neq \Phi(l, u) \wedge \mathrm{W}(l, u) \wedge \mathrm{W}(l, v) \rightarrow \Phi(l, w) \neq \Phi(l, u) \; .$$

If the property is stated in this way, it becomes easy to argue that an observer function satisfying Definition 14, if executed under the standard read semantics (Postulate 4), yields an execution of the computation that satisfies Definition 11. Indeed, suppose we have an observer function satisfying Definition 14. Then, Property 11.1 is satisfied by definition of observer functions (Conditions 3.1 and 3.2). Moreover, Property 11.1 is satisfied because of the semantics of read operations and 14.2′.

Conversely, we now argue that, from an execution that satisfies Definition 11, it is possible to build an observer function that satisfies Definition 14 and yields the same execution. The observer function is built as follows. For any location l and node w, we have that

- if w writes to l, let $\Phi(l, w) = w$;
- if w reads l, receiving a value written by node u, let $\Phi(l, w) = u$;
- otherwise, let u be any node that writes to l, such that $w \not\prec u$, and such that no node v satisfying $u \prec v \prec w$ writes to l. Such a node always exists, because the initial node writes to all locations: just pick the last write on any path from the initial node to w.

By construction, the function Φ yields an equivalent execution of the computation under the standard read semantics. We must now verify that Φ is a WW observer function. By construction, Φ is indeed an observer function, since we always set $\Phi(l, u)$ to a node that writes to l and does not follow u. Property 14.1 also holds by construction. As for Property 14.2′, suppose that $\Phi(l, v) \neq \Phi(l, u)$, and that $\mathrm{W}(l, u)$ and $\mathrm{W}(l, v)$ hold. Then, by construction and because of Definition 11, we have that $\Phi(l, w) \neq \Phi(l, u)$.

2.4.4 Other models

The preceding discussion provided computation-centric definitions of sequential, location, and dag consistency. These three models are further developed and investigated in the following chapters. For completeness, we now suggest how to give computation-centric definitions of processor consistency and other models from the literature. These models are presented here, but not further discussed in the rest of the thesis.

The following definition of ***processor consistency*** appeared in [Goodman 1989].

> A multiprocessor is said to be *processor consistent* if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program.

The first thing to notice is that the definition is very ambiguous. It seems to me that there are at least two legal interpretations of the word “appear”. In the first interpretation, “appear” means “appear to a processor”. In other words, each processor has its own idea about the order of memory accesses performed by the system, but each processor can have a different idea. In the second interpretation, “appear” means “appear to a memory operation”.

In other words, each operation is free to have its own idea about the rest of the system. We shall use the first interpretation, which seems to be endorsed by the rest of Goodman's paper.

In order to translate the definition into computation-centric terms, we must first identify a notion of "processor" in a computation. One possibility is to postulate that a "processor" is a maximal connected component of the graph consisting of the computation minus the initial node. This postulate is justified if we imagine the computation as composed of P chains of instructions, each one representing the instruction stream of one processor. The P chains are not connected (except for the initial node). Processor consistency says that, for each processor, there exists an order of memory operations observed by that processor. We now translate the definition in computation-centric terms.

Definition 15 ***Processor consistency*** *is the memory model*

$$\begin{aligned} PC \;=\; \{(G,\Phi) :\ & \Phi(l,u) = \mathcal{L}_{\mathcal{T}(\mathcal{C})}(l,u),\ u \in \mathcal{C}, \\ & \mathcal{C} \textit{ is a maximal connected component of } G - \textit{initial}(G), \\ & \textit{and } \mathcal{T}(\mathcal{C}) \textit{ is a topological sort of } G\}\ . \end{aligned}$$

We observe a common theme in previous definitions. In sequential consistency there is a single topological sort of the computation. In processor consistency there is a topological sort per component ("processor"). In location consistency there is a topological sort per location. And, in dag consistency (Definition 11) there is a topological sort per node (although the definition is not stated in that way).

Observe also that PC *per se* does not imply location consistency. Consider, for example, two processors, and one memory location. The first processor writes 1 to the location and then reads it. The second processor writes 2 to the location and then reads it. Processor consistency allows the first processor to receive 2, and the second processor to receive 1. This situation cannot happen in location consistency. On the other hand, the assumption of location consistency/coherence is often implicit in all memory models [Adve and Gharachorloo 1995] (except for the model in Gao and Sarkar [1994] and dag consistency). But, by their definitions alone, PC and LC are incomparable.

Some memory models (like ***weak ordering***) [Dubois et al. 1986; Adve and Hill 1990] distinguish between ordinary and synchronizing memory accesses. For completeness, we now briefly sketch how to extend our model to account for this case. First, the two kinds of memory operations must be distinguished. Then, we could define synchronized versions of, say, location consistency, along the lines that follow. We demand that there exist a topological sort $\mathcal{T}$ of all synchronizing operations. Instead of allowing each location to be serialized according to an arbitrary topological sort of the computation, we constrain the topological sort to be consistent with $\mathcal{T}$. Of course, many other variations are possible, and we did not investigate the whole spectrum of memory models. The point is that our framework seems to be powerful enough to encompass many (and maybe all) interesting memory models.

2.5 Summary

In this chapter, we set up the computation-centric theory of memory models. The important concepts are *computations* (Definition 1) and *observer functions* (Definition 3), which lead to the definition of a memory model (Definition 5).

We gave examples of computations in Section 2.1.1, and discussed the relative merits of computation-centric and processor-centric frameworks in Section 2.1.3.

Finally, we showed how to express sequential, location, dag, and processor consistency within the computation-centric theory. These models are analyzed in Chapter 3, where we introduce five properties of memory models and argue whether each model enjoys them.

The theory of dag consistency is further developed in Chapter 4, where we define a class of models similar to WW and WN-dag consistency, and understand their properties and mutual relationships.

It is my hope that with this very skeletal model I have constructed the reader will perceive some simple unifying principles of the field—principles which might otherwise be obscured by the enormously intricate interplay of phenomena at many different levels. What is sacrificed is, of course, strict accuracy; what is gained is, I hope, a little insight.

D. R. Hofstadter, *Gödel, Escher, Bach: an Eternal Golden Braid*, page 505

Chapter 3

Properties of memory models

In this chapter we discuss five properties of memory models. I argue that a model is not "reasonable" unless it obeys all five properties. The first three properties are formally specified. The other two are not well defined and have a more psychological flavor.

A model is ***complete*** if it allows at least one observer function for each computation.

A model is ***monotonic*** if an observer function is still in the model after some dag edges are removed.

A model is ***constructible*** if it is always possible to extend an observer function to a "bigger" dag.

A model ***confines nondeterminism*** if, whenever there is a join node that follows all nondeterministic memory accesses, the successors of the join node do not observe any nondeterministic behavior. (See Section 3.4 for a definition of nondeterminism.)

A model is ***classical*** (as opposed to "quantum") if reads behave like no-ops (i.e., successor nodes have no way to deduce that a read operation is performed by a predecessor).

Recall that in Chapter 2 we defined the memory models SC, LC, WW, WN, and PC. Table 3.1 summarizes the properties enjoyed by these models. From the table, we can see that WW and WN consistency are "unreasonable", since they lack some property.

In the rest of the chapter, we define and discuss the five properties, trying also to show which conditions should be met in order for a property to be true.

3.1 Completeness

A memory model that defines observer functions only for certain computations would be pretty useless. (Imagine having a memory system that, fed with a computation, says "I cannot do this".) We say that a model is ***complete*** if it defines an observer function for every computation. Completeness is thus a necessary property of all useful memory models.

Definition 16 *A memory model Δ is **complete** if, for any computation G, there exists an observer function Φ such that $(G, \Phi) \in \Delta$.*

model	complete	monotonic	constructible	*confines nondet.*	*classical*
SC	✓	✓	✓	✓	✓
LC	✓	✓	✓	✓	✓
WW	✓	✓	✓	no	✓
WN	✓	✓	no	✓	✓
PC	✓	✓	✓	??	✓

Table 3.1: Summary of the properties enjoyed by the memory models defined in Chapter 2. A check mark ✓ means that the model has the property. The first three columns refer to properties for which there is a mathematical definition. The last two columns refer to properties that are not precisely defined, and that I believe are desirable from a programmer's perspective. Consequently, a check mark in these columns means "I believe the model has the property". A "no" means, however, that the model definitely does not have the property. A "??" means that my intuition of the property fails to apply to the model, and thus I don't know how the model behaves.

A model weaker than a complete model is also complete.

Theorem 17 *Let Δ and Δ' be two memory models. If Δ is complete and $\Delta \subseteq \Delta'$, then Δ' is also complete.*

Proof: Since Δ is complete, then for any computation G, there exists an observer function Φ such that $(G, \Phi) \in \Delta \subseteq \Delta'$, proving that Δ' is also complete. ∎

For the proof that all the models in Table 3.1 are complete, we argue that sequential consistency is complete, and that all those models are weaker than sequential consistency. The proofs are not very interesting, and the reader can skip the rest of this section with no harm.

Lemma 18 *SC is complete.*

Proof: Let G be a computation. We know that there exists a topological sort $\mathcal{T}$ of G. By definition, $(G, \mathcal{L}_{\mathcal{T}}) \in$ SC. ∎

Lemma 19 *The memory models LC, WW, WN, and PC are weaker than SC, that is, the four inclusions SC $\subseteq$ LC, SC $\subseteq$ WW, SC $\subseteq$ WN, and SC $\subseteq$ PC hold.*

Proof: The inclusions SC $\subseteq$ LC and SC $\subseteq$ PC are immediate from the definitions of the models, since if there exists a global topological sort, then there also exists a topological sort for each location and for each connected component.

Proof that SC $\subseteq$ WN: Let $(G, \Phi) \in$ SC and $\Phi = \mathcal{L}_{\mathcal{T}}$ for some topological sort $\mathcal{T}$. We want to prove that Φ obeys Definition 13. By definition of $\mathcal{L}_{\mathcal{T}}$, Property 13.1 holds. We now prove Property 13.2. Suppose, by contradiction, that there exists a triple u, v, and w

of nodes such that $u \prec v \prec w$, and that $\Phi(l, w) = \Phi(l, u)$ holds, but $\Phi(l, v) \neq \Phi(l, u)$. The relation $\Phi(l, v) \neq \Phi(l, u)$ implies that $\mathcal{L}_{\mathcal{T}}(l, v)$ is some node x on the path from u to v. Consequently, x writes to l, and x lies between u and w in $\mathcal{T}$, whence $\mathcal{L}_{\mathcal{T}}(l, u)$ cannot be the same as $\mathcal{L}_{\mathcal{T}}(l, w)$. This contradiction proves the property.

Proof that SC $\subseteq$ WW: The proof is the same as the proof that SC $\subseteq$ WN. ∎

We can now conclude that all the above models are complete.

Theorem 20 *The memory models SC, LC, PC, WN, and WW are complete.*

Proof: Since SC is complete (Lemma 18) and SC is stronger than every model in the statement (Lemma 19), the theorem follows from Theorem 17. ∎

3.2 Monotonicity

We now define ***monotonicity***. Suppose we remove some edges from a computation. Then, it becomes possible to execute the dag in more ways than it was possible before. Yet, all valid executions of the original dag should still be valid, as should be all valid observer functions. Monotonicity says that a valid observer function is still valid after we remove edges from a dag.

Definition 21 *A memory model Δ is **monotonic** if, for all $((V, E), \Phi) \in \Delta$, we have that $((V, E'), \Phi) \in \Delta$ for any $E' \subset E$.*

We now informally argue that the memory models SC, LC, PC, WN, and WW are monotonic. (A formal proof would be tedious and not say anything new.) For the models defined in terms of the last writer (that is, SC, LC, and PC) observe that a topological sort of a computation is still a topological sort after some edges are removed from the computation. In other words, removing edges can only grow the set of topological sorts, and thus of observer functions. For WN and WW, observe that the implications in Properties 13.2 and 14.2 can become vacuously true when edges are removed, but can never become false if they were true before the removal.

3.3 Constructibility

In this section we define ***constructibility***, which says that if we have a dag and an observer function in some model, it is always possible to extend the observer function to a "bigger" dag. Constructibility tries to capture the idea that a memory model can be implemented exactly, i.e., without implementing a stronger model. Remarkably, not all memory models are constructible. We show that for an arbitrary memory model, however, there is a natural way to define a unique constructible version of it. Finally, we give necessary and sufficient conditions for the constructibility of monotonic memory models.

We now explore the basic idea behind constructibility. Suppose that, instead of being specified completely at the beginning of an execution, a computation is revealed online by an adversary.[1] Suppose also that a consistency algorithm exists that maintains a given memory model online. In a sense, the consistency algorithm is just building an observer function while the computation is being revealed. Suppose that, at some point in time, an observer function exists for the part of computation known so far, but when a new computation node is revealed by the adversary, the observer function cannot be extended to the new node. In this case, the consistency algorithm is "stuck". It should have chosen a different observer function in the past, but that would have required some knowledge of the future behavior of the adversary. Constructibility prohibits this situation from occurring. A valid observer function in a constructible model can always be extended to a bigger dag when new dag nodes are revealed.

Suppose now that we have a nonconstructible model, and an online consistency algorithm that supports the model. The algorithm cannot run the risk of producing an observer function that cannot be extended when new nodes are revealed, because it does not know what the adversary does. Therefore, the algorithm can produce only a strict subset of the observer functions allowed by the model—that is, it must maintain a strictly stronger model. Consequently, computer architects and programmers should not adopt a nonconstructible memory model; they should just adopt the stronger model.

The formal definition of constructibility depends on the notion of a "prefix" of a dag.

Definition 22 *A dag G' is a **prefix** of a dag G if, for all nodes u and v, we have that*

$$(u \prec_G v) \wedge (v \in G') \Longrightarrow u \prec_{G'} v \ .$$

In other words, a dag G' is a prefix of another dag G if, whenever a node is in G', all the predecessors of the node are also in G':

We also need to introduce a new notation. If Φ is an observer function for a computation G, and G' is a subgraph of G, we say that $\Phi' = \Phi|_{G'}$ is the ***restriction*** of Φ to G' if Φ' is an observer function for G' and coincides with G over its domain. We also say that Φ is an ***extension*** of Φ' to G.

We now define constructibility.

Definition 23 *A memory model Δ is **constructible** if the following property holds: for any computation G and for any prefix G' of G, if $(G', \Phi') \in \Delta$, then there exists an extension Φ of Φ' to G such that $(G, \Phi) \in \Delta$.*

Definition 23 says that if the memory model allows an observer function for a prefix, then the function must be extensible to the entire dag.

[1] Such is indeed the case with multithreaded languages, such as Cilk [Blumofe 1995; Joerg 1996], where the adversary corresponds to the programmer.

A simple (almost trivial) consequence of constructibility is given by the next theorem.

Theorem 24 *A constructible memory model is complete.*

Proof: Immediate from the fact that the empty computation is a prefix of all computations and, together with its unique observer function, belongs to every memory model. ■

The rest of this section addresses three topics. We first show that, for any memory model (whether constructible or not), a well-defined ***constructible version*** of the model exists. We then give necessary and sufficient conditions for the constructibility of complete and monotonic memory models. Finally, we use these conditions to prove the constructibility of SC, LC, and PC. We state that WW is constructible and that WN is not, but the proofs are delayed until Section 4.4, where a wider class of dag-consistent models is defined and their properties analyzed.

3.3.1 Constructible version of a model

In this section, we prove that the weakest constructible model Δ^* stronger than a given model Δ exists and is unique. We call Δ^* the ***constructible version*** of Δ.

We start by proving that the union of constructible models is constructible, and then define the constructible version as an infinite union of constructible models.

Lemma 25 *Let $\mathcal{S}$ be a (possibly infinite) set of constructible memory models. Then $\bigcup_{\Delta \in \mathcal{S}} \Delta$ is constructible.*

Proof: Let G be a computation, and let G' be a prefix of G. We want to prove that, if $(G', \Phi') \in \bigcup_{\Delta \in \mathcal{S}} \Delta$, then there exists an extension Φ of the observer function Φ' such that $(G, \Phi) \in \bigcup_{\Delta \in \mathcal{S}} \Delta$.

If $(G', \Phi') \in \bigcup_{\Delta \in \mathcal{S}} \Delta$, then $(G', \Phi') \in \Delta$ for some $\Delta \in \mathcal{S}$. Since Δ is constructible, there exists an observer function Φ for G such that $(G, \Phi) \in \Delta$ and $\Phi|_{G'} \equiv \Phi'$. Consequently, $(G, \Phi) \in \bigcup_{\Delta \in \mathcal{S}} \Delta$, proving that $\bigcup_{\Delta \in \mathcal{S}} \Delta$ is constructible. ■

Theorem 26 *For any memory model Δ, there exists a unique memory model Δ^* such that*

26.1. $\Delta^* \subseteq \Delta$;

26.2. Δ^* *is constructible;*

26.3. for any constructible model Δ' such that $\Delta' \subseteq \Delta$, we have $\Delta' \subseteq \Delta^*$.

Proof: Let Δ^* be the union of all constructible models Δ' such that $\Delta' \subseteq \Delta$. Then Δ^* satisfies Condition 26.1. Also, Δ^* satisfies Condition 26.2, because of Lemma 25. Condition 26.3 is satisfied by construction of Δ^*.

To show uniqueness, suppose that there exists another model Δ' satisfying the three properties. Then $\Delta' \subseteq \Delta^*$, because of Condition 26.3, and $\Delta^* \subseteq \Delta'$, again because of Condition 26.3 applied to Δ'. Therefore, we have $\Delta^* = \Delta'$. ∎

We now define what we mean by the constructible version of a model.

Definition 27 *For any memory model Δ, the unique Δ^* that satisfies all the conditions of Theorem 26 is called the **constructible version** of Δ.*

The constructible version Δ^* of a model Δ is therefore the weakest constructible model that is stronger than Δ. Observe that $\Delta^* = \Delta$ if and only if Δ is constructible.

Giving a simple expression for the constructible version of a memory model can be very hard in general. For example, I have not been able to characterize the constructible version of WN other than by means of Definition 27. In Chapter 4 we define a related model called NN, and in Section 4.5 we prove that location consistency is the constructible version of NN.

3.3.2 Conditions for constructibility

In this section, we establish necessary and sufficient conditions for memory models to be constructible.

Recall our intuition of constructibility as the property that allows us to extend a valid observer function to a bigger dag. One might try to extend the observer function one node at the time (including of course all the incoming edges), and hope to prove constructibility in this way. While fundamentally correct, this intuition suffers from two problems. First, the memory model may not be complete, and there may be no valid observer function for the intermediate dags. Second, the order in which nodes and edges are inserted may make a difference, in the sense that one order of insertion may lead to an extension, while another order might not. Neither of these difficulties arises if the memory model is complete and monotonic, however.

We start by defining the concept of an ***augmented dag***.

Definition 28 *Let $G = (V, E)$ be a dag, and let $\mathcal{O}$ be a memory operation (either a write to location l, a read from location l, or a no-op). The **augmented dag** of G, denoted $aug_{\mathcal{O}}(G)$, is a dag $G' = (V', E')$, where*

$$\begin{aligned} V' &= V \cup \{\mathit{final}(G)\} \ , \\ E' &= E \cup \{(u, \mathit{final}(G)) : u \in V\} \ , \end{aligned}$$

and $\mathit{final}(G) \notin V$ is a new node that performs the memory operation $\mathcal{O}$.

The augmented dag, therefore, consists of the original dag plus a new node which is a successor of every node in the original dag.

In the next theorem, we give necessary and sufficient conditions for the constructibility of monotonic memory models.

Theorem 29 *A monotonic memory model Δ is constructible if and only if for all $(G, \Phi) \in \Delta$ and for all memory operations $\mathcal{O}$, there exists an observer function Φ' such that we have $(aug_{\mathcal{O}}(G), \Phi') \in \Delta$ and $\Phi'|_G \equiv \Phi$.*

Proof: The "$\Rightarrow$" part is obvious, since G is a prefix of $aug_{\mathcal{O}}(G)$.

For the "$\Leftarrow$" direction we must prove that, in the given hypotheses, if G is a prefix of G' and $(G, \Phi) \in \Delta$, then $(G', \Phi') \in \Delta$ for some extension Φ' of Φ.

We first state without proof an important property of prefixes of a finite dag. If G is a prefix of G', then there exists a sequence of dags $G_1, G_2, \ldots, G_k$ such that G_i is a prefix of G_{i+1}, the graph G_{i+1} has the same nodes as G_i plus one new node, and the sequence starts with $G_1 = G$, and ends with $G_k = G'$.

The proof of the theorem uses induction on the length k of such a sequence. The base case $k = 1$ is obvious. Now, suppose inductively that there exists Φ_k such that $(G_k, \Phi_k) \in \Delta$, and let $\mathcal{O}$ be the memory operation performed by the node in G_{k+1} that does not belong to G_k. By assumption, there exists an observer function Φ'_k such that $(aug_{\mathcal{O}}(G_k), \Phi'_k) \in \Delta$. Let $(V'_k, E'_k) = aug_{\mathcal{O}}(G_k)$ and $(V_{k+1}, E_{k+1}) = G_{k+1}$. Then, up to isomorphisms, we have $V_{k+1} = V'_k$ and $E_{k+1} \subset E'_k$. By monotonicity of Δ, we have $(G_{k+1}, \Phi'_k) \in \Delta$, completing the inductive step. ∎

One interpretation of Theorem 29 is particularly significant. Consider an execution of a computation. At any point in time some prefix of the dag has been executed. If at all times it is possible to define a "final" state of the memory (given by the observer function on the final node of the augmented dag), then the memory model is constructible.

3.3.3 Constructibility of example models

We now prove the constructibility of SC, LC, and PC. Theorem 29 is used in the proof. We also know that WW is constructible and WN is not constructible, but we delay the proof until Section 4.4, where we discuss a whole class of dag-consistent models.

Theorem 30 *SC, LC, and PC are constructible memory models.*

Proof: We just give the proof for LC, since the proof for the other models is similar. The proof uses Theorem 29. Since LC is monotonic, we just need to prove that it is possible to extend any observer function to the final node.

Let $(G, \Phi) \in$ LC. Therefore, for all locations l there exists a topological sort $\mathcal{T}(l)$ of the dag G such that for all nodes u, we have $\Phi(l, u) = \mathcal{L}_{\mathcal{T}(l)}(l, u)$. Now, for any memory operation $\mathcal{O}$, consider the augmented dag $aug_{\mathcal{O}}(G)$, and let $\mathcal{T}'$ be the following total order of $aug_{\mathcal{O}}(G)$: all the nodes of in G in the order specified by $\mathcal{T}(l)$, followed by *final*(G).

It is immediate that $\mathcal{T}'$ is a topological sort of $aug_{\mathcal{O}}(G)$. Then, the function Φ defined by $\Phi'(l, u) = \mathcal{L}_{\mathcal{T}'}(l, u)$ is a valid LC observer function on $aug_{\mathcal{O}}(G)$, and it coincides with Φ on G. The theorem follows by applying Theorem 29. ∎

3.4 Nondeterminism confinement

In this section and in Section 3.5 we discuss two more "properties" of memory models. Unlike completeness, constructibility, and monotonicity, these two properties are by no means well defined. Instead, they try to capture my intuition of reasonableness. In this section we address memory models that confine nondeterminism, or, more precisely, memory models that *do not* confine nondeterminism. Specifically, WW-dag consistency is one such model.

A computation is ***nondeterministic*** when there exist two incomparable nodes that either write to the same location, or one of which writes and the other reads the same location. We also say that these memory accesses ***conflict***. This notion of nondeterminism captures the idea that there are some conflicting memory accesses and the conflict is not resolved *by the computation*. It is worth pointing out that the conflict might indeed be resolved *by the memory model* in some funny way. For example, the memory model might say that all nodes with a prime number of outgoing edges win when conflicting with other nodes, and thus the execution of the computation can be deterministic even if the computation is not. For our purposes, these subtleties can be ignored.

I want to discuss the case where part of a computation is nondeterministic, but the rest is deterministic, and I want to argue that the deterministic part should behave deterministically. This requirement, however, is too strong in general (after all, every node is deterministic, if considered in itself), but there is one case where it makes sense. The specific case I have in mind is when there is a "join" node u, and every node in the computation is either a predecessor or a successor of u. Suppose that the part of computation composed by the successors of u is deterministic, and the part composed by the predecessors is not. I want to argue that the deterministic part should behave deterministically. WW-dag consistency does not obey this requirement.

Consider the computation in Figure 3-1, and suppose the computation is executed under a WW-dag consistent memory model. One possible outcome of the computation is that $l_1 = 1$ and $l_2 = 0$. We now argue that this behavior is undesirable. The above computation may be representative of the following pseudo-Cilk program:

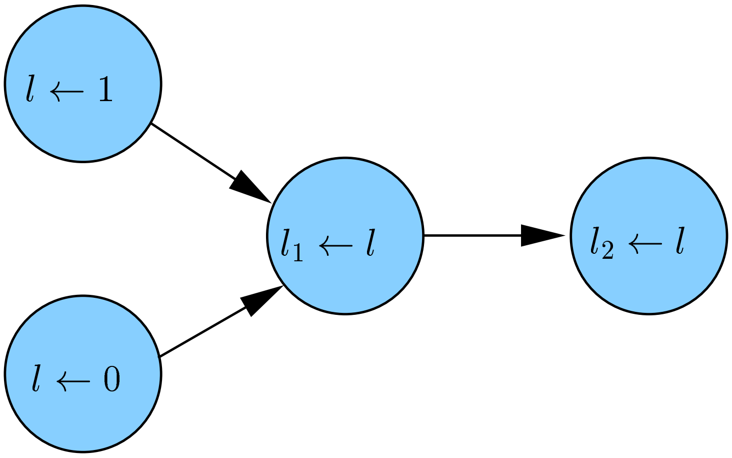


Figure 3-1: A computation used to show that WW does not confine nondeterminism. The computation consists of two nodes (on the left) that write to location l, and of two other nodes that read l.

```
spawn (write 0.0 to A);
spawn (write 1.0 to A);
sync;

if (A != 0.0) {
     /* A can now be == 0.0 ! */
     B = 1.0 / A;
}
```

My point is that one cannot write meaningful nondeterministic programs in such a model, since a single pair of incomparable writes makes the rest of the program nondeterministic, and the system is permitted to never resolve the concurrent writes in a way or the other. As a programmer, I believe that, at any node, all predecessors should be regarded as "past" and ignored thereafter, for reasons of abstraction and modularity, at the very least. For example, you might want to call a subroutine, and wait for the result. You certainly do not want to know how the subroutine works internally in order to call it. If nondeterminism is not confined inside the subroutine, you must be aware that nondeterminism can escape from it. This behavior seems unreasonable, because it breaks the abstraction boundary. WW does not confine nondeterminism. Since GS-location consistency is the same model as WW-dag consistency (as we argued in Section 2.4.3), it has the same problem.

The other models we discussed so far do indeed confine nondeterminism. For example, Figure 3-1 does not apply WN-dag consistency. We now argue that WN confines nondeterminism. Historically, this was the reason why WN was introduced in the first place. The main idea is that every node is forced to have a viewpoint on memory, given by the observer function. This viewpoint influences all the successors and hides past nondeterminism from their sight. In the example in Figure 3-1, the middle node is forced to "choose" between one of the preceding writes, and the last node must be consistent with that choice because of the "convexity" Property 13.2.

Sequential consistency and location consistency arguably confine nondeterminism, because every node has a clear idea of whether a write happened or not, and successors must see the same order of writes.

Finally, my intuition fails to see whether processor consistency confines nondeterminism or not. The main problem is that processor consistency is naturally defined only when the computation consists of P chains of instructions. In this case, there is no notion of join points that resolve nondeterminism, and the property holds vacuously.

3.5 Classical and "quantum" models

In this section we discuss a strange behavior of memory models that treat read operations in a way different from no-ops.

Historically, the following definition was proposed to confine nondeterminism in WW-dag consistency. The definition is expressed using the same wording as Definition 11.

Definition 31 *The shared memory $\mathcal{M}$ of a multithreaded computation $G = (V, E)$ is **dag consistent** if the following three conditions hold.*

1. *Whenever any node $u \in V$ reads any location $l \in \mathcal{M}$, it receives a value x written by some node $v \in V$ such that v writes x to l and $u \not\prec v$.*

2. *For any three nodes $u, v, w \in V$, satisfying $u \prec v \prec w$, if v writes some location $l \in \mathcal{M}$ and w reads l, then the value received by w is not written by u.*

3. *For any three nodes $u, v, w \in V$, satisfying $u \prec v \prec w$, if v reads some location $l \in \mathcal{M}$, receiving a value written by u, and moreover w reads l, then the value received by w is not written by u.*

Definition 31, however, never appeared in the literature, and Definition 12 (WN-dag consistency) was used instead. We now discuss what is wrong with Definition 31.

Observe that Definition 31 contains a condition of the form "and node w *reads* a location". This phrasing is dangerous, because reading a location can possibly influence the memory semantics. Consider, for example, the computation in Figure 3-2. Suppose that the computation is as shown in the figure, with the middle node being a no-op. In this case, a possible outcome of the computation is that $l_1 = 1$ and $l_2 = 0$ (if Definition 31 is used as memory model). Now, suppose that the middle node *reads* location l. Then, all executions must satisfy the property $l_1 = l_2$. The system implementing the memory model, therefore, exhibits a behavior of this kind: If the middle node does nothing, the value of l is indeterminate when the node is executed. If the middle node *observes* l, then suddenly the location becomes determinate.

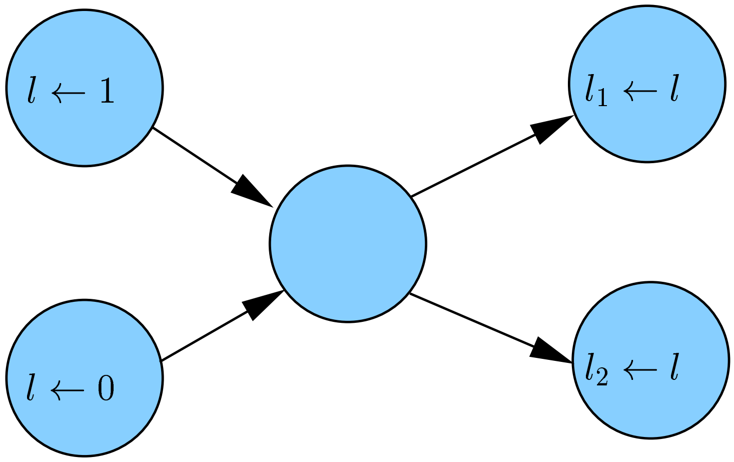


Figure 3-2: A computation used to show "quantum" effects in Definition 31. See the text for an explanation. The middle node may either do nothing, or read location l. If the read occurs, then Definition 31 confines nondeterminism, otherwise it does not.

Like the case where nondeterminism is not confined, I regard this sort of behavior as undesirable for a programmer. The observer function ensures that it is always possible to speak about the "value of a location at a node", no matter what operation the node performs on memory (including a no-op). The point of this section is that one should be careful in the treatment of reads and no-ops. The technical machinery introduced by observer functions is a way to give proper memory semantics to no-ops.

None of the models considered in Chapter 2 exhibits quantum effects, as can be seen by inspecting the definitions, where reads and no-ops are always treated in the same way. These model are, therefore, "classical".

3.6 Summary

In this chapter we defined five properties that every reasonable memory model should obey: completeness, monotonicity, constructibility, nondeterminism confinement, and classicality.

The first two are almost trivial and not very important (in the sense that you would not even call "memory model" something that does not obey them).

The discovery of constructibility is a major contribution of this thesis, and the consequences of non constructibility are discussed in detail in Chapter 4.

Whether the other two properties are necessary can be matter of discussion. I think they are (would you buy a quantum computer?). Other people can disagree, but, at the very least, this thesis has the merit of showing these properties explicitly.

Chapter 4

The weakest memory models

In this chapter we tackle the problem of identifying the weakest reasonable memory model. In order to follow the exposition, it will be helpful to look at the diagram in Figure 4-1, which depicts all the memory models discussed in this chapter.

We have already seen that WW-dag consistency is unreasonable, because it does not confine nondeterminism. Yet, dag consistency has been useful as a memory model for Cilk. After we (the Cilk authors) discovered this problem of WW, we proposed WN as a better memory model for Cilk. In this chapter, we show that WN is not good either, because it is not constructible.

This chapter will show how to fix dag consistency so that it obeys the five properties discussed in Chapter 3.

In the same spirit of WW and WN, in this chapter we define a whole class of dag-consistent memory models, including NW- and NN-dag consistency, which complement the definitions of WW and WN. The strongest form of dag consistency is NN.

Of the four models, only WW dag consistency turns out to be constructible. Recall now that, even if a model is not constructible, we can always look at its constructible version, which has all the properties of the model and moreover is constructible. Remarkably, in this chapter we prove that $NN^* = LC$. In other words, if you want to implement the strongest form of dag consistency, you cannot avoid implementing location consistency.

What about WN^* and NW^*? I know very little about them. NW does not confine nondeterminism, and it is likely that NW^* has the same problem. WN^* seems to obey all the properties described in Chapter 3, but I could not find any simple way to define it, other than by means of the definition of constructible version.

The rest of this chapter is organized as follows. We first define the class of dag-consistent memory models, and identify WW, WN, NW, NN among them. (WW and WN are the models that we defined in Section 2.4.3.) We then show that these four models are distinct, and prove that NN is the strongest dag-consistent model. We then determine the constructibility properties of these four models, and prove the equivalence of NN^* and LC. We finally comment about what this discussion can tell us about the weakest reasonable memory model.

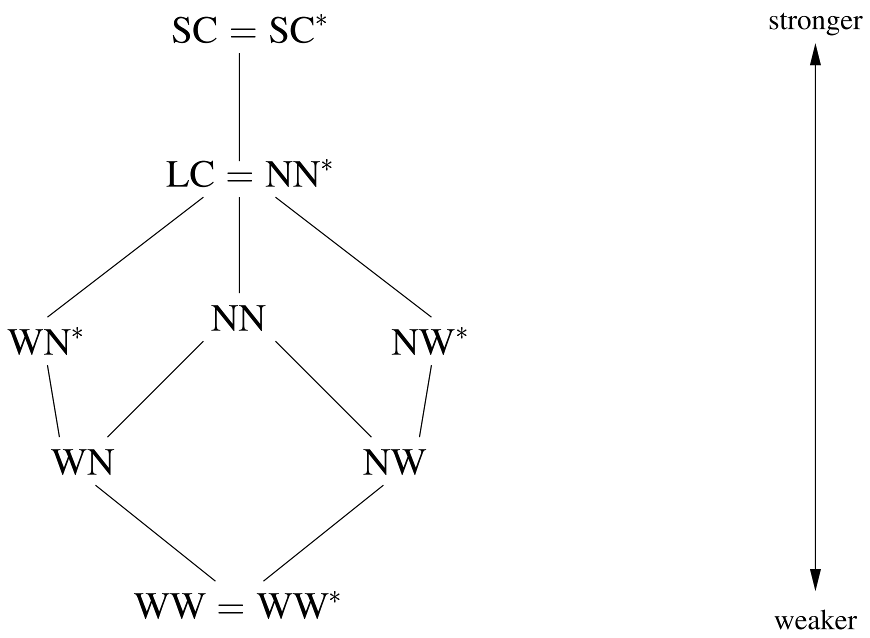


Figure 4-1: Summary of the relations among the memory models discussed in this chapter. A line means that the model at the lower end of the line is strictly weaker than the model at the upper end. For example, LC is strictly weaker than SC. We do not know where exactly WN* and NW* lie in the diagram.

4.1 Dag-consistent memory models

In Section 2.4.3 we introduced two memory models: WW and WN. We remarked that these models are equivalent to the two dag consistency models of [Joerg 1996; Blumofe et al. 1996b] and [Blumofe et al. 1996b], respectively. We also observed that the definitions of WW and WN are very similar. We now define a class of ***dag-consistent memory models*** whose definition is similar to WW and WN, and complete the selection of W's and N's by defining the two models NW and NN.

Recall the definitions of WW (Definition 14) and WN (Definition 13). Of the two properties required by each definition, the first one is the same for both models. The second property is different, but it can abstracted as follows. Let $Q(l, u, v, w)$ a predicate on nodes and locations, and consider the following version of the property:

> For all locations l, and for each triple u, v, and w of nodes such that $u \prec v \prec w$, we have that
>
> $$\Phi(l, w) = \Phi(l, u) \wedge Q(l, u, v, w) \to \Phi(l, v) = \Phi(l, u) \ .$$

If we let $Q(l, u, v, w) \equiv \mathrm{W}(l, u)$, we get WN. In the same way, if we let $Q(l, u, v, w) \equiv \mathrm{W}(l, u) \wedge \mathrm{W}(l, v)$, we get WW. In other words, we get different forms of dag consistency by varying the predicate Q.

In an analogous fashion, we can define a Q-dag consistency model for any predicate Q we might think of.

Definition 32 *Let $Q : \mathcal{M} \times V \times V \times V \mapsto \{true, false\}$. Q**-dag consistency** is the set $DC(Q) = \{(G, \Phi)\}$, where $G = (V, E)$ is a computation, and Φ is an observer function for G such that the following properties hold:*

32.1. For all $l \in \mathcal{M}$ and for all $u \in V$, if $W(l, u)$ then $\Phi(l, u) = u$.

32.2. For all locations l, and for each triple u, v, and w of nodes such that $u \prec v \prec w$, we have that

$$\Phi(l, w) = \Phi(l, u) \wedge Q(l, u, v, w) \to \Phi(l, v) = \Phi(l, u) \ . \tag{4.1}$$

By varying Q, we get the whole class of ***dag-consistent memory models***. They all obey a convexity condition (Property 32.2) similar to the defining property of WW and WN (Property 14.2 and Property 13.2). The rest of this chapter is concerned with four particular dag-consistent models: WW, WN, NW, and NN. They are defined as follows.

Definition 33 *The memory models WW, WN, NW, and NN are defined as follows.*

$$WW \stackrel{\Delta}{=} DC(W(l, u) \wedge W(l, v))$$

$$
\begin{aligned}
\mathit{WN} &\triangleq \mathit{DC}(W(l, u)) \\
\mathit{NW} &\triangleq \mathit{DC}(W(l, v)) \\
\mathit{NN} &\triangleq \mathit{DC}(\mathit{true})
\end{aligned}
$$

Rationale: We are interested in NN because it is the strongest dag-consistent memory model. (See Section 4.2.) We introduced WW and WN in Section 2.4.3, remarking that they are equivalent to the two dag-consistent models that, with Blumofe *et. al*, I published in the past. I consider NW for completeness and symmetry reasons, although we do not fully understand what the definition really means and whether this memory model has any benefit.

Monotonicity of dag-consistent models We now argue informally that all dag-consistent models are monotonic. Consider a computation and a dag-consistent observer function. Removing edges of the computation cannot create new paths in the dag. Consequently, if Property 32.2 was true before the removal for any three nodes u, v and w, it is still true after the edges are removed.

4.2 Relationships among dag-consistent models

We now analyze the mutual relationships among the four models we just defined. In summary, NN is the strongest model, WW is the weakest, and all the models are distinct. Moreover, NN in stronger than Q-dag consistency, for any predicate Q. (Stronger is taken here in the technical sense of Definition 6, i.e., stronger or equal.)

Theorem 34 *The following inclusions hold among the NN-, NW-, WN- and WW-dag consistency models:*

34.1. $\mathit{NN} \subsetneq \mathit{NW}$

34.2. $\mathit{NW} \subsetneq \mathit{WW}$

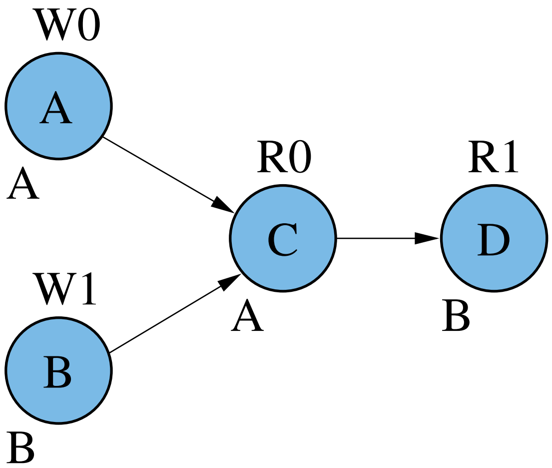
34.3. $\mathit{NN} \subsetneq \mathit{WN}$

34.4. $\mathit{WN} \subsetneq \mathit{WW}$

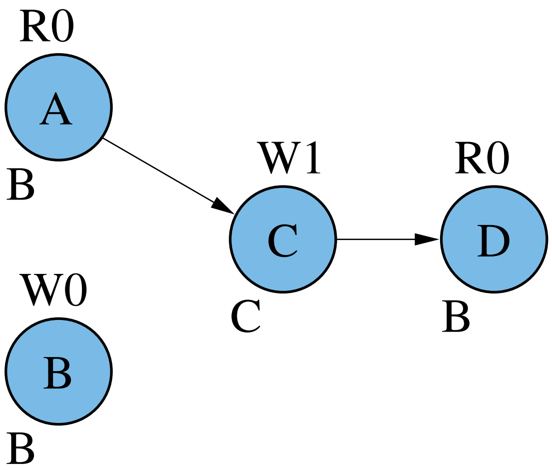
34.5. $\mathit{WN} \not\subseteq \mathit{NW}$

34.6. $\mathit{NW} \not\subseteq \mathit{WN}$

Proof: The four inclusions in Relations (34.1)-(34.4) are immediate from the definitions. We must now prove that the inclusions are strict, and also prove Relations (34.5) and (34.6). We shall exhibit two pairs of computation/observer function. The first pair belongs to WW



(a)



(b)

Figure 4-2: (a) Example of computation/observer function belonging to WW and NW, but not to WN or NN. (b) Example of computation/observer function belonging to WW and WN, but not to NW or NN. Each of the two dags in the figure have four nodes, A, B, C and D (the name of the node is shown inside the node). The memory consists of only one location, which is understood. Every node performs a read or a write operation on the location (for example, $W0$ means that the node writes a 0 to the location, and $R1$ means that it reads a 1). The value of the observer function is displayed below each node. For example, in part (a), the value of the function for node C is A, which accounts for the fact that node C reads the value written by node A.

and NW, but not to the other two models; the second belongs to WW and WN, but not to the other two models. These two counterexamples therefore suffice to prove the theorem.

For the counterexamples, it is enough to consider a single memory location l, which will henceforth be left unspecified. Figure 4-2 shows two dags. The first belongs to WW and NW, but not to WN or NN, as it is easily verified. In the same way, the reader can verify that the second example in Figure 4-2 belongs to WW and WN, but not to NW or NN. ∎

Incidentally, NN $\subsetneq$ NW $\cap$ WN and WW $\supsetneq$ NW $\cup$ WN, as can be shown by suitable counterexamples.

The model NN is indeed the strongest dag-consistent model, in an absolute sense, as stated by the following theorem.

Theorem 35 *For all predicates Q, we have that*

$$\textit{NN} \subseteq Q\textit{-dag consistency} .$$

Proof: The proof is immediate from the definition. An observer function satisfying Property 32.2 with $Q(u, v, w, l) = \textit{true}$ also satisfies Property 32.2 for any other predicate Q. ∎

4.3 Dag consistency vs. location consistency

In this section we investigate the relation between the dag-consistent models and location consistency. It turns out that location consistency is strictly stronger than NN-dag consistency. Since NN-dag consistency is the strongest dag-consistent model, location consistency is also stronger than all the dag-consistent models.

We now prove that location consistency is strictly stronger than NN-dag consistency. We split the proof into two parts. We first prove that LC $\subseteq$ NN, and then that LC $\neq$ NN.

Theorem 36 *Location consistency is stronger than NN-dag consistency, that is, LC $\subseteq$ NN.*

Proof: Let $(G, \Phi) \in$ LC. We want to prove that $(G, \Phi) \in$ NN. For all locations l, we argue as follows. Suppose, by contradiction, that $(G, \Phi) \notin$ NN, and therefore there exist three nodes u, v and w that violate Property 32.2. In other words, suppose that $u \prec v \prec w$, and $\Phi(l, u) = \Phi(l, w)$, but $\Phi(l, v) \neq \Phi(l, w)$.

Since $(G, \Phi) \in$ LC, there exists a topological sort $\mathcal{T}(l)$ of the dag G such that $\Phi(l, x) = \mathcal{L}_{\mathcal{T}(l)}(l, x)$ for all nodes x. Therefore, we have that $\mathcal{L}_{\mathcal{T}(l)}(l, u) \neq \mathcal{L}_{\mathcal{T}(l)}(l, v)$, which implies that $\mathcal{L}_{\mathcal{T}(l)}(l, v)$ is some node x on the path from u to v. Consequently, x writes to l, and x lies between u and w in $\mathcal{T}(l)$, whence $\mathcal{L}_{\mathcal{T}(l)}(l, u)$ cannot be the same as $\mathcal{L}_{\mathcal{T}(l)}(l, w)$. This contradiction proves the theorem. ∎

As a corollary, LC is also stronger than all the other dag consistency models.

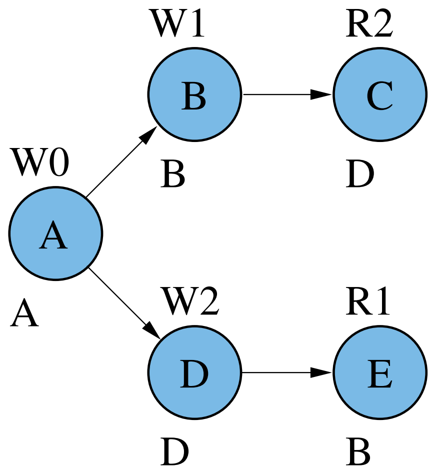


Figure 4-3: Example of computation/observer function belonging to NN but not to LC. An explanation of the conventions used in the figure appears in the caption of Figure 4-2.

Corollary 37 *The following relations hold:*

$$\begin{aligned} LC &\subseteq NW \\ LC &\subseteq WN \\ LC &\subseteq WW \end{aligned}$$

Proof: The corollary follows immediately from Theorem 36 and Theorem 34. ■

Corollary 38 *NN, NW, WN and WW are complete memory models.*

Proof: The corollary follows from completeness of LC, Theorem 36, Corollary 37 and Theorem 17. ■

In order to prove that location consistency is strictly stronger than NN-dag consistency we shall present a counterexample in the same style as Theorem 34.

Theorem 39 *LC $\neq$ NN.*

Proof: Figure 4-3 shows a pair (G, Φ) that belongs to NN, as can be verified. As in the proof of Theorem 34, there is a single memory location l, which is understood.

We shall now prove that the pair shown in the figure does not belong to LC. Suppose by contradiction that there exists a topological sort $\mathcal{T}$ of G such that $\Phi(l, u) = \mathcal{L}_{\mathcal{T}}(l, u)$ for all nodes u. Then $\Phi(l, C) = D$ implies that $B \prec_{\mathcal{T}} D$. Since also $D \prec_{\mathcal{T}} E$, and D writes, we have that $\Phi(l, E) \neq B$. Thus, the pair in the figure does not belong to NN, proving the theorem. ■

We can now conclude that location consistency is strictly stronger than NN-dag consistency.

Corollary 40 *LC $\subsetneq$ NN.*

Proof: The proof follows immediately from Theorem 36 and Theorem 39. ■

4.4 Constructibility of dag-consistent models

Nothing can be said, in general, about the constructibility of dag-consistent models. Some are constructible, some are not. We have, however, results for the four models we are concerned with. In this section, we prove that NN, WN, and NW are not constructible, and that WW is constructible.

Theorem 41 *The memory models NN, WN, and NW are not constructible.*

Proof: We first prove that NN is not constructible. The same proof also applies to WN.

Consider Figure 4-4. The dag on the left of the dashed line is a prefix of the whole dag, and a valid NN observer function Φ is shown below the nodes. There is one memory location l, which is understood.

We now prove that it is not possible to NN-extend the observer function to node F. There are three writes in the computation. It cannot be that $\Phi(l, F) = A$, because $\Phi(l, B) \neq A$. It cannot be that $\Phi(l, F) = B$, because $\Phi(l, C) \neq B$. Finally, it cannot be that $\Phi(l, F) = D$, either, because $\Phi(l, E) \neq D$.

We now prove that NW is not constructible. The proof is similar to the proof for NN, but we need a different computation, shown in Figure 4-5. As before, the reader can verify that the figure shows a valid NW observer function, but there is no way to extend it to the new node F.

■

Theorem 42 *The memory model WW is constructible.*

Proof: Recall that WW is monotonic (see Section 4.1). Therefore, by Theorem 29, it is sufficient to show that for all $(G, \Phi) \in$ WW and for all memory operations $\mathcal{O}$, there exists an observer function Φ' such that we have $(aug_{\mathcal{O}}(G), \Phi') \in$ WW and $\Phi'|_G \equiv \Phi$.

Let $(G, \Phi) \in$ WW, and let $\mathcal{O}$ be any memory operation. Let $G' = aug_{\mathcal{O}}(G)$, and Φ' be an extension of Φ to G'. The extended observer function Φ' is completely specified, except for the final node $\mathit{final}(G)$. Let $\mathcal{T}$ be a topological sort of G'. We complete the definition of Φ' as follows:

$$\Phi'(l, \mathit{final}(G)) = \mathcal{L}_{\mathcal{T}}(l, \mathit{final}(G)) \ .$$

It is immediate that Φ' is an observer function. We now prove that $(G', \Phi') \in$ WW.

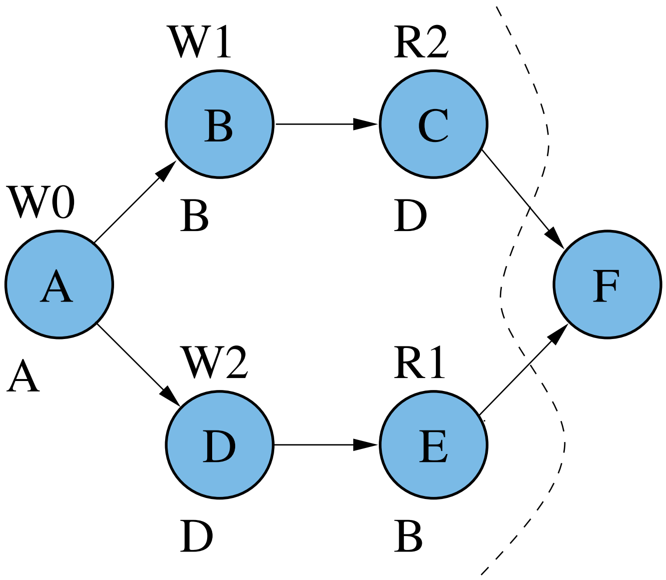


Figure 4-4: Computation used in Theorem 41. There is one memory location, which is understood. The name of a node is shown inside the node. The dag on the left of the dashed line is a prefix of the whole dag, and a valid NN observer function for the prefix is shown below the nodes.

We first prove that Property 1 of Definition 14 is satisfied. If $u \in G$, then $\Phi'(l, u) = \Phi(l, u)$ and the property is true because Φ obeys the definition of WW. Moreover, by construction, if $\mathrm{W}(l, \mathit{final}(G))$ then $\Phi'(l, \mathit{final}(G)) = \mathit{final}(G)$, thus verifyng the property.

We now prove Property 2 of Definition 14. Consider any triple of u, v, and w such that $u \prec v \prec w$, and moreover we have $\mathrm{W}(l, u)$ and $\mathrm{W}(l, v)$. If $w \in G$, we have that Φ' verifies the property, because Φ does. Otherwise, we have that $w = \mathit{final}(G)$. Because of the definition of last writer and the assumption that $\mathrm{W}(l, u)$ holds and that $\mathrm{W}(l, v)$ holds, we never have that $\Phi(l, w) = \Phi(l, u)$, and therefore the property holds trivially. This concludes the proof of the theorem. ∎

4.5 LC is the constructible version of NN

In this section we prove that NN* is equivalent to LC. This is a major result of this thesis, and it implies that, whenever you want to implement NN, you must also implement LC. From a practical point of view, NN and LC are thus the same model.

Our strategy is to prove that NN* is both weaker and stronger than LC, wherefore they are the same set. The difficult part is to prove that NN* $\subseteq$ LC.

Lemma 43 *NN** $\subseteq$ *LC*.

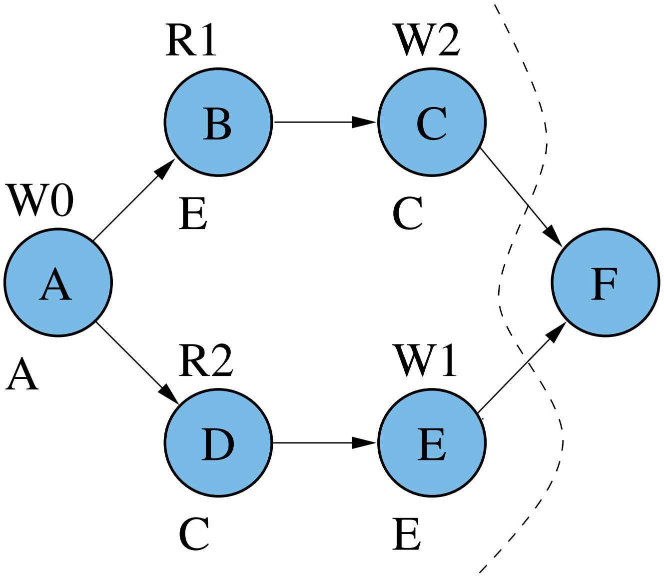


Figure 4-5: Computation used in Theorem 41 for proving that NW is not constructible. There is one memory location, which is understood. The name of a node is shown inside the node. The dag on the left of the dashed line is a prefix of the whole dag, and a valid NW observer function for the prefix is shown below the nodes.

Proof: Let $(G, \Phi) \in \text{NN}^*$. We want to prove that $(G, \Phi) \in \text{LC}$.

Consider a single memory location l. We claim that there exists a topological sort $\mathcal{T}(l)$ of the dag such that Φ coincides with the last writer function $\mathcal{L}_{\mathcal{T}(l)}$. If the claim is true, then the theorem follows immediately. We then prove the claim by induction on the number of nodes in G.

The base case of the empty dag holds trivially.

Suppose the claim is true for all dags with less than k nodes. We now prove it holds for all dags with k nodes.

Since $(G, \Phi) \in \text{NN}^*$ and NN^* is constructible, we know that there exists an observer function Φ' such that $(aug_{\text{no-op}}(G), \Phi') \in \text{NN}^*$. Let $w = \Phi'(l, \mathit{final}(G))$ be the value of the observer function on the final node of G, and let G' be the subdag of G consisting of all nodes u where $\Phi(l, u) \neq w$. We partition G into three parts: G', the single node w, and whatever remains (let's call it H). The dag G' does not contain w and therefore has strictly less than k nodes. Then, by inductive hypothesis, there exists a topological sort $\mathcal{T}'$ of G' such that $\Phi|_{G'}$ coincides with the "last writer" function $\mathcal{L}_{\mathcal{T}'}$.

Let $\mathcal{T}(l)$ be the following total order on G: all the nodes in $\mathcal{T}'$ in that order, followed by w, followed by any topological sort of H. By construction, Φ coincides with the "last writer" function $\mathcal{L}_{\mathcal{T}(l)}$.

If we can show that $\mathcal{T}(l)$ is a legitimate topological sort of G, then the claim is proven and the lemma follows.

To prove that $\mathcal{T}(l)$ is a legitimate topological sort of G, we reason as follows.

1. No node in H precedes w.

   This property holds because, for all $u \in H$, $u \not\prec \Phi(l, u) = w$, as stated by Condition 3.2 of Definition 3.

2. The vertex w does not precede any node in G'.

   To see why, suppose by contradiction that there exists a node $u \in G'$ such that $w \prec u$. Then $w \prec u \prec \mathit{final}(G)$. Since $\Phi'(l, w) = w = \Phi'(l, \mathit{final}(G))$, the definition of NN* implies that $\Phi'(l, u) = w$, and thus $u \notin G'$. This contradiction proves that $\mathcal{T}(l)$ is a total order on G.

■

Lemma 44 *LC* $\subseteq$ *NN**.

Proof: Theorem 36 proves that LC $\subseteq$ NN. Moreover, LC is constructible, as stated by Theorem 30. Therefore, by Condition 26.1 of Theorem 26, we have that LC $\subseteq$ NN*. ■

In summary, we have the desired equivalence of NN* and LC.

Theorem 45 *LC* $=$ *NN**.

Proof: Immediate from Lemmas 43 and 44. ■

The exact characterization of WN* and NW* is, however, an open problem. With reference to Figure 4-1, all we know is that they do not lie in the path between NN and LC (unless they coincide with LC). The previous statement follows from the fact that WN* and NW* are both constructible, and LC is the weakest constructible model stronger than NN.

4.6 The weakest reasonable memory model

It is now time to draw some conclusion after so many pages of definitions and theorems. Figure 4-1 will again be helpful for visualizing the rest of the discussion.

We have a few candidates for the weakest reasonable memory models, but most of them are inadequate for one reason or another. Let's recapitulate what we know about them, starting from bottom up.

WW is the original dag-consistent model [Blumofe et al. 1996b; Joerg 1996]. We argued in Section 3.4 that WW does not confine nondeterminism.

WN is the dag-consistent model of [Blumofe et al. 1996a]. We proved in Section 4.4 that it is not constructible.

NW is a strange model. I introduced it because of symmetry reasons, but I do not know what it means. Moreover, the same example used in Section 3.4 against WW also proves that NW does not confine nondeterminism. We also know it is not constructible. In a sense, this model combines the worst aspects of WW and WN, and I see no reason to adopt it.

NN is a "nice" model. Its definition is very symmetric: it imposes the "convexity" condition (Property 32.2) to all computation nodes, without exclusion. It is the strongest dag-consistent model. It is not constructible, but its constructible version has a simple form, that is, location consistency.

In a sense, NN is *the* dag consistent model. The original intuition of dag consistency was that some nodes (specifically, the writes) in a path should enjoy some sort of convexity. NN just says that *all* nodes should have this property.

We also know something about the constructible versions. We proved in Section 4.5 that $\mathrm{NN}^* = \mathrm{LC}$. LC obeys all the five properties (Table 3.1), and consequently we should regard it as a reasonable model. Unfortunately, I know very little about NW* and WN*. Since NW does not confine nondeterminism, I suspect that NW* has the same problem, although I do not know how to prove it. I do not see how constructibility could possibly imply the confinement of nondeterminism in this case. Unfortunately, since this property of nondeterminism confinement is not defined precisely, and NW* is not known explicitly, it is very hard to argue formally in one way or another.

I do not know WN* explicitly, either. Again, since WN *does* confine nondeterminism, I expect WN* to do the same, although I do not know how to prove it.

In conclusion, we have: LC is definitely OK, WN* is probably OK, NW* is probably unreasonable.

Given my experience in investigating these memory models, I think it is safe to say that WN* is a highly artificial model, whose explicit characterization is cumbersome. I actually have a tentative definition of WN*, that runs, more or less, as follows. For every location l there exists a topological sort $\mathcal{T}(l)$ such that, for every node u, there exists a topological sort $\mathcal{T}(u)$ such that I) the observer function is the last writer according to $\mathcal{T}(u)$, and II) $\mathcal{T}(u)$ is consistent with $\mathcal{T}(l)$ with respect to all predecessors of u (i.e., the predecessors of u appear in the same order in both topological sorts).[1] I tried some examples, and the previous definition seems right, but I do not know how to prove that it is. If the definition is right, I would certainly discard WN* because it is too complex (contrast it with the simplicity of LC).

At the semantics level, we have therefore evidence that LC is the weakest reasonable model, albeit with the caveats about WN*. No matter how we argue at the semantics level, however, the question of the weakest reasonable memory model will eventually be resolved by implementation issues. Results on this side also suggest that location consistency is the model of choice.

[1] I have the feeling that *every* constructible model must be definable in terms of topological sorts and last writers, but this statement is rather mystical at this point.

The BACKER coherence algorithm has been proven fast both analytically [Blumofe et al. 1996a] and empirically [Blumofe et al. 1996b]. Indeed, BACKER is the *only* coherence algorithm for which there is any kind of performance guarantee. We now know, thanks to Luchangco [1997], that BACKER maintains location consistency. Indeed, since BACKER is defined as operating on every location independently of other locations, it cannot support more than location consistency, and therefore location consistency seems to be the exact semantics of BACKER. Given this situation and our results about constructibility, I conclude that there is no reason to relax LC.

4.7 Summary

This chapter is a discussion and an explanation of Figure 4-1. We investigated the properties of the weak models shown in the figure, and proved that the relations among the models are as shown in the figure. In particular, we proved that $\text{NN}^* = \text{LC}$.

Most of the models are unreasonable for some reason. We argued that only LC and WN^* are possible candidates for the weakest reasonable memory model, and that WN^* should be discarded too, because of its complexity. Analytical and experimental results that were proved by myself and other people in the past also prove that LC can be maintained using simple and efficient algorithms.

We next consider the degrees of the angels in their hierarchies and orders, for it was said above that the superior angels illumine the inferior angels, and not conversely.

Under this head there are eight points of inquiry: (1) Whether all the angels belong to one hierarchy? (2) Whether in one hierarchy there is only one order? (3) Whether in one order there are many angels? (4) Whether the distinction of hierarchies and orders is natural? (5) The names and properties of each order. (6) The comparison of orders to one another. (7) Whether the orders will outlast the Day of Judgment? (8) Whether men are taken up into angelic orders?

Thomas Aquinas, *Summa Theologica*, Question CVIII

Chapter 5

Conclusion

The motivation of this thesis is best expressed in a well-reasoned position paper by Gao and Sarkar [1997]. Gao and Sarkar argue that it is essential to adopt an end-to-end view of memory consistency that must be understood at all levels of software and hardware. They say that this goal is possible with a memory consistency model based on partial order execution semantics. I agree wholeheartedly with their viewpoint. In fact, at this point it seems so natural to me that I would have given the computation-centric/partial order framework as an obvious axiom, if my advisor had not forced me to explain to others what I was talking about.

Indeed, this thesis already provides some answers to the questions that are raised in Gao and Sarkar's paper. They say that the primary open question is to identify a "sound but simple specification of memory consistency models based on partial order execution semantics." I believe that the framework developed in Chapter 2 is a good solution to this problem (although not perfect). Observer functions, while very simple, have proven to be a useful device for defining memory models and understand their properties. Gao and Sarkar think that it is important to do research in the "design and implementation of (more) scalable cache consistency protocols for shared-memory multiprocessor". The previous work on the BACKER algorithm [Blumofe et al. 1996b; Blumofe et al. 1996b] already resulted into a scalable cache consistency protocol, which is also *provably efficient*.

Nonetheless, many open research questions still remain in this area. I will now try to formulate the issues I think are most interesting.

On the idea that weaker is simpler I have the feeling that it is not really possible to implement anything less than location consistency. For example, BACKER keeps multiple incoherent copies of objects, and yet it supports location consistency. I have studied the protocol for GS-location consistency by Merali [Merali 1996]. It is an efficient protocol that, among other things, is supposed to prove that GS-location consistency is a good idea, because it can be implemented efficiently. I suspect that the protocol actually supports location consistency, however. Although I do not have a formal proof, I could not find a single instance in which the protocol violates NN-dag consistency. If the protocol supports

NN-dag consistency, it must also support location consistency, because of my results on constructibility.

While developing BACKER, our group at MIT thought that dag consistency was a good model, in part because BACKER supports it efficiently. In the same way, Gao and Sarkar argue that GS-location consistency is a good model because they can support it efficiently. In both cases, the argument is bogus: you have to make sure that your consistency protocol does not actually implement a stronger model. Otherwise, adopt the stronger model. Neither BACKER nor, for what I can see, the consistency protocol by Merali supports as weak a model as for what it was designed. Luchangco has shown that BACKER actually supports LC. It would be nice to know for sure what Merali's algorithm exactly does.

Location consistency imposes extremely weak conditions, and I suspect you have to work hard if you want something weaker. In other words, I think that supporting anything weaker than location consistency actually requires a *more expensive* protocol. Here is an argument by which it might actually be possible to prove this counterintuitive statement. Location consistency is maintained by BACKER with one bit of state per cache location (the dirty bit), plus knowledge of whether a location is in the cache or not. When you read a value, you either read the cache or main memory, i.e., you have two choices. Suppose you want a weaker model (i.e, more choices) than location consistency. Then you must keep more bits of state to keep track of the various choices that are available, resulting in a more expensive protocol. I also suspect that every system composed of a home location and caches, where the caches only communicate with the home location, must support at least location consistency.

I would really like to know of a consistency protocol that is good for anything and which provably *does not* maintain location consistency.

On WN-dag consistency I do not know any simple definition of WN^*. I suspect that every definition of WN^* in "closed form" is messy. This open problem demands a more intensive theoretical study.

On the reasonableness of location consistency The five properties that we regarded as necessary for reasonableness may not be sufficient. Many reasonable algorithms exist that cannot be programmed with location consistency alone (for example, a parallel hash table). My current intuition is that location consistency is the right default, and that special cases should be treated specially with stronger rules. I do not know what the stronger rules are, however, and, more importantly, how to implement them efficiently.

On constructibility Constructibility is a necessary condition for the existence of an online consistency algorithm maintaining a memory model exactly, but it is probably not sufficient. It would be nice to identify what the conditions are under which an exact consistency algorithm for a memory model exists.

On release consistency I have discussed with Leiserson, Müller, and Randall [1997] the possibility of defining a sort of "release consistency" within the computation-centric framework. The basic idea is to augment the computation model with locks, whose purpose is to introduce new edges in the dag. We then apply the existing LC model to the resulting dag. This model does not correspond exactly to the release consistency model from [Gharachorloo et al. 1990], because it specifies semantics for ill-formed programs, while release consistency does not. Nonetheless, this approach to release consistency yields to the first definition of release consistency that I could understand.

One important consequence of providing semantics for ill-formed programs is that memory operations performed outside the critical regions are given meaningful semantics, which appears to be desirable in certain applications. For example, suppose a program is creating a certain data structure, and then it inserts a pointer to it in a shared queue, which is arbitrated by means of a lock. As soon as a computation node unlocks the queue, the pointer *and* the data structure become visible to the next node that acquires the lock. I am optimistic that further research on this area will clarify the tradeoffs between these stronger semantics and the efficiency of an implementation.

Acknowledgements

Bobby Blumofe, Chris Joerg, Charles Leiserson, and Keith Randall were part of the group that first tried to define and implement dag consistency. Bobby first invented WW-dag consistency, initiating the whole topic. Keith first proposed to use a function as a device to force computation nodes to fix their viewpoint on memory. Observer functions are a formalization of his idea. Chris had the amazing ability to point out, on the fly, many subtle differences among the memory models that we were investigating.

Victor Luchangco, while attempting to understand dag consistency, first suggested the "anomaly" in Figure 4-5. His observation led me to the definition of NN, and to the definition of constructibility.

Arvind has been a constant source of inspiration. In the early days he complained that he "did not understand what dag consistency means". Very naively, the dag consistency authors believed that dag consistency was obvious and did not listen to Arvind's complaints. If anything, this thesis proves that Arvind had good reasons to worry, and was indeed right since the beginning.

Silvia Müller, Vivek Sarkar, and Nir Shavit provided many ideas and suggestions. Mingdong Feng patiently read a draft of this thesis and suggested bug fixes.

Finally, it has been a honor and a pleasure to work with Charles Leiserson, without whose continuous support and encouragement this work would not have been possible, and without whose infinite patience this work would have been completed a long time ago.

Bibliography

[Adve and Gharachorloo 1995] ADVE, S. AND GHARACHORLOO, K. 1995. Shared memory consistency models: A tutorial. Technical Report 9512 (Sept.), Rice University. Also available at http://www-ece.rice.edu/ece/faculty/Adve/publications/models_tutorial.ps.

[Adve and Hill 1990] ADVE, S. V. AND HILL, M. D. 1990. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, May 1990), pp. 2–14.

[Arvind et al. 1996] ARVIND, MAESSEN, J. W., NIKHIL, R. S., AND STOY, J. 1996. Lambda-S: an implicitly parallel lambda-calculus with letrec, synchronization and side-effects. Technical report (Nov), MIT Laboratory for Computer Science. Computation Structures Group Memo 393, also available at http://www.csg.lcs.mit.edu:8001/pubs/csgmemo.html.

[Bellman 1957] BELLMAN, R. 1957. *Dynamic Programming*. Princeton University Press.

[Bershad et al. 1993] BERSHAD, B. N., ZEKAUSKAS, M. J., AND SAWDON, W. A. 1993. The Midway distributed shared memory system. In *Digest of Papers from the Thirty-Eighth IEEE Computer Society International Conference (Spring COMPCON)* (San Francisco, California, Feb. 1993), pp. 528–537.

[Blumofe 1995] BLUMOFE, R. D. 1995. *Executing Multithreaded Programs Efficiently*. Ph. D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[Blumofe et al. 1996a] BLUMOFE, R. D., FRIGO, M., JOERG, C. F., LEISERSON, C. E., AND RANDALL, K. H. 1996a. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (Padua, Italy, June 1996), pp. 297–308.

[Blumofe et al. 1996b] BLUMOFE, R. D., FRIGO, M., JOERG, C. F., LEISERSON, C. E., AND RANDALL, K. H. 1996b. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium* (Honolulu, Hawaii, April 1996).

[Blumofe et al. 1995] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (Santa Barbara, California, July 1995), pp. 207–216.

[Boethius 512] BOETHIUS, A. M. S. 512. Contra Eutychen et Nestorium.

[Dubois et al. 1986] DUBOIS, M., SCHEURICH, C., AND BRIGGS, F. A. 1986. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (June 1986), pp. 434–442.

[Gao and Sarkar 1994] GAO, G. R. AND SARKAR, V. 1994. Location consistency: Stepping beyond the barriers of memory coherence and serializability. Technical Report 78 (Dec.), McGill University, School of Computer Science, Advanced Compilers, Architectures, and Parallel Systems (ACAPS) Laboratory. Revised December 31, 1994. Available at `ftp://ftp-acaps.cs.mcgill.ca`.

[Gao and Sarkar 1997] GAO, G. R. AND SARKAR, V. 1997. On the importance of an end-to-end view of memory consistency in future computer systems. In *Proceedings of the 1997 International Symposium on High Performance Computing* (Fukuoka, Japan, Nov. 1997). To appear.

[Gharachorloo et al. 1990] GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, June 1990), pp. 15–26.

[Goodman 1989] GOODMAN, J. R. 1989. Cache consistency and sequential consistency. Technical Report 61 (March), IEEE Scalable Coherent Interface (SCI) Working Group.

[Halstead 1997] HALSTEAD, R. H., JR. 1997. Personal communication.

[Hennessy and Patterson 1996] HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: a Quantitative Approach* (second ed.). Morgan Kaufmann, San Francisco, CA.

[Iftode et al. 1996] IFTODE, L., SINGH, J. P., AND LI, K. 1996. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (Padua, Italy, June 1996), pp. 277–287.

[Joerg 1996] JOERG, C. F. 1996. *The Cilk System for Parallel Multithreaded Computing*. Ph. D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

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

[Leiserson et al. 1997] LEISERSON, C. E., MÜLLER, S., AND RANDALL, K. H. 1997. Personal communication.

[Luchangco 1997] LUCHANGCO, V. 1997. Precedence-based memory models. In *Eleventh International Workshop on Distributed Algorithms (WDAG97)* (1997).

[Merali 1996] MERALI, S. 1996. Designing and implementing memory consistency models for shared-memory multiprocessors. Master's thesis, McGill University, Montréal, Canada.

[Scheurich and Dubois 1987] SCHEURICH, C. AND DUBOIS, M. 1987. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture* (Pittsburgh, PA, June 1987), pp. 234–243.