Preprint of Chapter 24, pp. 699–722,
in *Parallel and Distributed Computing Handbook*,
Albert Y. Zomaya, editor.
McGraw-Hill, 1996.

Chapter X

Memory Models

Leonidas I. Kontothanassis and Michael L. Scott

Department of Computer Science

University of Rochester

Rochester, NY 14627-0226

{kthanasi,scott}@cs.rochester.edu

May 1994

Contents

The rapid evolution of RISC microprocessors has left memory speeds lagging behind: as of early 1994, typical main memory (DRAM) access times are on the order of 100 processor

cycles, and this number is expected to increase for at least the next few years. Multiprocessors, like uniprocessors, use memory to store both instructions and data. The cost of the typical memory access can therefore have a major impact on program execution time.

To minimize the extent to which processors wait for memory, modern computer systems depend heavily on caches. Caches work extremely well for instructions: most programs spend most of their time in loops that fit entirely in the cache. As a result, instruction fetches usually have a negligible impact on a processor's average number of cycles per instruction (CPI). Caches also work well for data, but not as well. If we assume a simple RISC processor that can issue a single register-to-register instruction every cycle and that can hide the latency of a cache hit, then we can calculate CPI from the following formula:

$$\begin{aligned} CPI \;\; = \;\; & \%\ non\ memory\ instructions \; + \\ & \%\ memory\ instructions \; * \; miss\ rate \; * \; memory\ access\ cost \end{aligned}$$

The percentages of memory and non-memory instructions vary with the type of application, the level of optimization, and the quality of the compiler. A good rule of thumb, however, is that somewhere between one fifth and one third of all dynamically executed program instructions have to access memory. For a memory access cost of 100 and a dynamic instruction mix consisting of one quarter memory instructions, a 4% drop in the hit rate, from 95% to 91%, translates into a 50% increase in the CPI, from 2 to 3. Maximizing the hit rate (and minimizing the memory access cost) is clearly extremely important. As we shall see in section 0.2, this goal is made substantially more difficult on shared-memory multiprocessors by the need to maintain a consistent view of data across all processors.

Performance issues aside, the memory model exported to the user can have a large effect on the ease with which parallel programs can be written. Most programmers are accustomed to a uniprocessor programming model in which memory accesses are all of equal cost, and in which every program datum can be accessed simply by referring to its name. Physical constraints on multiprocessors, however, dictate that data cannot be close to all processors at the same time. Multiprocessor hardware therefore presents a less convenient view of memory, either making some data accessible to a given processor only via a special interface

(message passing), or making some data accesses substantially more expensive than others (distributed shared memory).

In this chapter we describe a variety of issues in the design and use of memory systems for parallel and distributed computing. We focus on tightly-coupled multiprocessors, for these are the systems that display the greatest diversity of design alternatives, and in which the choice between alternatives can have the greatest impact on performance and programmability.

We begin in section 0.1 with technological issues: the types of memory chips currently available and their comparative advantages and costs. We then consider the major architectural alternatives in memory-system design (section 0.2), and the impact of these alternatives on the programmer (section 0.3). We argue that a shared-memory model is desirable from the programmer's point of view regardless of the underlying hardware architecture. Assuming a shared-memory model, we turn in section 0.4 to the issue of memory consistency models, which determine the extent to which processors must agree about the contents of memory at particular points in time. We discuss the implementation of various consistency models in section 0.5. We conclude in section 0.6 with a description of current trends in memory system design, and speculation as to what one may expect in future years.

0.1 Memory Hardware Technology

The ideal memory system from a programmer's point of view is one that is infinitely large and infinitely fast. Unfortunately physics dictates that these two properties are mutually antagonistic: the larger a memory is the slower it will tend to respond.

Memory systems are generally measured in terms of *size*, *latency*, and *bandwidth*. Size is simply the number of bytes of information that can be stored in the memory. Latency is usually characterized by two measures: *access time* and *cycle time*. Access time is the time it takes memory to produce a word of data requested by a processor, while cycle time is the minimum time between requests. Bandwidth measures the number of bytes that memory

can supply per unit of time. Bandwidth is related to cycle time but is also dependent on other memory organization features discussed later in this section. These features include the *width* of the memory and the number of memory *banks*.

There are two basic types of memory chips available in the market today: dynamic random access memory (DRAM) and static random-access memory (SRAM). DRAMs require that the data contained in the chip be occasionally rewritten if they are not to be lost. DRAM chips must therefore be made unavailable to service requests every few milliseconds, in order to refresh themselves. Furthermore read accesses to DRAM chips must be followed by writes of the data read, because a read destroys the contents of the accessed location. As a result, DRAMs have a larger cycle time than access time. SRAMs on the other hand require neither refresh nor write-back, and thus have equal cycle and access times. Unfortunately, SRAM memories require more transistors per bit than DRAMs and thus have lower density. A general rule of thumb is that the product of size and speed is constant given a particular technology level. Current SRAMs are approximately 16 times faster than DRAMs but have about 16 times less capacity.

SRAM and DRAM are not the only options when building a memory system. There are also hybrid options that take advantage of common memory-access patterns. *Cached DRAMs* [16] attempt to combine the best features of DRAM and SRAM (high density and fast cycle time) by integrating a small SRAM buffer and a larger DRAM memory on a single integrated circuit. Recently-accessed data is cached in the SRAM buffer. If the memory access pattern displays a high degree of temporal locality (see section 0.2), then many requests will be able to be serviced by the SRAM, for a faster average cycle time. The SRAM buffer in a cached DRAM is obviously slower than a processor-local cache (it is on the other side of the processor-memory interconnect), but it is automatically shared by all processors using the memory, and does not introduce the need to maintain coherence (see section 0.2.3).

Pagemode DRAMs [31] take advantage of the fact that addresses are presented in two steps to a DRAM chip. The first step specifies a row in the internal chip grid; the second

Memory Type	Size	Access Time	Cycle Time
DRAM	16Mbit	85ns	140ns
SRAM	1Mbit	8ns	8ns
Cached DRAM	16Mbit	8ns (85ns)	8ns (140ns)
Pagemode DRAM	16Mbit	30ns (85ns)	80ns (140ns)

Table 1: Performance characteristics of different memory types. Numbers in parentheses denote worst case response times.

step specifies the column. If successive accesses are to the same row then the first addressing step can be omitted, improving the access time. Table 1 presents the basic performance characteristics of current technology RAM.

The bandwidth of a memory system is related to the cycle time, but we can get high bandwidth out of slow memory chips by accessing more of them in parallel, or by *interleaving* successive memory references across different memory modules. Interleaving allows a pipelined processor to issue memory references to either DRAM or SRAM at a faster rate than the memory cycle time. Interleaving also allows a DRAM memory to re-write the accessed data before being accessed again.

Most memory chips are one to eight bits wide, meaning that a given access returns from one to eight bits of data. Since most systems access data with at least word granularity (and many retrieve many-word cache lines), several chips are used in parallel to provide the desired granularity in a single memory cycle. A collection of memory chips that can satisfy one memory request is termed a *memory bank*.

There is a strong motivation to build a system from the densest available chips, to minimize cost, physical bulk, power requirements, and heat dissipation. Unfortunately, the goals of high density and interleaving are at odds. If we use 16 Mbit chips and assume that each chip can supply 4 bits of data in one access, then 8 chips in parallel are needed to supply a 32 bit word. This means that a single bank in our system would have 16 Mbytes of

memory and our total memory size would have to increase in 16 Mbyte increments. Using lower-density DRAM to increase interleaving sacrifices much of the cost advantage over SRAM.

In practice, most memory systems for microprocessors (including those employed in multiprocessors) are currently built from DRAM, with caches at the processor. Most memory systems for vector supercomputers (including modestly-parallel machines) are built from highly-interleaved SRAM without caches. The Cray Research C-90, for example, can have up 1024 banks of memory. Historically, supercomputer workloads have not displayed sufficient locality of reference to make good use of caches, and supercomputer customers have been willing to pay the extra cost for the SRAM memory.

0.2 Memory System Architecture

Beyond issues of chip technology, there are many architectural decisions that must be made in designing a multiprocessor memory system. In this section we address three principal categories of decisions. We begin with the high level organization of memory into modules, the physical location of those modules with respect to processors and to each other, the structure of the physical address space, and the role to be played by caches. We then turn to lower-level issues in cache design, most of which pertain equally well to uniprocessors. Finally we consider the issue of coherence, which arises on shared-memory multiprocessors when copies of data to be modified may reside in more than one location.

0.2.1 High-level Memory Architecture

The most basic design decisions for multiprocessor memory systems hinge on the concept of *locality*. Programs that display a high degree of locality make heavier use of different portions of the address space during different intervals in time. Computer architects can take advantage of locality via *caching*: keeping copies of heavily used information near to

Figure 1: Levels in a memory hierarchy

the processors that use it, thereby presenting the illusion of a memory that is both large and fast.

There are two dimensions of locality of importance on a uniprocessor, and a third of importance on parallel machines:

1. *Temporal Locality.* If a data item is referenced once, then it is likely to be referenced again in the near future.

2. *Spatial Locality.* If a data item is referenced, then items with nearby addresses are likely to be referenced in the near future.

3. *Processor Locality.* If a data item item is referenced by a given processor, then that data item (and others at nearby addresses) are likely to be referenced by that same processor in the near future.

Hierarchical memory systems take advantage of locality by providing multiple levels of memory, usually organized so that each higher level contains a subset of the data in

Level Name	Capacity	Response Time
Processors Registers	64-256 bytes	0 cycles
First Level Cache	4-64 Kbytes	1-2 cycles
Second Level Cache	256-4096 Kbytes	10-20 cycles
Main memory	16-4096 Mbytes	60-200 cycles
Swap & Disk	$>$ 1Gbyte	$>$ 200,000 cycles

Table 2: Characteristics of memory hierarchy levels

the level below it. Figure 1 presents the levels in a typical memory hierarchy and table 2 summarizes typical size and speed characteristics for each level. The performance of the memory hierarchy depends heavily on the success of the cache level(s) in satisfying requests, so that the number of references that need to access slow main memory is minimized. Successful caches reduce both average memory latency and the bandwidth required of main memory and the interconnection network [36], allowing the use of cheaper parts.

Because it imposes lookup costs, a cache can actually reduce performance if the hit rate is very low. For programs without significant amounts of locality, it may make sense to build *flat* memory systems, in which all references access main memory directly. Workloads on vector supercomputers, for example, have traditionally shown little locality, and require more memory bandwidth than can be provided by a typical cache. As a result, supercomputers are often built with flat memory. For the sake of speed, this memory usually consists of highly-interleaved SRAM, and is a major—perhaps the dominant—component in the cost of these machines. Even so, supercomputer compilers must employ aggressive prefetching techniques, and supercomputer processors must be prepared to execute instructions out of order, to hide the latency of memory.

Current hardware and software trends suggest that caches are likely to become more effective for future supercomputer workloads. Hardware trends include the development of very large caches with multiple banks, which address the bandwidth problem. Software

Figure 2: Simplified Distributed and Dance-Hall Memory Architecture Multiprocessors

trends include the development of compilers that apply techniques such as blocking [7] to increase locality of reference.

Independent of the existence of caches, designers must address the question of where to locate main memory. They can choose to co-locate a memory module with each processor or group of processors, or to place all memory modules at one end of an interconnection network and all processors at the other. The first alternative is known as a *distributed memory architecture*; the second is known as a *dance-hall* architecture.[1] Figure 2 depicts the basic difference between the distributed and dance-hall designs. Distributed memory architectures improve the scalability of systems when running applications with significant amounts of processor locality. This scalability comes at the expense of a slightly more complicated addressing scheme: node controllers must figure out where in the system to send each memory request that cannot be satisfied locally.

Dance-hall architectures dominate among small-scale multiprocessors, where all system components can share a single bus. The bus makes it easy for modules to monitor each other's activities, e.g. to maintain cache coherence. Large-scale dance-hall machines have also been designed. Examples include the Illinois Cedar machine [47], the BBN Monarch proposal [67], and the forthcoming Tera machine [4]. The dominant wisdom, however, holds

[1] Dance-hall machines take their name from the image of boys and girls standing on opposite sides of a high school dance floor.

that scalability issues dictate a hierarchical distributed memory organization for large-scale multiprocessors, and the trend toward such machines is likely to continue.

There are several important classes of distributed memory machines. The simplest are the so-called *multicomputers*, or *NORMA* (no remote access) machines. In these each processor is able to access only its local memory. All communication between processors is by means of explicit *message passing*. Commercially-available NORMA machines include the Intel Paragon, the Thinking Machines CM-5, the NCube 2, and a variety of products based on the INMOS Transputer.

The remaining classes of distributed memory machines share with the dance-hall machines the provision of a single global physical address space. The machines in these classes are sometimes referred to as a group as *shared-memory multiprocessors*. Among them, the simplest are the so-called *NUMA* (non-uniform memory access) multiprocessors. Modern NUMA machines have caches, but the hardware does nothing to keep those caches consistent. Examples of NUMA machines include the BBN TC2000, the Hector machine at the University of Toronto [73], the Cray Research T3D, and the forthcoming Shrimp machine [12].

The most complex of the large-scale machines are those that maintain cache coherence in hardware, beyond the confines of a single bus. Examples of such machines include the commercially-available Kendall Square KSR-1 and 2, the Convex Exemplar (based on the IEEE Scalable Coherent Interface standard [43]), and the Dash [52], Flash [48], and Alewife [3] research projects. Alternative approaches to cache coherence are discussed in section 0.4.

A cache-less dance-hall machine is sometimes called an *UMA* (uniform memory access) multiprocessor. Several companies produced such machines prior to the RISC revolution, when processor cycle times were closer to memory cycle times. The Tera machine will be an UMA, but will incorporate extremely aggressive techniques to hide memory latency. The UMA name is sometimes applied to bus-based machines with caches, but this is an abuse of terminology: caches make memory highly non-uniform.

Figure 3: Lookup in a typical cache organization

0.2.2 Basic Issues in Cache Design

There are several metrics for judging the success of caches in a hierarchical memory system, the most common being *hit rate*, *miss rate* and *mean cost per reference (MCPR)*. Hit rate is defined as the percentage of a program's total memory references that are satisfied by the cache; miss rate respectively indicates the percentage of references that could not be satisfied by the cache. The hit rate and miss rate sum to one.

Figure 3 illustrates a typical single level cache organization and its cache lookup mechanism. When a tag check succeeds (the desired tag is found in the cache), we have a cache hit and data is returned by the cache. When a tag check fails we have a miss and data must be fetched from main memory, possibly replacing some data already in the cache.

MCPR is defined to be the average cost of a memory access. It provides a more detailed performance measure for a cache, because it takes into account the cost of misses. If we assume a single level of cache, then the formula for MCPR is as follows:

$$MCPR = HitRate * HitTime + MissRate * MissTime$$

The actual MCPR observed by an application depends on many factors. Some of these are application dependent, while others are architecture dependent. The application dependent factors involve the locality properties of the application and will be discussed in more detail in section 0.3. The architecture dependent parameters are: **cache size, cache block size, associativity, tag and index organization, replacement policy, write policy, and choice of coherence mechanism**. These parameters can affect performance by changing the cost of cache hits,[2] changing the cost of cache misses, or changing the ratio of hits and misses.

With the exception of the coherence mechanism which will be discussed in section 0.2.3 the remaining parameters apply equally well to uniprocessor cache architectures, and are discussed in most computer architecture books [41].

0.2.3 The Cache Coherence Problem

Simply stated, the *coherence problem* is to ensure that no processor reads a stale copy of data in a system in which more than one copy may exist. The coherence problem arises on uniprocessors equipped with direct memory access (DMA) I/O devices, but it can generally be solved in this context via ad-hoc OS-level mechanisms. The problem also arises in multiprocessors with caches, and is not as easily solved.

In small scale multiprocessors the coherence problem has been addressed by *snooping* on a shared bus. A processor writing a shared datum broadcasts its action on the bus. The remaining processors monitor the bus for all transactions on shared data and take whatever actions are needed to keep their caches consistent. The main problem with this approach is

[2]It is common but not entirely accurate to assume that cache hits cost a single cycle. In reality, most on-chip caches take 2 or 3 cycles to respond. The compiler attempts to hide cycles beyond the first by scheduling independent operations in the next 1 or 2 instructions, but it cannot always do so.

that the bus is a serial bottleneck, and limits system scalability. With current technology, buses can supply as much as 1.2 Gbytes/sec of data, while processors may consume data at a peak rate of as much as 200 Mbytes/sec. At these rates fewer than ten processors suffice to saturate the bus. In practice caches satisfy most of the processor requests, but even so the number of processors that can successfully share a bus is limited to 20 or 30.

In the absence of a fast, system-wide broadcast mechanism, the cache coherence problem is addressed by maintaining some form of directory data structure [20]. A processor accessing a shared datum consults the directory, updates it if necessary, and sends individual messages to all processors that may be affected by its actions, so that they can keep their caches consistent. Directory data structures have to maintain information for every cache line and every processor in the system. The obvious organization poses serious scalability problems, since the memory overhead for directory information increases with the square of the number of processors ($\Theta(P^2)$) [21]. Fortunately, studies indicate that most data are shared among a relatively small number of processors, even on very large machines [39], so a directory structure can be effective while maintaining only a small number of pointers for each sharable line [22]. In the exceptional case in which the small number of pointers is not sufficient, the system can emulate broadcast with point-to-point messages, or use dynamically allocated (slower) memory to store the additional pointers.

Most directory-based machines store the information about a given cache line at the processor that holds the corresponding portion of main memory. Machines of this sort are said to have a *CC-NUMA* (cache-coherent NUMA) architecture. An alternative approach is to treat all of a processor's local memory as a secondary or tertiary cache, with a more dynamic directory structure and with no particular designated location for a given physical address. Machines of this sort are referred to as *COMA* (cache-only memory architecture) [40].

The Stanford Dash machine [52] uses the $\Theta(P^2)$ directory organization. The MIT Alewife machine has a limited number of pointers for each directory entry; it traps to software on overflow. The Convex Examplar is based on the IEEE SCI standard [43], which maintains a distributed directory whose total space overhead is linear in the size of the sys-

tem's caches. The newer Stanford Flash machine [48] has a programmable cache controller; its directory structure is not fixed in hardware. The KSR-1 and 2 are COMA machines; their proprietary coherence protocol relies in part on hardware-supported broadcast over a hierarchical ring-based interconnection network. The Swedish Data Diffusion Machine [40] is also a COMA, with a hierarchical organization whose space overhead grows as $\Theta(P \log P)$.

Independent to the scalability issue is the question of the actual mechanism used to maintain coherence. The available alternatives include *write-invalidate* and *write-update* [29]. Write-invalidate makes sure that there is only a single copy of a datum before it can be written, by sending invalidation messages to processors that may currently have a copy of the cache line in which the datum resides. Subsequent accesses to this line by processors other than the writer will contact the writer for the latest value of the data. Write-update makes sure that all copies are consistent by sending new values as they are written to every processor with a copy of the line in its cache. The basic tradeoff is that write-update leads to lower average latency for reads, but generates more interprocessor communication traffic. In bus based multiprocessors the tradeoff has traditionally been resolved in favor of write-invalidate [44, 63], because the interconnection network (the bus) is a scarce resource.

In large scale multiprocessors with network-based interconnects the latency for cache misses on read accesses is the most serious impediment to parallel program performance. Write update helps reduce this latency at the expense of higher network traffic. Depending on the amount of bandwidth available in the system and the sharing pattern exhibited by the program, write-update may yield better performance than write-invalidate. Hybrid protocols that choose between the two mechanisms dynamically have also been suggested [72] and have been shown to provide performance advantages over each of the individual mechanisms in isolation. The question of which is the best coherence mechanism for large scale multiprocessors is still a topic of active research.

Techniques that tolerate (hide) latency can serve to tilt the balance toward write-invalidate protocols. Examples of such techniques include aggressive data prefetching [60] and the use of *micro-tasking* processors [2, 5], which switch contexts on a cache miss. Ex-

tensive discussion of the issues encountered in designing coherence protocols can be found in several surveys [10, 38, 55, 71].

The choice of cache line size also has a significant impact on the efficiency of coherence mechanisms. Long cache lines amortize the cost of cache misses by transferring more data per cache miss in the hope of reducing the miss rate. In this sense they constitute a form of hardware-initiated prefetching. At the same time, large cache lines can lead to *false sharing* [14, 27, 30], in which processors incur coherence overhead due to accesses to non-overlapping portions of a line. False sharing results in both higher miss rates and useless coherence traffic. Conventional wisdom has historically favored short cache lines (8-32 bytes) for multiprocessor systems, but the evolution of compiler techniques for partitioning data among cache lines, and the effort by programmers to produce programs with good locality, will likely allow future machines to use longer cache lines safely.

0.3 User-Level Memory Models

0.3.1 Shared Memory v. Message Passing

Memory models are an issue not only at the hardware level, but also at the programmer level. The model of memory presented to the user can have a significant impact on the amount of effort required to produce a correct and efficient parallel program. Depending on system software, the programmer-level model may or may not resemble the hardware-level model.

We have seen that hardware memory architectures can be divided into *message-passing* and *shared-memory* classes. In a very analogous way, programmer-level memory models can be divided into message-passing and shared-memory classes as well. In a shared-memory model, processes can access variables irrespective of location (subject to the scoping rules of the programming language), simply by using the variables' names. In a message-passing model, variables are partitioned among the instances of some language-level abstraction of

a processor. A process can access directly only those variables located on its own processor, and must send messages to processes on other processors in order to access other variables.

There is a wide variety of programming models in both the message-passing and shared-memory classes. In both cases, models may be implemented

- via library routines linked into programs written in a conventional sequential language;
- via features added to some existing sequential language by means of preprocessors or compiler extensions of various levels of complexity; or
- via special-purpose parallel or distributed programming languages.

Library packages have the advantage of portability and simplicity, but are limited to a subroutine-call interface, and cannot take advantage of compiler-based static analysis of program control and data flow. Languages and language extensions can use more elaborate syntax, and can implement compile-time optimizations. Because message-passing is based on *send* and *receive* operations, which can for the most part be expressed as subroutine calls, library-based implementations of message-passing models have tended to be more successful than library-based implementations of shared-memory models. The latter are generally forced to access all shared data indirectly through pointers obtained from library routines.

Andrews and Schneider [8] provide an excellent introduction to parallel programming models; a more recent book by Andrews [9] provides additional detail. Bal, Steiner, and Tanenbaum provide a survey of message-passing programming models [11]. A technical report by Cheng [23] surveys a large number of programming models and tools in use in the early 1990s. Probably the most widely-used library-based shared-memory model is a set of macros for Fortran and C developed by researchers at Argonne National Laboratory [15]. Widespread experience with library-based message-passing models has led to efforts to standardize on a single interface [26].

It is widely believed that shared memory programming models are easier to use than message passing models. This belief is supported by the dominance of (small-scale) shared-

memory multiprocessors in the market, and by the many efforts by compiler writers [32, 68, 56] and operating system developers [19, 62] to provide a shared memory programming model on top of message-passing hardware. We focus on shared-memory models in the remainder of this chapter.

0.3.2 Implementation of Shared-Memory Models

Any user-level parallel programming model must deal with several issues. These include

- specifying the computational tasks to be performed and the data structures to be used;
- identifying data dependences among tasks;
- allocating tasks to processes, and scheduling and synchronizing processes on processors, in a way that respects the dependences;
- determining (which copies of) which data should be located at which processors at which points in time, so that processes have the data they need when they need it;
- arranging for communication among processors to effect the location decisions.

Programming models and their implementations differ greatly in the extent to which the user, compiler, run-time system, operating system, and hardware are responsible for each of these issues. The programmer's view of memory is intimately connected to the final two issues, and more peripherally to all of the others.

At one extreme, some programming models make the user responsible for all aspects of parallelization and data placement. Message-passing models fall in this camp, as do some models developed for non-cache-coherent machines. Split-C [25], for example, provides a global namespace for C programs on multicomputers, with a wealth of mechanisms for data placement, remote load/store, prefetch, bulk data transfer, etc. Other simple models (e.g. Sun's LWP (Light Weight Processes) and OSF's pthreads) require the user to manage processes explicitly, but rely on hardware cache coherence for data placement.

At the next level of implementation complexity, several models employ optimizing compilers to reduce the burden of process management. Early work on parallel Fortran dialects [65] focused on the problem of restructuring loops to minimize data dependences and maximize the amount of work that could be performed in parallel. These dialects were primarily intended for vector supercomputers with an UMA memory model at both the user and hardware levels. Gelernter and Carriero's work on Linda has focused on minimizing communication costs for an explicitly-parallel language in which processes communicate via loads and stores in an associative global "tuple space" [18]. More recently, several groups have addressed the goal of increasing processor locality on cache-coherent machines by adopting scheduling techniques that attempt to place processes close to the expected current location of the data they access most often [57].

For large-scale scientific computing, where multicomputers dominate the market, much attention has recently been devoted to the development of efficient shared-memory programming models. Most notably, a large number of research groups have cooperated in the development of High Performance Fortran (HPF) [56]. HPF is intended not as an ideal language, but as a common starting point for future developments. To first approximation, it combines the syntax of Fortran-90 (including operations on whole arrays and slices) with the data distribution and alignment concepts developed in Fortran-D [42]. Users specify which elements of which arrays are to be located on which processors, and which loops are to be executed in parallel. The compiler then bases its parallelization on the *owner computes* rule, which specifies that the computations on the right-hand side of an assignment statement are performed on the processor at which the variable on the left-hand side of the assignment is located. Similar approaches are being taken in the development of pC++, a parallel C++ dialect [32], and Jade, a parallel dialect of C [68].

Many alternatives and extensions to the owner-computes approach are currently under development. Li and Pingali, for example, have addressed the goal of restructuring loops and assigning computations to processors in order to maximize data locality and minimize communication on machines with a single physical address space [54]. Saltz et al. have

developed code generation techniques that defer dependence analysis to the execution of loop prologues at run time, to maximize parallelism and balance load in programs whose dependence patterns are determined by input data [69].

Ideally, programmers would presumably prefer a programming model in which *both* parallelization *and* locality were managed by the compiler, with no need for the user to specify process creation, synchronization, or data distribution. As of 1994 no system approaches this ideal. It is possible that full automation of process and locality management will require that users program in a non-imperative style [17].

0.3.3 Performance of Shared-Memory Models

While bus-based shared memory machines dominate the market for general-purpose multiprocessors, message passing machines have tended to dominate the market for large-scale high-performance machines. There are two main reasons for this contrast. First, large-scale shared-memory multiprocessors are much harder to build than either their small-scale bus-based counterparts or their large-scale message-based competitors. Second, widespread opinion (supported by some good research; see e.g. [61]) holds that the shared memory programming paradigm is inefficient, even on machines that provide hardware support for it. We believe, however, that the observed performance problems of shared memory models are actually artifacts of the way in which those models have been used, and are not intrinsic to shared memory itself. Specifically:

- Shared memory models have evolved in large part out of concurrent ("quasi-parallel") computing on uniprocessors, in which it is natural to associate a separate process with each logically distinct piece of work. To the extent, then, that shared-memory programs are written with a large number of processes, they will display high overhead for process creation and management. This overhead can be minimized by adopting a programming style with fewer processes, or by using more lightweight thread management techniques [6].

- The shared memory programming style requires use of explicit synchronization. Naive implementations of synchronization based on busy-waiting can perform extremely badly on large machines. Recent advances in both hardware and software synchronization techniques have eliminated the need to spin on non-local locations, making synchronization a much less significant factor in shared-memory program performance [1, 58].

- Communication is a dominant source of overhead in parallel programs. Communication in shared-memory models is implicit, and happens as a side-effect of ordinary references to variables. It is consequently easy to write shared-memory programs that have very bad locality, and that therefore generate large amounts of communication. This communication was much less costly in early shared-memory machines than it is today, leading many shared-memory programmers to assume incorrectly that locality was not a concern. This should be viewed as an indictment not of shared-memory, but of the *naive use* of shared memory. Recent studies have shown that when programmed in a locality conscious way, shared memory can provide performance equal to or better than that of message passing [51, 57].

Reducing communication is probably the most important task for a programmer/compiler that wants to get good performance out of a shared memory parallel program. Improving locality is a multidimensional problem that is a topic of active research. Some of its most important aspects (but by no means the only ones) are:

Mismatch between the program data structures and the coherence units. If data structures are not aligned with the coherence units, accesses to unrelated data structures will cause unnecessary communication. This effect is known as false sharing [14, 27, 30] and can be addressed using sophisticated compilers, advanced coherence protocols, or careful data allocation and alignment by the programmer.

Relative location of computational tasks and the data they access. Random placement of tasks (as achieved by centralized work queues) results in very high commu-

nication rates. Programmer/compilers should attempt to place all tasks that access the same data on the same processor to take advantage of processor locality.

Poor spatial locality. In some cases programs access array data with non-unit strides. In that case a potentially large amount of data fetched in the cache remains untouched. Blocking techniques and algorithmic restructuring can often help alleviate this problem.

Poor temporal locality. Data is invalidated or evicted from a processor's cache before it is touched again. As in the previous case restructuring the program can help improve its access pattern and as a consequence its temporal locality.

Conflicts between data structures. Data structures that map into the same cache lines will cause this problem in caches with limited associativity. If these data structures are used in the same computational phase performance can suffer severe degradation. Skewing or relocating data structures can help to solve this problem.

Communication intrinsic in the algorithm. Such communication is necessary for correctness and cannot be removed. However it may be tolerated if data is pre-fetched before it is needed, or if the processor is able to switch to other computation while a miss is being serviced.

One might argue that if considerable effort is going to be needed to tune programs to run well under a shared memory programming model then the conceptual simplicity argument in favor of shared memory is severely weakened. This is not the case however, since tuning effort is required only for the computationally intensive parts of a shared-memory program, whereas message passing would require similar effort throughout the program text, including initialization, debugging, error recovery, and statistics gathering and printing. Furthermore most people find it easier to write a correct program and then spend effort in refining and tuning it, as opposed to writing a correct and efficient program from the outset. The principal argument for shared memory is that it provides *referential*

transparency: everything can be referenced with a common syntax, and a simple and naive programming style can be used in non-performance-critical sections of code.

0.4 Memory Consistency Models

Shared-memory coherence protocols, whether implemented in hardware or in software, must resolve conflicting accesses—accesses to the same coherence block that are made by different processors at approximately the same time. The precise semantics of this conflict resolution determine the *memory consistency model* for the coherence protocol. Put another way, the memory consistency model determines the values that may be returned by read operations in a given set of parallel reads and writes. The goal of most parallel system architects has been to exhibit behavior as close as possible to that of sequential machines. Therefore the model of choice has traditionally been *sequential consistency* [50] (see definition later in the section). Unfortunately, sequential consistency imposes a strict ordering on memory access operations, and precludes many potentially valuable performance optimizations in coherence protocols. Relaxing the constraints of sequential consistency offers the opportunity to achieve significant performance gains in parallel programs. The memory models discussed in this section achieve these gains at the expense of a slightly more complicated programming model.

Several attributes of memory references can be considered when defining a consistency model. These attributes include the locations of the data and the accessing processor, the direction of access (read, write, or both), the value transmitted in the access, the causality of the access (what other operations caused control to reach this point, and produced the value(s) involved), and the "category" of the access [59]. *Non-uniform* or *hybrid* consistency models distinguish among memory accesses in different categories; *uniform* models do not.

Of the uniform memory models the two most important are *sequential consistency* [50] and *processor consistency* [37]. Others include atomic consistency, causal consistency, pipelined RAM, and cache consistency. A concise definition of all memory models men-

Processor 0	Processor 1
X = 10	Y = 10
A = Y	B = X
Print A	Print B

Figure 4: Code segment that may yield different results under sequential and processor consistency

tioned in this section can be found in a technical report by Mosberger [59].

- Sequential Consistency: A memory 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. In simpler words this definition requires all processors to agree on the order of memory events. If for example a memory location is written by two different processors, then all processors must agree as to which of the writes happened first.

- Processor Consistency: A memory system is processor consistent if all the reads issued by a given processor appear to all other processors to occur before any of the given processor's subsequent reads and writes, and all the writes issued by a given processor appear to all other processors to occur before any of the given processor's subsequent writes. Processors can disagree on the order of reads and writes by different processors, and a write can be delayed past some of its processor's subsequent reads.[3] ("Subsequent" here refers to the ordering specified by the processor's program.)

[3]This definition is from the Stanford Dash project [33]. Processor consistency was originally defined informally by Goodman, and has been formalized differently by other researchers.

Figure 4 shows a code segment that has a potential outcome under processor consistency that is impossible under sequential consistency. Under sequential consistency, regardless of the interleaving of instructions, variable A, variable B, or both will have the value 10 so at least one 10 will be printed. Under processor consistency, however, processors do not have to agree on the order of writes by different processors. As a result, both processors may fail to see the other's update in our example, so that neither prints a 10. While processor consistency seems unintuitive it is easier to implement than sequential consistency, and it allows several important performance optimizations. Most programs written with the sequential consistency model in mind execute correctly under processor consistency.

By constraining the order in which memory accesses can be observed by other processors, sequential and processor consistency force a processor to stall until its previous accesses have completed.[4]. This prevents or severely limits the pipelining of memory requests, sacrificing a potentially significant performance improvement. More relaxed consistency models solve this problem by taking advantage of the fact that user programs do not depend on the coherence protocol alone to impose an ordering on the operations of their programs. Even with the strictest of consistency models (i.e. sequential consistency) there are still too many valid interleavings of a parallel execution. Higher level synchronization primitives are therefore used to impose a desired ordering. Most relaxed consistency models distinguish between regular and synchronization accesses, and sometimes between different categories of synchronization accesses. They then impose different ordering constraints on the accesses in different categories. Figure 5 shows a possible categorization of memory accesses [59]. Other categorizations can be found in other papers [33, 35].

Relaxed consistency models can be viewed either from an architectural point of view (what does the coherence protocol do?) or from the programmer's point of view (what does the program have to do to ensure the appearance of sequential consistency?). The two most important relaxed consistency models from the architectural point of view are

[4]A read access is considered completed when no future write can change the value returned by the read; a write access is considered completed when all future reads will return the value of this or a future write

Figure 5: Categories of memory references

weak consistency [28] and *release consistency* [33]. The most important models from the programmer's point of view are DRF0 (data-race-free 0), DRF1, and *PLpc* (properly-labeled processor consistent) [35]. Other relaxed models include TSO (total store ordering), PSO (partial store ordering), and entry consistency.

- Weak Consistency: A system is weakly consistent if a) accesses to synchronization variables are sequentially consistent, b) accesses to synchronization variables are issued (made visible to other processors) only after all of the local processor's previous data accesses have completed, and c) accesses to data are issued only after all of the local processor's previous synchronization accesses have completed.

- Release Consistency: A system is release consistent if a) accesses to data are issued only after all of the local processor's previous acquire accesses have completed, b) release accesses are issued only after all previous data accesses have completed, and c) special (synchronization) accesses are processor consistent.

Both weak and release consistency violate the constraints of sequential consistency as shown in figure 4. The difference between the relaxed models and processor consistency

stems from the ordering of writes. Under processor consistency writes by a single processor must be observed by all processors in the order they were issued. Under the weaker models writes that occur between a matched pair of synchronization operations can be observed in any order, so long as they all complete before the second operation of the pair. This relaxation allows a given processor's writes to be pipelined, and their ordering changed. Release consistency improves on weak consistency by allowing acquire accesses to occur immediately (regardless of what data accesses may be outstanding at the moment) and by allowing data accesses to proceed even if there are release accesses outstanding.

The programmer-centric consistency models take the form of a contract between the programmer and the coherence system. The contract defines a notion of *conflicting accesses*, and specifies rules that determine a partial order on the operations in a program. A program obeys the contract if for every pair of conflicting accesses, one access comes before the other in the partial order. A coherence system obeys the contract if it provides the appearance of sequential consistency to programs that obey the contract.

DRF0 says that two accesses conflict if they access the same location, and at least one of them is a write. It defines an order on a pair of accesses if they are issued by the same processor, or if they are synchronization references to the same location (or if there is a transitive chain of such orderings between them). DRF1 extends DRF0 by distinguishing between acquire and release operations. Synchronization reference A precedes synchronization reference B iff A is a release operation, B is an acquire operation, and A and B are *paired* (e.g. B returns a value written by A). PLpc extends DRF1 by further distinguishing *loop* and *non-loop* synchronization accesses. Informally, the final read in a busy-wait loop is ordered with respect to the write that changed the value, but earlier reads in the loop are not ordered with respect to that write, or with respect to previous loop-terminating writes made by the spinning processor.

0.5 Implementation and Performance of Memory Consistency Models

The implementation of memory consistency models presents architects with several design decisions:

- Should coherence be implemented in hardware, software, or some combination of the two?
- What should be the size of coherence blocks?
- Should copies of data be updated or invalidated on a write?
- Should the updates/invalidates be performed in an eager or lazy fashion?

While all combinations of answers to the above question are possible the choice between hardware and software has a strong influence on the remaining questions. For this reason we will discuss hardware and software implementation of consistency models separately.

0.5.1 Hardware Implementations

The coherence unit of choice under a hardware implementation is usually a cache line (cache line sizes currently vary between about 32 and 256 bytes). Existing hardware systems use an *eager* coherence protocol, performing updates or invalidations as soon as inconsistencies arise. A *lazy* protocol would delay updates or invalidations until an inconsistency might be detected by the user program. It would require memory and logic that have generally been considered too expensive to implement in hardware. The choice between updating and invalidating has generally been made in favor of invalidation, due to its lesser amount of communication (see section 0.2), but recent studies have shown that hybrid protocols using an update mechanism on some program data may provide performance benefits [72].

Hardware coherence protocols have been implemented with a wide range of consistency models. The differences between the models are mainly seen in the design and use of write

buffers.[5] Specifically, the consistency model dictates answers to the following questions: a) can reads bypass writes? b) can writes be pipelined? and c) can writes that belong to the same cache line be merged? Sequential consistency does not allow any of these optimizations, since the concept of sequential execution requires that each operation complete before the next one can be started.

All of the relaxed consistency models described in section 0.4 allow reads to bypass writes. That is, they allow a read instruction to complete (e.g. from the cache) before one or more previous write instruction(s) have completed. The non-uniform models relax the constraints even further by allowing writes between synchronization accesses to be pipelined and merged. The models that distinguish between different categories of synchronization accesses allow some of those accesses to be pipelined with neighboring data accesses.

Pipelining of writes means more than simply requesting another cache line before a previous one has arrived from memory:[6] it also means temporally overlapping the coherence protocol messages required to obtain writable copies of the lines. In an invalidate-based protocol, a processor performing a write must obtain an exclusive copy of the relevant line. If the protocol is sequentially or processor consistent, then acknowledgments must be received from all former holders of copies of the line before invalidations are sent out for lines written by any subsequent instructions: otherwise different processors might see the invalidations, and hence the writes, in different orders. Non-uniform relaxed models allow a processor to retire a write from its write buffer once all the invalidation messages have been sent, and to issue a subsequent write before acknowledgments of the earlier invalidations have been received. Table 3 summarizes the differencies in behavior of the four most common consistency models encountered in the literature. A more detailed version of the table

[5] A write buffer allows a processor to continue executing immediately after a write miss. It holds the written data until the appropriate line can be retrieved. A several-entry-deep write buffer can hide the performance impact of a burst of write instructions.

[6] Note that cache lines must generally be fetched from memory on a write miss, because only one part of the line is being written at the moment, and the entire line will be written back (under a write-back policy) when it is again evicted.

Action	SC	PC	WC	RC
1. Read	Processor stalls for pending writes to perform (or, in very aggressive implementations, to gain ownership). Processor issues read and stalls for read to perform.	Processor issues read and stalls for read to perform. Note: Reads are allowed to bypass pending writes.	Processor issues read and stalls for read to perform. Notes: Read are allowed to bypass pending writes. For interaction with pending releases, see point 4.	Processor issues read and stalls for read to perform. Note: Reads are allowed to bypass pending writes and releases.
2. Write	Processor sends write to write buffer (stalls if write buffer is full) Note: Write buffer retires a write only after the write is performed, or in very aggressive implementations, when ownership is gained		Processor sends write to write buffer (stalls if write buffer is full) Notes: Write buffer does not require ownership to be gained before retiring a write. For interactions with acquires/releases, see points 3,4.	
3. Acquire	Treated as Read	Treated as Read	Processor stalls for pending writes and releases to perform. Processor issues acquire and stalls for acquire to perform.	Processor issues acquire and stalls for acquire to perform. Note: Processor does not need to stall for pending writes and releases.
4. Release	Treated as Write	Treated as Write	Processor sends release to write buffer (stalls if write buffer is full) Notes: Write buffer can not retire the release until all previous writes are performed. Write buffer stalls for release to perform. Processor stalls at next read after release for release to perform.	Processor sends release to write buffer (stalls if write buffer full). Note: Write buffer can not Write buffer can not retire the release until all previous writes and releases are performed.

Table 3: Implementation issues of consistency models (Reproduced with permission from [34])

along with a complete description of the behavior of hardware implementations of different memory consistency models can be found in a paper by Gharachorloo et al. [34].

Allowing reads to bypass writes is crucial for achieving good performance, and all relaxed memory models (processor consistency, weak consistency, release consistency, etc.) provide significant performance benefits over sequential consistency. The pipelining of writes and the ability to overlap some of the synchronization operations with data accesses is of secondary importance and the benefits depend on specific program patterns that are not very common in practice. The ability to merge writes provides performance improvements mainly for update protocols. It allows one to collect writes to the same line and then send them in one operation. Such an approach reduces interconnect traffic and therefore the potential latency of other operations.

The importance of letting reads bypass writes stems from the fact that current processors stall on a read miss. Any latency added to reads is completely lost to the processors. Bypassing becomes less important if processors are able to perform non-blocking loads (i.e. prefetch operations), or to switch to a different context. In this case pipelining and merging of writes become significantly more important and the hybrid consistency models provide significant performance advantages over both sequential and processor consistency.

0.5.2 Software Implementations

Coherence can be implemented in software in two very different ways: via compiler code generation, as in HPF (see section 0.3), or via runtime or OS-level observation of program behavior. We focus here on the *behavior-driven* approach, which is analogous to the functioning of hardware cache coherence.

Most behavior-driven software coherence schemes are the intellectual descendants of Li and Hudak's Ivy [53]. Ivy was designed to run on a network of workstations, and uses conventional virtual memory protection bits to implement a simple single-writer sequentially-consistent coherence scheme for pages. "Uncached" pages are marked invalid in the (per processor) TLB and page table. On a page fault, the OS interrupt handler (or user-level

signal handler) uses message-passing to obtain a copy of the page and place it in local memory, possibly invalidating other copies, and modifying the page table to make the new copy accessible.

Most systems since Ivy have abandoned sequential consistency and exclusive writers, but continue to implement consistency at the level of pages, using VM hardware. These systems are generally referred to as *DSM* (distributed shared memory) or *SVM* (shared virtual memory) emulations. They have been implemented both on networks of workstations and on more tightly-coupled NORMA machines. Recent work suggests that it may be feasible to implement finer-grain software coherence, by exploiting special hardware features (e.g. error-correcting codes), or by automatically modifying programs to perform in-line coherence checks prior to accessing memory [70, 74].

While an eager hardware protocol can run in parallel with the execution of the application, a software protocol must compete with the application for processor cycles. An eager software protocol therefore incurs as much overhead per transaction as a lazy protocol, and may introduce a larger number of transactions[45]. As a result, most recent DSM systems use lazy protocols. The choice of update versus invalidate is less clear-cut in software than it is in hardware protocols: the large (page sized) coherence blocks imply that reacquiring an invalidated block can be very expensive, while updating it a small piece at a time can be comparatively cheap.

Page sized coherence blocks also tend to result in a large amount of false sharing. To minimize the performance impact, recent software systems (e.g. Munin [19]) permit a page to be written concurrently by multiple processors, so long as none of them executes a synchronization operation. The assumption is that the processors would use explicit synchronization to protect any truly-shared data, so any non-synchronized accesses to the same page must constitute false sharing. At synchronization points the protocol must determine the set of changes made by each processor, and must combine these changes to obtain a new consistent copy of the page. This process usually involves a word-by-word comparison of the modified copies of the page with respect to an unmodified, shadow copy.

The existence of a shared physical address space on NUMA machines permits several optimizations in software coherence systems [64]. Simple directory operations can be performed via (uncached) remote reference, rather than by sending a message. Pages can also be mapped remotely, so that cache fills bring lines into the local node on demand, eliminating the need for a copy in local memory. If the caches use write-through, or write-back of only dirty words, then remote mapping of a single main-memory copy eliminates the need to merge inconsistent copies. It is also possible to map pages remotely and uncached, so that each individual reference traverses the interconnection network. This option was heavily used in so-called *NUMA memory management* for older CISC-based NUMA machines without caches [13, 24, 49], and may still be desirable in unusual cases for pages with very poor processor locality.

The main performance benefits of relaxed consistency models over sequential consistency, when implemented in software, stem from the reduction in coherence transactions due to the removal of the single writer restriction. Processors perform protocol actions only at synchronization points and thus spend more time doing useful work and less time processing protocol operations. Reduced coherence traffic can in turn help alleviate memory and network congestion, thereby speeding up other memory operations. The choice between software and hardware implementation of the different consistency models is a topic of active research. Recent work suggests that software coherence with lazy relaxed consistency can be competitive with the fastest hardware alternatives (eager relaxed consistency). Hardware architects also seem to be concluding that the flexibility available in software can be a significant performance advantage: some designs are beginning to incorporate programmable protocol engines [48, 66].

0.6 Conclusions and Trends

The way we think of memory has an impact at several layers of system design, starting with hardware and spanning architecture, choice of programming model, and choice and

implementation of memory consistency model. Three important concepts can help us build cheap, efficient and convenient memory systems:

1. Referential transparency. Shared memory is closer to the sequential world to which most programmers are accustomed. Message passing can make an already difficult task (parallel programming) even harder. Message passing provides performance advantages in some situations, however, and some newer machines and systems support both classes of programming model [12, 46, 48].

2. The principle of locality. The importance of locality is evident throughout the memory hierarchy, with page mode DRAMs at the hardware level, hierarchical memory systems at the memory architecture level, and NUMA programming models at the user level. Taking advantage of locality can help us build cheap memory systems, and having locality principles in mind when programming can help us get good performance out of the systems we build.

3. Sequentially consistent ordering of events. When imposed on arbitrary parallel programs, sequential consistency precludes a large number of hardware/software optimizations. Weaker models allow programs to run faster but make programming harder. A promising way to deal with the complexity is to adopt programmer-centric consistency models, which guarantee sequential consistency when certain synchronization rules are obeyed.

Effective memory system design and usage is probably the most active research topic in multiprocessors today. Several trends can be discerned. Deeper memory hierarchies are likely, with two-level on-chip caches and the possibility of a third level off chip. Locality will become even more important as faster superscalar processors increase the cost of communication relative to computation. Current compiler research focuses on techniques that can improve the temporal and spatial locality of programs by restructuring the code that accesses data. The advent of programmable cache controllers may allow us to customize

protocols and models to the needs of the program at hand, rather than imposing a system-wide decision. Finally, as the relative cost of memory systems and processors tilts farther toward expensive memory, it may make sense to change our processor-centric view of the world. Processor utilization may become a secondary performance metric and memory (cache) utilization may start to dominate performance studies. If program execution time depends primarily on memory response time, then peak performance will be achieved when the memory system is fully utilized regardless of the processor utilization.

Bibliography

[1] N. M. Aboulenein, J. R. Goodman, S. Gjessing, and P. J. Woest. Hardware Support for Synchronization in the Scalable Coherent Interface (SCI). In *Proceedings of the Eighth International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[2] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[3] A. Agarwal and others. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1992.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *1990 ACM International Conference on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, June 1990. In *ACM SIGARCH Computer Architecture News 18*:3.

[5] G. A. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. In *1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[6] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[7] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[8] G. R. Andrews and F. B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–44, March 1983.

[9] G. R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, CA, 1991.

[10] J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[11] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[12] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[13] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, Santa Clara, CA, April 1991.

[14] W. J. Bolosky and M. L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proceedings of the Fourth USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–71, September 1993. Also available as MSR-TR-93-1, Microsoft Research Laboratory, September 1993.

[15] L. Bomans, D. Roose, and R. Hempel. The Argonne/GMD Macros in FORTRAN for Portable Parallel Programming and Their Implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, September 1990.

[16] D. Bondurand. Enhanced Dynamic RAM. *IEEE Spectrum*, 29(10):49, October 1992.

[17] D. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, August 1992. Originally presented at *Supercomputing '91*.

[18] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989. Relevant correspondence appears in Volume 32, Number 10.

[19] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, Pacific Grove, CA, October 1991.

[20] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[21] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *Computer*, 23(6):49–58, June 1990.

[22] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, Santa Clara, CA, April 1991.

[23] D. Y. Cheng. A Survey of Parallel Programming Languages and Tools. RND-93-005, NASA Ames Research Center, Moffet Field, CA, March 1993.

[24] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the*

Twelfth ACM Symposium on Operating Systems Principles, pages 32–44, Litchfield Park, AZ, December 1989.

[25] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings Supercomputing '93*, Portland, OR, November 1993.

[26] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment. ORNL/TM-12231, October 1992.

[27] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 88–97, San Diego, CA, May 1993.

[28] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the Thirteenth International Symposium on Computer Architecture*, pages 434–442.

[29] S. J. Eggers and R. H. Katz. Evaluation of the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the Sixteenth International Symposium on Computer Architecture*, pages 2–15, May 1989.

[30] S. J. Eggers and R. H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, MA, April 1989.

[31] M. Farmwald and D. Mooring. A Fast Path to One Memory. *IEEE Spectrum*, 29(10):50–51, October 1992.

[32] D. Gannon and J. K. Lee. Object Oriented Parallelism: pC++ Ideas and Experiments. In *Proceedings of the Japan Society for Parallel Processing*, pages 13–23, 1991.

[33] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 15–26, Seattle, WA, May 1990.

[34] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, Santa Clara, CA, April 1991.

[35] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

[36] J. R. Goodman. Using Cache Memory to Reduce Processor/Memory Traffic. In *Proceedings of the Tenth International Symposium on Computer Architecture*, pages 124–131, June 1983.

[37] J. R. Goodman. Cache consistency and sequential consistency. Computer Sciences Technical Report #1006, University of Wisconsin — Madison, February 1991.

[38] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the Eighteenth International Symposium on Computer Architecture*, pages 254–263, Toronto, Canada, May 1991.

[39] A. Gupta and W. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[40] E. Hagersten, A. Landin, and S. Haridi. DDM — A Cache-Only Memory Architecture. *Computer*, 25(9):44–54, September 1992.

[41] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[42] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992. Originally presented at *Supercomputing '91*.

[43] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. Scalable Coherent Interface. *Computer*, 23(6):74–77, June 1990.

[44] R. Katz, S. Eggers, D. A. Wood, C. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the Twelfth International Symposium on Computer Architecture*, pages 276–283, Boston, MA, June 1985.

[45] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[46] P. T. Koch and R. Fowler. Integrating Message-Passing with Lazy Release Consistent Distributed Shared Memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[47] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh. Parallel Supercomputing Today and the Cedar Approach. *Science*, 231:967–974, February 1986.

[48] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The FLASH Multiprocessor. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[49] R. P. LaRowe Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[50] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[51] T. J. LeBlanc and E. P. Markatos. Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 254–263, Dallas, TX, December 1992.

[52] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[53] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989. Originally presented at the *Fifth ACM Symposium on Principles of Distributed Computing*, August 1986.

[54] W. Li and K. Pingali. Access Normalization: Loop Restructuring for NUMA Compilers. *ACM Transactions on Computer Systems*, 11(4):353–375, November 1993. Earlier version presented at the *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[55] D. J. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[56] D. B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, February 1993.

[57] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994. Earlier version presented at *Supercomputing '92*.

[58] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[59] D. Mosberger. Memory Consistency Models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, January 1993. Relevant correspondence appears in Volume 27,

Number 3; revised version available Technical Report 92/11, Department of Computer Science, University of Arizona, 1993.

[60] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[61] T. Ngo and L. Snyder. On the Influence of Programming Models on Shared Memory Computer Performance. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[62] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, August 1991.

[63] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the Eleventh International Symposium on Computer Architecture*, pages 348–354, May 1984.

[64] K. Petersen and K. Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. In *Proceedings of the Seventh International Parallel Processing Symposium*, Newport Beach, CA, April 1993.

[65] C. D. Polychronopoulos and others. Parafrase-2: A Multilingual Compiler for Optimizing, Partitioning, and Scheduling Ordinary Programs. In *Proceedings of the 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[66] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level Shared-Memory. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 325–336, Chicago, IL, April 1994.

[67] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson. The Monarch Parallel Processor Hardware Design. *Computer*, 23(4):18–30, April 1990.

[68] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High-Level Machine-Independent Language for Parallel Programming. *Computer*, 26(6):28–38, June 1993.

[69] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[70] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.

[71] P. Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12–24, June 1990.

[72] J. E. Veenstra and R. J. Fowler. A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 149–160, Boston, MA, October 1992.

[73] Z. G. Vranesic, M. Stumm, D. M. Lewis, and R. White. Hector: A Hierarchically Structured Shared-Memory Multiprocessor. *Computer*, 24(1):72–79, January 1991.

[74] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for Distributed Shared Memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.