1 Overview

In this thesis, I propose to develop a formal framework for specifying and reasoning about models for distributed memory, particularly memories with weak consistency guarantees. The main audience for this framework is programmers of systems or machines that support some form of concurrent processes which share memory, so they can reason about the correctness and performance of their programs. It should also be useful for implementors of such systems, in determining what guarantees their implementations provide, and in assessing the advantages of providing one memory model over another. This should be the foundation for a simple formal theory that captures key concepts and mechanisms proposed in contemporary memory models, and can be extended to accommodate new memory guarantees, synchronization primitives, and programming disciplines as they are proposed.

The framework should be expressive enough to specify various memory models that have already been proposed, and should define concepts that make it easy to express both the programmers' intentions and the guarantees provided by system implementors. It should be completely formal, yet tractable, so that it can be used to prove theorems about these memory models rigorously. It should support a high level of abstraction, since its primary audience is programmers who should not need to think about the details of the system design, and so that it can model memories provided by tightly controlled hardware, as well as memories implemented by loosely coupled message passing systems. It should also be as general as possible, allowing arbitrary data types, and a wide variety of synchronization primitives and programming disciplines. Finally, it should fit into a larger theory of distributed systems, so that it can be used to prove the correctness of implementations of the memory model, and of systems that use the memory model.

The theory should include characterizations of common data types such as read/write memory, and properties of memory operations, synchronization primitives, and programming disciplines that are useful in characterizing the behaviors allowed by a memory model. Since the primary reason for using weakly consistent memory models is to improve performance, the theory should also indicate some means to compare the performance of programs running under different models.

2 Background and Motivation

Distributed implementations of shared memory arise both in networks, such as in shared databases, and in multiprocessor architectures. (Although not traditionally considered distributed systems, multiprocessors look like distributed systems inside.) As such systems become commonplace, it is important to develop convenient ways to program them. Ideally, we would like programs for these systems to express naturally the programmer's intention, and to be easy to understand and reason about carefully. It should also be possible to implement them efficiently, exploiting locality, re-ordering and other techniques that mask disk and communication latency, to deliver high performance.

Because processes may access the data in shared memory concurrently, a shared memory system must provide programmers with a conceptual model, called the *memory consistency model*, for the semantics of such concurrent access. This consists of restrictions on the possible values that memory operations may receive. For example, *sequential consistency*, proposed by Lamport [Lam79], requires that the memory should appear as if all memory accesses were handled by a single process. This model is attractive because it provides programmers with a simple and intuitive model, the

natural extension to the memory model of a uniprocessor.

However, in some systems [BLNS82, FM82], communication delays and failures make this model difficult to maintain. Because some applications, such as Web-based data servers and naming services such as DNS [DNS90], do not require such strong consistency guarantees, some systems are willing to tolerate some inconsistency in order to improve performance. Furthermore, because of performance-oriented architectural features such as caching, write-buffering and pipelining, as well as compiler optimizations such as re-ordering of code, many modern multiprocessors, such as the DEC Alpha, IBM PowerPC, Sun SPARC, and Stanford FLASH [Sit92, SW95, MSSW94, SUN91, WG94, KOH+94], do not guarantee sequential consistency. It is important to specify weaker memory consistency models that are maintained by these systems, to enable application programmers to reason about their programs.

Experience has shown that distributed systems are difficult to reason about in general. (For example, Lampson and Shvartsman make this claim in their course [LS97, Handout 16].) There are at least two different elements that contribute to the difficulty in reasoning about concurrent accesses to shared memory. The first, called *concurrency*, refers to situations where two or more "threads of control" are "active" simultaneously, and may interfere with each other, by accessing the same parts of memory, for example. We assume interleaving semantics, so that no two atomic actions can actually occur simultaneously. The difficulty with concurrency is that processes can no longer be characterized by a simple input/output relation, but instead must be modelled by *reactive* systems, characterized by behaviors, or sequences of input and output actions.[1] However, this by itself does not impel us to adopt weaker memory models. For example, a time-sharing system may have many concurrent processes, but it can easily maintain a sequentially consistent memory. Rather it is the second element, *distribution*, that makes weaker models important.

Intuitively, distribution refers to the distance between various parts of a system. Abstractly, this is manifested by different costs associated with communication between these parts. This does not mean there is concurrency in the sense indicated above. For example, caching is a technique for dealing with distribution even with only one processor, reducing the overhead associated with accessing main memory. In the absence of concurrency, techniques to hide distribution typically do so transparently, and should be viewed as optimizations that do not affect the specification of the system (except for its performance). However, transparency is much more difficult to maintain in the presence of concurrency.

Effective programming of distributed systems has typically been done in one of two ways. Most commonly, one obeys a programming discipline that has been proven to provide strong guarantees about the way actions interleave. This may be considered "easy concurrent programming", because the strong guarantees make it easy to reason about the correctness of the programs. For example, shared memory may only be accessed in *critical sections* protected by *locks*, which must be "acquired" by following a particular locking discipline, such as two-phase locking. This guarantees that critical sections appear atomic. Without such a discipline, programmers must reason directly about their programs. Because there are many ways in which actions may interleave, and intuition often turns out to be mistaken when reasoning about concurrent programs, this is usually a tedious and error-prone process, and thus is sometimes called "hard concurrent programming".

[1] Reactivity and concurrency are not exactly the same; a system that interacts with its environment, even if it has a single thread of control, is reactive. However, if we view the environment as a separate thread of control, then the system and its environment are concurrent.

Providing a formal framework for describing shared memory models, and developing a theory of how to reason about these models which includes programming disciplines, will serve programmers of distributed systems by providing a variety of "easy concurrent programming" disciplines, each with its own guarantees, so a programmer may choose a discipline that best suits the requirements of the program being written. Furthermore, specifying models completely formally within a common framework should make it easier to reason carefully about the behaviors of systems, even when no prescribed discipline is being obeyed.

In addition, clear and unambiguous definitions that capture important characteristics of memory models and programming disciplines, and demonstrations of how these relate to each other should serve as a basis for building a better intuition for reasoning informally about these memory models. This is important for programmers who must often reason informally as they decide the general structure of their programs, before writing them up and reasoning about them formally. It is useful, therefore, to provide clients with the means to express what they desire directly and at as high a level of abstraction as possible, providing them with such concepts as transactions, for example, rather than lower level synchronization mechanisms such as memory barriers. This improved understanding of memory models should also influence system and architecture designs, as it becomes clearer what kinds of memories are most useful to programmers. Finally, a formal framework opens the possibility of using automated tools in the development of applications.

3 Approach

Two criteria determine our approach in designing the formal framework for memory consistency models. We want our framework to be expressive enough to capture the memory models that have been and may be proposed and implemented, and we want the accompanying theory to be as simple and general as possible.

To achieve the first goal, we derive our framework by formalizing models (or simplifications of models) of existing or proposed systems and architectures. We then try to derive results that are claimed in the "folklore", many of which have no complete formal proof that we are aware of, and extensions of these results. This helps ensure that our results are relevant.

We also examine current concurrent programming practice, for clues on what mechanisms are useful for programmers, and what effects they are using these mechanisms to achieve. We will endeavor to provide a means to express these effects directly, to ease the task on the programmer. Then we will show how these can be provided by the mechanisms that are actually provided by real systems, either automatically (i.e., by compilation or having a front end mediate the system calls) or by following a programming discipline, or some combination of both.

To achieve the second goal, we develop the theory in several stages, first defining serial data types, then doing static analysis on *computations* (cf. [FL98]), and finally modelling dynamic (reactive) systems. Rather than being tied down to a specific language or logic, as many formal methods are, we formalize memory models directly in mathematics, allowing us greater flexibility in defining whatever concepts turn out to be useful. (See [Lam93] for an argument of the value of this.) For example, computations are modelled by directed acyclic graphs, and the dynamic systems are modelled by *input/output automata* [LT89].

We begin with a careful definition of the serial semantics of data types, including a characteriza-

tion of how different operators on a data type interact with each other, and of the relation between some data types. We also prove theorems about the equivalence (with various notions of equivalence) of different sequences of operators. This forms the foundation for later results establishing the equivalence of memory models.

We then do a careful study of *computations*, extending work began in [FL98]. A computation captures the way a program has unfolded in a particular execution at a particular moment in time. This allows us to examine the memory semantics separately from any linguistic and timing issues. We develop a theory of memory models based on computations, and prove as much as possible within this theory, before adding the additional complications of reactivity.

Finally, we examine dynamic systems, in which the memory access pattern may depend on the responses received for previous memory operations, as well other factors such as timing that are not modelled by computations. This allows us to model real systems and architectures more accurately. We will use I/O automata to model these systems, and the resulting theory will depend heavily on the computation-based theory, extending the results there to automata.

I/O automata [LT89] support a general theory for distributed systems, in which our framework can be fit, including structured description of systems and their components using both *parallel composition* based on shared actions and *levels of abstraction*. This allows us to take advantage of techniques developed for reasoning about I/O automata, including invariant assertions, compositional methods, and various simulation methods that support hierarchical reasoning. I/O automata also support performance and fault tolerance analysis, which is important for the systems we study.

As mentioned, our primary goal is not to prove the correctness of any particular system, nor to advocate a particular memory model, but to develop a framework which helps illuminate memory consistency models, making it easier to reason both formally and informally about them, and the correctness of programs that run on them. In our analysis, we do not focus exclusively on whether a system is correct (i.e., implements its specification), but rather on modelling the structure inherent in the system, and identifying the key characteristics of a system, particularly the interactions with its memory components.

Although the choice of programming language can have a profound effect on how memory is accessed, we will avoid linguistic issues as much as possible. Instead, we will enrich the client-memory interface so that clients can specify arbitrary precedence relations between operations, and a variety of synchronization primitives, including that a group of operations should appear atomic. Although no programming language may provide such general features, this generality is useful for modelling the range of features that may be provided by a variety of systems. It is easy to restrict it when modelling a particular system. The study of what features should be provided by a programming language, and how programs should be compiled into the memory operations considered in this thesis, is an important area of research, but falls beyond the intended scope of this thesis. Of course, we hope that the framework developed will shed light on this.

To develop and evaluate this framework, we will express some proposed memory models within it, compare these with other specifications of these models (some of which are formal), and prove theorems relating these models, especially how they can be combined with programming conventions to appear stronger than they really are. We will also model implementations of a few of these models, and prove formally that they implement the model. Finally, we will demonstrate how this framework can be used to prove application-level correctness and performance guarantees.

This work will build heavily on our earlier work on precedence-based and computation-centric memory models [Luc97, FL98], which will serve as a basis for our framework. These papers show how to model memories in which only precedence restrictions can be specified between operations. This includes many common memory consistency models, including *sequential consistency* [Lam79], the *location consistency* of Gao and Sarkar [GS95], and *per-location sequential consistency*.[2] The framework allows arbitrary acyclic precedence dependencies to be expressed between operations, without regard to which processes or processors issue these operations. This differs from the *processor-centric* approach of most of the work on multiprocessor memory models, in which dependencies between operations at different processors are only allowed for special synchronization operations. Thus, the sequential consistency and per-location sequential consistency models defined are generalizations of the standard versions found in the literature. These generalized models can express some synchronization primitives, including fork/join, cobegin/coend, and spawn/sync constructs.

4 Specific Work

In our study of memory models, we expect to discover various properties of data types and the possible relationships among their operations that will be useful in maintaining memory consistency. For example, Shasha and Snir define two operations on a read/write memory to *conflict* if they access the same location and at least one of them is a write [SS88], and use as a basis for characterizing the executions of a program. The *independence* property of [Luc97] is a refinement of the opposite notion, and is used to show that some models are indistinguishable from others to clients that obey certain restrictions. We will identify and formally define other such properties.

As mentioned earlier, we will use a computation-centric theory of memory as a basis for a theory for memory consistency in dynamic systems. A *computation* is a directed acyclic graph of operations, where edges represent the precedence dependencies between operations. A *computation-centric memory model* specifies the possible values that may be returned for these operations. Formally, a *return value function* for a computation assigns a return value for each operation in the computation. Every operation must have a return value, though it may be constant, such as acknowledgments for writes, or indicate an error, such as attempting to access a non-existent object. A computation-centric memory model specifies, for each computation, the return value functions allowed by the model.

There are two ways in which the theory developed in [FL98] is insufficient to serve as the basis for the dynamic systems theory developed in [Luc97]. First, there is no method for incorporating restrictions on the clients. We will extend the framework to incorporate models that only specify their behavior on special classes of computations, which determine the restrictions that the clients must obey. For example, a "race-free" program is one that only produces computations in which there are no *data races*, that is, concurrent conflicting operations. We can show that any memory that respects the precedence constraints specified by the clients cannot be distinguished by such a program from a sequentially consistent memory.

Second, the theory developed in [FL98] only describes models for read/write memories, and used *observer functions* rather than general return value functions to define the memory models.

[2] This is often called *coherence* in the literature [HP96]. It is called location consistency in [FL98] and [Fri98], though it is not the same as the model by that name in [GS95]. See [Fri98] for a justification of this terminology.

The additional structure of observer functions was useful in understanding some of the properties desired of memory models, but it does not readily extend to general data types in which a single instruction may read or write several locations atomically. We will extend the theory to allow general data types, building on a formal specification of serial data types, such as presented in [Luc97].

With this foundation in place, we will develop a formal characterization of the relationship between the computation-centric models and the automata-theoretic models for dynamic systems. Because computation-centric models tend to be easier to reason about and understand, we will show how to use the results for computation-centric models to derive results for the corresponding automata-theoretic models.

We will then develop both theories to encompass memory models that include restrictions that cannot be modelled with only precedence constraints. We will do this primarily by enriching the client-memory interface so that programmers can express directly the restrictions they want to impose on the execution of the program. We will again do as much of the reasoning as possible on computations, and use these results to reason about automaton memory models.

Many memories provide explicit synchronization mechanisms, such as memory barriers and locks, which can be used to further restrict the possible return values that operations may receive. Although some synchronization mechanisms, such as fork/join and spawn/sync constructs, can already be specified by the precedence constraints in our framework, others, such as locks and memory barriers, cannot be expressed in this way. We will show how such synchronization mechanisms can be incorporated into our theory by augmenting computations and client-memory automata interface with synchronization information. The synchronization information can be used to define additional memory models and also classes of restricted clients that arise from different programming disciplines. These can then in turn be used to prove more theorems relating different memory models under various programming disciplines.

In particular, we will show how to define different types of locks, and the memory models that guarantee that the locks will be "respected." We will also redefine the notion of "race-free" programs, so that operations protected by locks are not considered data races. We will show that a race-free program cannot distinguish any memory that respects both the locks and the precedence constraints from a sequentially consistent memory. This is similar to the results of Adve and Hill for *data-race-free* programs using memory that guarantees *weak ordering* [AH90] and Gharachorloo, et al. for *properly labelled programs* using *release consistent* memory [GLL+90]. We will show how to prove these results, generalized from their processor-centric contexts, in our framework.

In addition to more traditional synchronization mechanisms, we will also extend the client-memory interface so that clients can specify that a group of operations must be applied atomically. This is similar to the notion of transactions used in database systems,[3] and we will draw on that work [BHG87, LMWF94]. Most memory models provide locks or other synchronization mechanisms, which can be used to provide transactional semantics if used with an appropriate programming discipline. We believe it is better to provide an abstract means of specifying this, allowing programmers to express their intentions directly, rather than using mechanisms that may restrict

[3] In database systems, transactions may be aborted by the system in case of failure or interference from other transactions. In this case, the system must make it appear as though none of the operations were done at all. This will not be allowed in our system, as memories are not typically allowed to refuse to do an operation. However, if this behavior is desired, abortable transactions can be modeled with an explicit abort operation whose behavior is specified by the serial data type.

the possible executions in unintended ways. We can then prove that memory models with more traditional synchronization mechanisms implement this abstract model if the program obeys the appropriate discipline. This provides a higher level of programming abstraction, and also may shed light on what synchronization mechanisms are useful for programmers trying to get transactional semantics. This differs from most of the work that has been done on memory models, in which sequential consistency (or atomicity) is the strongest consistency model provided to the programmer by the memory. In such work, locks and other synchronization mechanisms play a dual role, both as a means to provide the appearance of sequential consistency for weaker consistency models, and, to implement transactions over sequentially consistent memory.

Building on this framework, we will specify several memory consistency models, and also several programming disciplines. Some of the memory models will be abstractions of real architectures, such as the Stanford DASH machine, the DEC Alpha, and the SPARC machines [LLG+92, Sit92, SUN91]. Others will be at a higher level of abstraction, including transactional semantics, that would be appropriate for a application programmer. We will also express other proposed memory models, such as Attiya and Friedman's *hybrid consistency* Midway's *entry consistency*, and Iftode, et al.'s *scope consistency*, in our framework [AF92, BZS93, ISL96]. Because these models are all processor-centric, our formalization of these memories should be generalizations of the original models, as they were for sequential consistency.

The *programmer-centric* memory models advocated by Adve and Gharachorloo [Adv93, AG95, Gha95] can also be formalized in our framework, though the approach is slightly different. Those models specify formal conditions on programs that, if satisfied, guarantee that the memory will appear sequentially consistent. We will define corresponding programming disciplines for the *data-race-free* and *properly labelled* families of programs. A memory would then be considered DRF0, for example, if, running with any client satisfying the DRF0 conditions, would only produce sequentially consistent responses.

With these memory models and programming disciplines defined, we can prove several theorems, demonstrating the relationships between various memory models, with or without particular programming disciplines. In addition to proving results corresponding to those already in the literature (e.g., release consistency implements proper labelling, for some version of each), we will also examine programming disciplines that guarantee transactional semantics over sequentially consistent (or weaker) memories augmented with locks or other synchronization operations.

Another characteristic that we think will be important is that of *determinacy*. A computation is *determinate* if all its schedules are *equivalent*, that is, every operation gets the same return value, and the final state is the same. A program is *determinate* if it produces a unique computation, and that computation is determinate. A program or computation is *determinate with respect to a memory model* if it is determinate when running using that memory model. Although many parallel programs are not determinate, we believe there are a significant set of programs that are. For such a program, it is enough to examine a single execution to determine if it is correct. We would like, if possible, to develop a formal and easily checkable list of criteria that will guarantee a non-trivial set of programs are determinate. This is similar to the result proven by Cheng, et al. about *abelian programs* in Cilk.

Finally, we will demonstrate that it is possible to prove some performance properties using our framework. This will build on work done using I/O automata to prove such properties for other systems [FGL+, DeP97]. As this work is relatively recent, we expect these results to be fairly

modest, as they are in those papers. We hope this will lay the groundwork for future research in this area, so that formal models can be used to predict the performance, as well as verify the correctness, of real systems.

5 Previous and Related Work

Our interest in this area began with the study of a lazy replication scheme for maintaining a replicated database [LLSG92], which led to the definition of *eventually-serializable data services* [FGL+96, FGL+]. From the joint panel discussion between ISCA and PODC in 1996 [ISC96], we saw that many of the ideas we used in that work could be applied to multiprocessor memories. This was reinforced while teaching the *Principles of Computer Systems* class at MIT [LL96]. In that class, we also tried to formally state and prove under what conditions several operations can be considered as a single atomic operation. This turned out to be more difficult than anticipated, and constitutes a major part of this research.

The work in this thesis draws from several different, but related, areas of research, each with a lot of previous or current relevant work, only a small subset of which can be mentioned here. It draws on the experience of programmers of concurrent systems, usually sequentially consistent ones. There is a rich tradition of such work, especially in the area of operating systems. There are many guides to concurrent and parallel programming, such as [Bir89, Sno92], that would be helpful in determining what abstractions would be useful to such programmers. Similarly, there is a rich literature on database transactions and concurrency control in database systems that is relevant to our work in characterizing transactional semantics (e.g., [BHG87, GR93, LMWF94]). Many of these issues are also covered in the Principles of Computer Systems class at MIT, most recently taught by Lampson and Shvartsman [LS97], and parts of this thesis will build on material from that course.

The main area, of course, is that of multiprocessor memory models. The earliest memory model, *sequential consistency*, was defined by Lamport in 1979 [Lam79]. This required a total ordering on all the operations that was consistent with the view of each processor. He also introduced the notion of *atomicity* [Lam86], and a theory of atomic objects [Lyn96, Chapter 13], which was generalized and renamed *linearizability* by Herlihy and Wing [HW90]. For our purposes, these two models are essentially equivalent. However, because of caching and other hardware optimizations, sequential consistency was not always guaranteed by the hardware. Instead, *coherence*, that is, per-location sequential consistency, was often considered the correctness condition [HP96]. Unfortunately, coherence alone was not sufficient for certain programming needs.

Relaxed (or weak) memory consistency models were introduced to bridge the gap between sequential consistency and coherence. These models were defined by asking which operations could be re-ordered with respect to the sequential order specified at each processor. For example, in a coherent memory, accesses to different locations may be re-ordered. Perhaps the earliest relaxed consistency model was that of the IBM/370 [IBM83].

Dubois, Scheurich, and Briggs introduced *weak ordering*, in which operations were designated by programmers as either "data" or "synchronization" operations, where data operations could be re-ordered among themselves, but not with synchronization operations, which had to be sequentially

consistent.[4] They stated general conditions sufficient to guarantee sequential consistency [DSB86, SD87].

Goodman defined the *processor consistency* model, which requires that the writes of each processor be seen in the same order by each processor, but each processor may see different orders for operations issued by different processors [Goo89].

Gharachorloo, et al. defined *release consistency*, which extends weak ordering by further differentiating synchronization operations into acquire, release, and nsync operations, each with weaker restrictions than the synchronization operations of weak ordering [GLL+90]. They also characterized a class of *properly labelled programs*, which when run using a release consistent memory, would appear as though it were running on a sequentially consistent memory. That is, properly labelled programs cannot distinguish between release consistency and sequential consistency.

Adve and Hill suggested that memory models should be defined by formal constraints on programs that guaranteed that the memory would appear sequentially consistent [AH90]. Specifically, they define a memory model as a contract between the software and the hardware, that if both sides are kept, the execution will be sequentially consistent. If, however, it is not kept, then there are no guarantees whatsoever. Adve developed this approach further in her thesis, where she defined the *sequential consistency normal form* method for defining memory models [Adv93]. In this sense, the properly labelled programs of Gharachorloo, et al. [GLL+90] define a memory model that implements by release consistency. Adve and Gharachorloo call this approach a *programmer-centric approach* to specifying memory consistency [AG95, Gha95].

There are also several memory models that were intended to capture the behavior of real multiprocessor architectures. The earliest was the IBM/370 [IBM83]. Others include the SPARC V8's *total store order* and *partial store order* [SUN91] and the V9's *relaxed memory order* [WG94], the DEC Alpha [Sit92, SW95], and the PowerPC [MSSW94]. Release consistency was designed to characterize the memory consistency guarantees of the Stanford DASH machine [LLG+92], and *entry consistency* characterizes the memory of the Midway distributed system [BZS93].

Many of the memory consistency models above are described informally, or at least not completely formally. There has been a lot of work, however, to give a completely formal characterization of memory models. Gibbons, Merritt and Gharachorloo [GMG91] defined a blocking variant of release consistency using I/O automata. Later, Gibbons and Merritt extended this work to non-blocking memories [GM92]. Attiya and Friedman give a formal definition for the DEC Alpha memory [AF94], and together with Chaudhuri and Welch extend that work to a more general framework [ACFW93]. This work is the basis for Friedman's thesis [Fri94]. Other formal work that takes a different approach than ours include the term-rewriting system of Shen and Arvind [SA97a, SA97b], and the work of Shasha and Snir, which relates to compiler optimizations [SS88].

There are several ways in which our approach differs from most of the other work in this area. First, we are completely formal, down to the specification of the data type, and up to the level of proving implementation using simulations and other standard techniques for I/O automata. Second, the framework is not processor-centric. While it is possible to annotate operations with their processors, this is not required by our framework, allowing it to model systems such as Cilk [BJK+95]. Third, it is not connected to any language or architecture, but it allows the client-memory interface to be extended easily, so that almost any language or architecture can

[4] Dubois, et al. actually used a condition called "strong ordering", which is slightly different from sequential consistency.

be modelled within it. Fourth, it requires, and allows, the programmer to express exactly what precedence constraints and what synchronization each operation has. This could be cumbersome if programmers had to write this down for every program, but good language design can make this fairly straightforward. Many other models are expressed as sequential code that can be re-ordered, which may lead to false assumptions about operation ordering. Fifth, each operation potentially has a completely different ordering of the operations which it uses to determine its effects. This is related to the previous point, in which the programmer, or the model, must explicitly specify what constraints each operation must respect. Sixth, we do not assume that sequential consistency embodies the strongest guarantees that programmers wish to have. Rather, we try to provide programmers with even higher levels of abstraction, such as transactional semantics. Seventh, we show how performance properties might be proven using our framework.

One shortcoming of our framework is that, by ignoring linguistic issues, it cannot take into account compiler optimizations, and "hints" such as pre-fetch, and register directives. It also cannot consider the global picture, as Shasha and Snir do in their work [SS88]. This is an important task, but we feel that the area is still sufficiently immature that a rigorous treatment of a restricted, though still quite broad, aspect of this problem is a valuable contribution.

References

[ACFW93] Hagit Attiya, Soma Chaudhuri, Roy Friedman, and Jennifer Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 241–250, July 1993. A detailed version appears as Technical Report LPCR 9302, Laboratory for Parallel Computing Research, Department of Computer Science, The Technion, 1993.

[Adv93] Sarita Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis, University of Wisconsin-Madison, 1993.

[AF92] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. In *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pages 679–690, 1992.

[AF94] Hagit Attiya and Roy Friedman. Programming DEC-Alpha based multiprocessors the easy way. In *Proceedings of the Sixth Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 157–166, Cape May, NJ, June 1994.

[AG95] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Research Report WRL 95/7, Digital Western Research Laboratory, 250 University Avenue, Palo Alto, CA 94301, September 1995.

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

[BFJ+96] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *10th International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Co., Reading, MA, 1987.

[Bir89] Andrew Birrell. An introduction to programming with threads. Technical Report 35, Digital Equipment Corporation, January 1989.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[BLNS82] A. Birrell, R. Levin, R. Needham, and M. Schroeder. An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, 1982.

[BZS93] Brian Bershad, Matthew Zekauskas, and Wayne Sawdon. The Midway distributed shared memory system. In *Proceedings of COMPCON'93*, pages 528–537, February 1993.

[DeP97] Roberto DePrisco. Revisiting the Paxos algorithm. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1997. Also, MIT/LCS/TR-717.

[DNS90] IETF, 1990. RFC 1034 and RFC 1035, Domain Name System.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[FGL+] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data service. *Theoretical Computer Science on Distributed Algorithms (Special Issue)*. To appear.

[FGL+96] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 300–309, Philadelphia, PA, May 1996.

[FL98] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA'98)*, pages 240–249, Puerto Vallarta, Mexico, June-July 1998.

[FM82] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability. In *Proceedings of the ACM Symposium on Principles of Database Systems 1*, pages 70–75, March 1982.

[Fri94] Roy Friedman. *Consistency Conditions for Distributed Shared Memories*. PhD thesis, The Technion, Haifa, Israel, June 1994.

[Fri98] Matteo Frigo. The weakest reasonable memory model. Master's thesis, Massachusetts Institute of Technology, 1998.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, Stanford, CA, December 1995.

[GLL+90] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GM92] Phillip B. Gibbons and Michael Merritt. Specifying nonblocking shared memories. In *Symposium on Parallel Algorithms and Architectures*, pages 158–168, June 1992.

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

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

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[GS95] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond memory coherence barrier. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages 73–76, Oconomowoc, Wisconsin, August 1995.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, second edition, 1996.

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

[IBM83] IBM. IBM System/370 Principles of Operation, May 1983. Publication Number GA22-7000-9, File Number S370-01.

[ISC96] Joint ISCA/PODC panel and discussion, May 1996. Held at the Federated Computing Research Conference in Philadelphia, PA.

[ISL96] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 277–287, Padua, Italy, June 1996.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor (87 kb). In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

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

[Lam86] Leslie Lamport. On interprocess communication, Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, April 1986.

[Lam93] Leslie Lamport. Verification and specification of concurrent programs. In *Decade of Concurrency: Proceedings of the REX Workshop*, pages 347–374, 1993.

[LL96] Butler Lampson and Nancy Lynch. POCS course notes, 1996. Available online at ftp://theory.lcs.mit.edu/pub/classes/6.826/www/6.826-top.html.

[LLG+92] Dan Lenoski, James Laudon, Kouroosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LLSG92] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Exploiting the semantics of distributed services. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[LMWF94] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[LS97] Butler Lampson and Alex Shvartsman. POCS Course Notes (Principles of Computer Systems), 1997. Available online at ftp://theory.lcs.mit.edu/pub/classes/6.826/www/6.826-top.html.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[Luc97] Victor Luchangco. Precedence based memory models. In Marios Mavronicolas and Philippas Tsigas, editors, *Distributed Algorithms* 11th International Workshop, WDAG'97, Saarbrücken, Germany, September 1997 Proceedings, volume 1320 of *Lecture Notes in Computer Science*, pages 215–229, Berlin-Heidelberg, 1997. Springer-Verlag.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[SA97a] Xiaowei Shen and Arvind. An adaptive cache coherence protocol that implements sequential consistency for dsm systems with multi-level caches. Technical Memo CSG-Memo-404, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 1997.

[SA97b] Xiaowei Shen and Arvind. Specification of memory models and design of provably correct cache coherence protocols. Technical Report CSG-Memo-398, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 1997. Work-in-progress (draft).

[SD87] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 234–243, Pittsburgh, PA, June 1987.

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

[Sno92] C. R. Snow. *Concurrent Programming.* Cambridge University Press, Cambridge, England, 1992.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[SUN91] SPARC architecture manual, January 1991. No. 800-199-112, Version 8.

[SW95] Richard L. Sites and Richard T. Witek, editors. *Alpha AXP Architecture Reference Manual.* Digital Press, 1995. Second edition.

[WG94] David L. Weaver and Tom Germond. *The SPARC Architecture Manual.* Prentice Hall, 1994. SPARC International Version 9.