# Compiling for Shared-Memory and Message-Passing Computer[1]

November 12, 1993

**James R. Larus**[2]
Computer Sciences Department
University of Wisconsin
1210 West Dayton St.
Madison, WI 53706 USA
larus@cs.wisc.edu

**Abstract**

Many parallel languages presume a shared address space in which any portion of a computation can access any datum. Some parallel computers directly support this abstraction with hardware shared memory. Other computers provide distinct (per-processor) address spaces and communication mechanisms on which software can construct a shared address space. Since programmers have difficulty explicitly managing address spaces, there is considerable interest in compiler support for shared address spaces on the widely available message-passing computers.

At first glance, it might appear that hardware-implemented shared memory is unquestionably a better base on which to implement a language. This paper argues, however, that compiler-implemented shared memory, despite its shortcomings, has the potential to exploit more effectively the resources in a parallel computer. Hardware designers need to find mechanisms to combine the advantages of both approaches in a single system.

Categories and Subject Descriptors: B.3.2: [**Memory Structures**]: Design Styles—*shared memory*; C.1.2: [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream (MIMD), parallel processors*; D.1.3: [**Programming Techniques**]: Concurrent programming—*parallel programming*; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization, run-time environments.*

General Terms: Languages, performance.

Additional Key Words and Phrases: Compilers, parallel programming languages, shared-memory multiprocessors, message-passing multiprocessors, memory systems, cache coherence, and directory protocols.

## 1.0 Introduction

This paper examines the implications—for compiler writers and hardware designers—of implementing a program's shared address space on computers with and without shared-memory hardware. The semantics of most programming languages presume a *shared address space* in which any portion of a computation can reference any datum. Many parallel computers, how-

1. To appear in ACM Letters on Programming Languages and Systems (LOPLAS).

2. This work is supported by NSF NYI Award CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, and a Univ. of Wisconsin Graduate School Grant.

ever, have physically distributed memory divided into distinct, per-processor address spaces. On these *message-passing computers*, a programmer or a compiler and run-time system can construct a shared address space by sending data between producers and consumers with explicit (program controlled) messages. By contrast, the hardware in a shared-memory computer implements a shared address space by fetching and updating remote data in response to memory reads and writes.

These two alternatives, which we call *compiler-implemented shared memory (CISM)* and *hardware-implemented shared memory (HISM)* respectively, differ in many important respects. HISM exploits hardware's speed to offer fine-grain, relatively low-latency access to any memory location without programmer or compiler assistance. On the other hand, hardware complexity limits systems design, and consequently memory systems provide few functional or performance-enhancing primitives and little flexibility. By contrast, CISM software completely controls communication, which enables static programs to use a computer's network and memories efficiently. Programs that a compiler cannot analyze or that have statically unpredictable (dynamic) behavior, can incur a large penalty because of the overhead of managing a shared address space with run-time software. From another perspective, shared-memory hardware provides fast, high-level semantic operations for a small set of policies. Message passing, on the other hand, provides hardware-level operations that enable a compiler to build application-specific policies.

It is important to keep in mind that many HISM systems are built on a message-passing hardware base. The two types of machines are fundamentally similar. The limitations of existing systems reflect implementation shortcomings rather than intrinsic limits on shared-memory hardware. These limitations are becoming less severe. For example, directories [4,27] eliminate the need to broadcast and so free shared memory from the processor limitations of bus-based Multis [6]. Newer directory systems, such as $Dir_1SW$ [18,38], reduce the complexity of directories and provide rudimentary facilities to improve performance by enabling software to inform the memory system of upcoming program behavior. MIT Alewife goes further and exposes the underlying message-passing mechanisms [24].

At first glance, it appears that a compiler writer would unquestionably prefer HISM. Compilers for shared-memory computers are simpler and provide a more uniform level of performance for a wide range of programs. CISM requires complex compilers and complicates applications programs by requiring programmers to assume the difficult task of partitioning data among processors' memories. This paper argues, however, that CISM has several advantages that enables it—in the best case—to use a parallel computer's memory and communication mechanism more effectively than HISM. The challenge is to identify each approach's advantages and either incorporate comparable performance-enhancing mechanisms into shared memory systems or extend message passing to work well for programs that cannot be statically analyzed. This paper takes a first step in this direction by trying to identify these advantages.

The next section of the paper briefly reviews how compilers and shared-memory hardware implement a shared address space. Section 2.0 investigates the differences between these two implementations, with an eye towards identifying CISM's advantages.

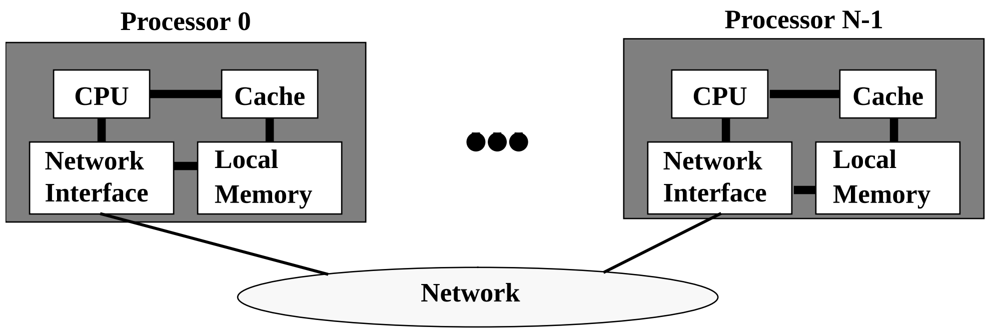

**Figure 1:** Message-passing computer. A typical message-passing machine consists of a collection of processors that communicate over a point-to-point network. Each processor has a local memory that cannot be directly accessed from other processors.

## 2.0 Shared Address Space Implementation

This section provides background by briefly reviewing how compilers and hardware implement a shared address space.

### 2.1 Compiler-Implemented Shared Memory

Compilers for various languages, mainly Fortran dialects, have constructed shared address spaces on message-passing computers [7,14,21,31,39]. Because this is an active research area, standard terminology has not yet emerged and the strengths and limitations of techniques are not yet widely known. The description below summarizes the approach used by many compilers.

The hardware base for these compilers is a message-passing computer (Figure 1). Each processor is connected to a local memory and network interface. Processors communicate by sending messages across a point-to-point network. Messages typically have a relatively high latency (100–1000's of instructions). The network, however, has high bandwidth, so large, infrequent messages are preferable to small, frequent messages. However, one system—the Thinking Machines CM-5 [20]—shifts the balance by reducing latency by an order of magnitude, at the cost of peak bandwidth.

The common approach to compiling for these machines is to distribute data (arrays) among processors' local memories according to programmer-supplied directives, and to partition a computation among processors in a way that reduces communication [8]. High-Performance Fortran (HPF), for example, provides a programmer with `template`, `align`, and `distribute` directives to distribute arrays across processors [29].

Although a programmer usually partitions data, the compiler partitions the computation among processors. A common approach is *owner computes*, in which the processor holding a location (its *home processor*) computes and assigns the location's new values. At each assignment statement, processors send their values to other processors, which receive the values necessary to compute the expression on the right-hand side of the assignment and performs the computation. In the best case, the compiler uses distribution directives to statically determine the processor that owns an value, the processors that require the value, and arranges for the home processor to directly send this value to the other processors. We call this approach (with the optimizations discussed below) *best-case CISM*.

The compile-time and run-time bookkeeping in CISM arises because of a fundamental characteristic of message-passing: the sender and receiver of a message must know each other's identity, at least to the extent of agreeing on a communication channel. Shared memory differs since memory locations have universally-known and accessible names. In effect, memory locations in HISM add a level of indirection that hold a value between its production and consumption. Consequently, a producer and consumer need not be aware of each other's identity.

Data in CISM is distributed by its producer—rather than demanded by its consumers, as in HISM—but is not delivered until the consumer is ready. A synchronous message send requires the processors to rendezvous before transferring data, but eliminates buffering. An asynchronous transfer permits the producer to continue, but may require buffering a message if the consumer is not ready for it. In both cases, a receive operation blocks until data is available. Message passing enables a producer to ship data as soon as it is produced, even if the receiver is not ready, which can help hide message latency. HISM, on the other hand, requires a rendezvous between the producer and consumer to ensure that the producer does not update a memory location before the consumer finishes with its old value and that the consumer does not read a value before it is produced. From another perspective, message passing effectively combines producer-consumer synchronization with communication.

The high communication latency of most message-passing machines prevents a fine-grained approach from being used directly. Compilers must amortize and hide message latency by combining and carefully issuing message sends and receives. M*essage vectorization c*ollects requests or responses for one processor in a single message, which incurs less overhead than separate messages. *Message pipelining* overlaps communication and computation by sending data early in a computation, before another processor attempts to read it.

The implementation and optimizations outlined above presuppose that a compiler can statically determine an array reference's home processor and predict a program's control flow. When the home processor mapping is unknown at compile time, it must be computed during program execution. Run-time techniques are necessary in common and mundane situations, for example doubly-subscripted array references. Two techniques for computing this mapping are run-time resolution and inspector-executor systems.

With *run-time resolution* [34], every processor at reference `A[B[i]]` computes `B[i]`, examines the resulting value *x* to determine if it owns A[*x*], and if so, finds the processor to which to send the value. Run-time resolution is costly because it requires fine-grain message passing, performs an expensive location-to-processor mapping in software, prevents program optimizations such as message vectorization and pipelining, and serializes a program's execution.

Many important problems, such as sparse matrix and unstructured mesh codes, have static communication patterns that are unknown at compile-time. *Inspector-executors* [23,35] exploit the static pattern by making an initial pass over a program's data to find the communication pattern and build an explicit map, which the program subsequently uses to communicate values. The map eliminates much of the run-time overhead to compute a location's owner and where to send values. However, communication optimizations, such as message vectorization and pipelining, are difficult to perform since the compiler is unaware of the communication pattern.

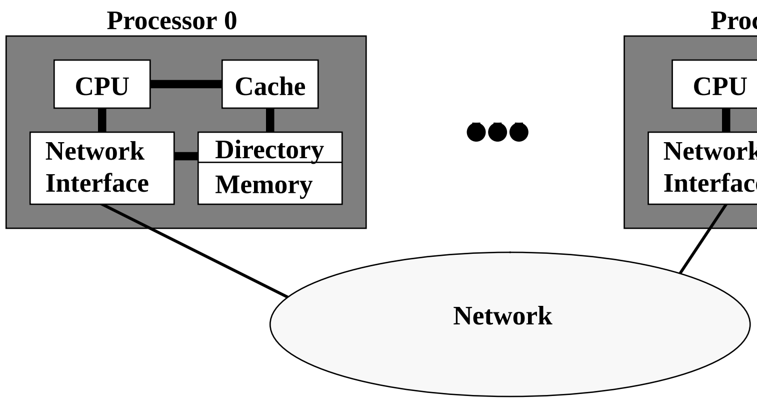

**Figure 2:** Directory-based shared-memory computer. The hardware base of a shared-memory computer is similar to a message-passing computer. The directory hardware helps keep the memory consistent by tracking which processors have cached copies of a memory location, so they can be invalidated when the location's contents change.

Preliminary measurements suggest that widespread use of run-time resolution is impractical. Hall et al. [17] implemented a parallel version of DAXPY in the DGEFA (Gaussian Elimination) routine from the Linpack Benchmarks. On a 32 processor Intel iPSC/860 computer, the run-time resolution version ran 130–180 times slower than the sequential code. By contrast, the best-case CISM version ran up to 8 times faster. Admittedly, these numbers represent a single datapoint from an immature system, but the magnitude of the improvement necessary to come close to breaking even with the sequential code is striking. Inspector-executor overheads are much smaller and the speedups for programs that fit the model are close to linear [23].

## 2.2 Hardware-Implemented Shared Memory

The other approach to constructing a shared address space is to support it with hardware. Machines have implemented shared memory in many different ways. Non-uniform access machines (NUMA), such as the BBN Butterfly [33], partition memory among processors and fetch a remote location at each reference, which results in sharply higher costs for remote accesses. Most shared-memory computers, however, use caches to keep copies of a memory location close to the processors that are actively accessing it. Caches can reduce effective memory access time and communication network bandwidth since interprocessor communication occurs when a block is brought in and flushed from a cache. Cache coherence protocols keep cached copies consistent as processors modify memory locations. Hardware for coherence distinguish classes of parallel computers. Multis [6] are bus-based multiprocessors in which all processors watch memory accesses over a shared bus and modify their caches appropriately. Directory-based computers—such as Stanford DASH [27] and MIT Alewife [3]—eliminate the non-scalable bus by having hardware, and sometimes software, maintain a directory that records which processors have copies of a cache block. A cache-coherence protocol uses the directory to serialize conflicting updates and to invalidate copies at updates. COMA machines, such as the KSR-1 [22], organize all of their memory as a cache. It is unclear how the complex COMA hardware performs or scales [36], so this paper concentrates on the more proven directory technology. This section briefly describes cache-coherent memory systems based on directories.

The primary logical difference between these systems and message-passing computers is that the memory, which is physically distributed in both, is referenced in a single address space in shared-memory computers (see Figure 2). The principal hardware change is the addition of directories [4]. The directory on node *i* maintains information on which nodes have copies of memory locations whose home is node *i*. When node *A* accesses a memory location that is not in its cache, shared-memory hardware uses a simple, static mapping from the location's address to its home node to determine where to request the location's current value. node *A*'s hardware sends a message to this node requesting a copy of the cache block containing the location. Depending on the type of request (read or write) and the state of the cache block, the home node's hardware may need to reclaim outstanding copies of the block by sending invalidate messages to other nodes' caches. The home node then updates its directory and sends the block to node *A*, which puts this block in its cache, where it can be repeatedly and quickly accessed until removed by cache replacement or invalidation.

Directory systems differ primarily on the data transfer granularity and the hardware protocol to maintain directory state. At one extreme is Li's Shared Virtual Memory [28] which uses a processor's virtual memory hardware to detect accesses to non-local locations, transfers pages of memory, and maintains the directory entirely in software. At the other extreme is Stanford DASH [27], which implements a complex directory protocol entirely in hardware and transfers 32-byte cache blocks. Cooperative shared memory [18] is an alternative approach that couples a simple directory protocol ($Dir_1SW$) with a programming model and a collection of performance-improving memory system directives. $Dir_1SW$ implements common directory operations in hardware and provides mechanisms that enable software to handle the less frequent and more complex cases efficiently. A recent study shows that a slightly improved version of $Dir_1SW$ performs comparably with a more complex protocol similar to the one used in DASH [38].

## 3.0 Compiler and Hardware Differences

This section discusses the implications of constructing a shared address space in these different ways. The two implementations have radically different functional requirements and performance characteristics. HISM offers relatively low latency access to any location without requiring a compiler to identify the accessed location in advance. However, shared-memory hardware offers few mechanisms for efficiently transferring large volumes of data, few ways to exploit static program analysis, and limited, unpredictable local data storage. CISM, on the other hand, requires a complex run-time system and extensive program analysis, but, in the best case, offers considerable leeway to hide latency and optimize data transfer and storage.

## 3.1 Logical to Physical Mapping

The two approaches use very different processes to translate a reference to a logical address, such as `A[i]`, into a physical memory reference. The CISM translation process has two alternatives shown in Figure 3. In the best case, when compile-time data-distribution information determines the home processor, the compiler can produce code that directly references the location. The code is similar to a sequential array access, except that it must account for the renaming that results from partitioning an array among the processors. With run-time resolution or inspector-

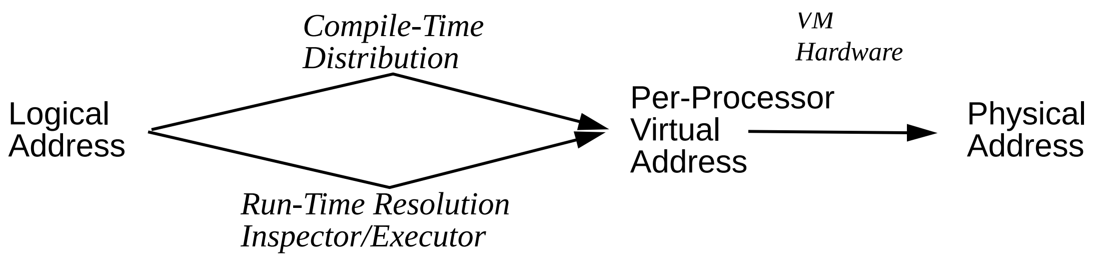

**Figure 3:** Logical to physical address translation in compiler-implemented shared memory (CISM). The mapping from a logical address, such as `A[I]`, to a virtual address is performed, in the best case, at compile time using data distribution information. Otherwise, a technique such as run-time resolution or inspector/executor computes the mapping during program execution.

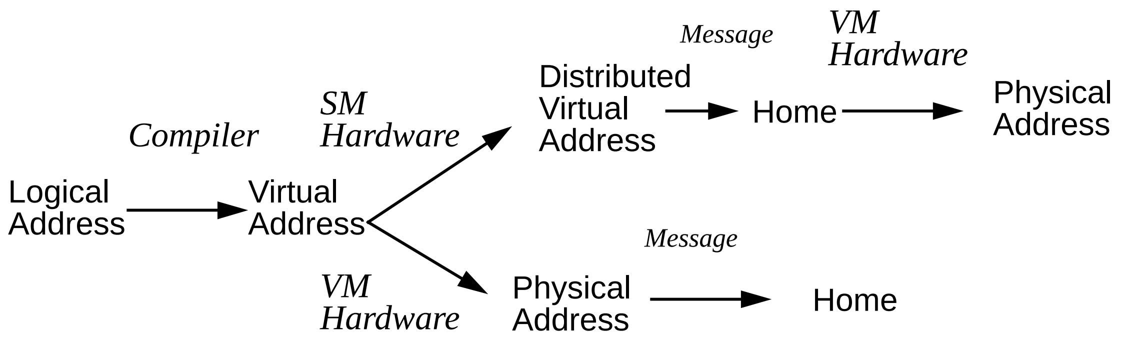

**Figure 4:** Logical to physical address translation processes in hardware-implemented shared memory (HISM). Conventional code produces a virtual address from a logical address. The shared-memory hardware can use either the virtual or physical address to determine the home node.

executors, the compiler produces code for all processors that computes the logical address and maps it to a per-processor virtual address. This mapping also identifies the location's home processor, which is the only one that references the physical address.

Figure 4 shows the translation process for HISM. A compiler generates code that translates an array reference to a memory address in the conventional manner. Hardware uses one of two approaches to map this shared-memory virtual address to a physical address on its home processor. In some systems—Dally's J-machine[11] and the Wisconsin Wind Tunnel [18]—shared-memory hardware maps a virtual address to a *distributed virtual address* consisting of a home processor number and virtual address. The shared-memory system sends this address to the home processor, which uses its virtual memory system to produce a physical address (top arrow). The other approach—as exemplified by Stanford DASH [27]—uses each processor's virtual memory hardware to translate a virtual to a physical address. Shared-memory hardware converts this address into a *distributed physical address* consisting of a home processor number and physical address. In both approaches, the hardware mapping is at the relatively coarse granularity of a virtual memory page. A program or compiler can mollify this mapping by using the logical to virtual mapping (i.e., array indexing) to collect related items on a page (see below).

CISM provides software with complete control over the logical to physical translation. This control exacts a high cost when the bookkeeping is done at run time. HISM is less flexible, but its dynamic bookkeeping is more efficient than run-time software. This efficiency is particularly valuable for programs whose behavior cannot be fully analyzed.

## 3.2 Data Distribution

Compilers for the two forms of shared memory take different approaches to data distribution. CISM statically distributes data among the disjoint memories on the basis of programmer-supplied annotations. Compilers for cache-coherent shared-memory machines typically are not concerned with static data distribution. Instead, they concentrate on making good use of caches' dynamic distribution mechanism.

It is seductive, though unrealistic, to expect caches to dynamically rectify shortcomings in a static distribution of a program's data. Caches are relatively small (64KB–2MB) and have limited associativity (1–4 way). Data brought into a cache can displace other useful data and itself be replaced before being reused [13,25].

Compilers for cache-coherent systems typically do not attempt to position data among physically-distributed memories because these systems provide only coarse-grain mechanisms for placing data on a processor node. Taking into account physical distribution could improve cache performance. For example, on DASH, a cache miss to local memory takes 8–29 cycles while a remote miss takes 34–132 cycles [27]. On this machine, data can be distributed by a virtual to physical mapping that exploits the partitioning of physical pages on high-order address bits. However, this approach is limited to page granularity (4KB).

Compilers and programs can partially sidestep the granularity limitation by using the logical to virtual mapping (i.e. array address calculation) to collect logically-related array elements on the same page [36]. This type of non-contiguous allocation is possible for statically blocked arrays, which CISM can also distribute and manipulate efficiently. Non-contiguous allocation complicates array index calculations and can increase the cost of an array access, but standard compiler optimizations can reduce this problem.

Most current message-passing compilers leave the difficult problem of data distribution to application programmers by requiring explicit domain decomposition directives. CISM statically distributes data according to these directives. Data may be explicitly redistributed during a program's execution, but the system provides no feedback mechanism to rectify a bad distribution. As with any abstraction, the performance of distribution directives is heavily dependent on the quality of a translator and a programmer's understanding of the translation process. Unfortunately, current compilers cannot recognize or rectify bad partitioning. Automatic (i.e., compiler) data distribution and alignment is an area of active research [9,16,32]. The generality and applicability of such techniques are still unclear.

## 3.3 Cache Parameters

A major advantage of CISM is that local storage is entirely under compiler control. A shared-memory machine's cache is typically far smaller (e.g., 64KB–2MB) than a processor's local memory (e.g., 32MB–256MB) and the cache is managed by hardware whose behavior is difficult to anticipate because of limited set-associativity. A location brought into a cache by a reference or prefetch can be evicted before being used or reused, because of either a capacity or conflict miss caused by a subsequent reference. A *capacity miss* is a replacement that would not have occurred in an infinite cache and a *conflict miss* is a replacement that would not have occurred in a fully-associative cache of the same size [19]. The latter misses are particularly difficult to anticipate since they depend on the relative addresses of the locations accessed after the original reference. Even assuming it could anticipate both misses, there is little that a compiler can do to avoid them except to change the program's access pattern. Misses adversely affect performance since an evicted location is returned to its home processor and it must be retrieved at high cost (8–132 cycles in DASH). COMA machines, such as the KSR-1, organize local memory as a highly associativity cache, which can alleviate cache constraints.

By contrast, CISM uses local memory to hold data in a position intermediate between a processor's cache and a remote processor's memory. When a location is evicted from a processor's cache because of a conflict or capacity miss, it can be retrieved at much lower cost (8–34 cycles in DASH). Local memory is much larger than a cache and its contents are fully controlled by a compiler, which can use it as a large, fully-associative cache (without coherence).

## 3.4 Bulk Transfer

Many network interconnects have high latency and bandwidth, which heavily favors large messages over sequences of short ones. Shared-memory hardware, however, demands nearly the opposite network characteristics. The impedance mismatch between HISM and networks arises from two factors: transfer granularity and bad buffering. The first difficulty is that memory values are prefetched or referenced at cache-block granularity and are obtained with asynchronous request messages. The request is short and only returns a cache block. Moreover, this pair of messages incurs a round-trip latency. Increasing cache block size increases the data per message, but may also increase coherence traffic because of problems such as false sharing [12]. The second difficulty is that limited cache size and associativity makes it impossible to transfer and buffer large amounts of data. Moving a large quantity of data into a cache can evict useful data.

Best-case CISM alleviates these problems, but new ones arise. CISM does not send request messages, but rather relies on sender-initiated communication, which potentially reduces the number of messages by a factor of two. Perhaps more important, it can reduce communication latency by a factor of two. However, this communication has a hidden cost since asynchronous messages must be buffered if the receiver is not ready. This buffering requires copying and may require memory allocations as well. Compiler analysis that collects together messages exacerbates the problem by increasing message size. Other approaches, such as Active Messages [10], eliminate buffering in the message-passing substrate but require application-level buffering to avoid having values change unexpectedly because of an arriving message.

HISM systems provide few mechanisms for optimizing remote accesses. One of the most common, *non-binding prefetch*, asynchronously moves a remote block into a processor's cache. This mechanism allows computation to overlap communication. However, the transfer unit is generally a single cache block so the overhead to produce and send prefetch requests is large. In addition, prefetched data can displace more immediately useful data or itself be replaced before being used. These considerations limit how early a prefetch should be issued and consequently, how much latency it can hide. In addition, large improvements can result from changing a program's reference pattern to ensure cached data is reused before being replaced [13,25]. Unlike prefetching, these transformations affect semantics and so require extensive compiler analysis to be applied safely. In addition, the profitability of transformations is difficult to predict [37].

## 3.5 Memory Renaming

Another difference between HISM and CISM is that the former uses the memory system to provide multiple instances of a memory location under the same name. In CISM, local instances are true copies that have different names. Each copy can be changed independently, while under HISM, the instances are kept coherent. When an array is partitioned, different processors may have to reference a location with a different base address and indices. This renaming is a major hurdle in writing message-passing programs by hand. For compilers, renaming is more than a bookkeeping chore. For example, overlap analysis in SUPERB [39] and other compilers identifies array sections obtained from other processors, so they can be kept in a contiguous portion of a local array from which they can be uniformly accessed. Currently, SUPERB only performs this analysis for specific, regular access patterns.

Array copying also has advantages. Compilers for shared-memory machines sometimes copy arrays for different reasons. Fortran 90 array operations have a read-before-write semantics that can be difficult to execute effectively on a MIMD processor without copying input or output arrays. In addition, array privatization, a transformation that enhances parallelism, requires processors to copy data from shared to private arrays on shared-memory machines [30]. In addition, the copying of values from remote locations into contiguous local memory can improve uniprocessor cache performance by reducing both conflict and capacity misses [25].

## 3.6 Synchronization

Transmitting values in a message-passing system, because it requires explicit actions by both processors, also transfers synchronization information. A compiler can assume that a value is not consumed until it is produced and sent. On the other hand, in a shared-memory system, a compiler generates code to access a value, but transmission is handled by hardware that executes on behalf of the consumer. To ensure a producer-consumer relationship, a compiler must insert synchronization.

Techniques exist that combine synchronization with shared memory. One alternative is fine-grained synchronization such as empty-full bits, such as those in the Tera processor [5]. This feature complicates the system and memory. In a sense, message passing provides the equivalent of fine-grained synchronization with little overhead when many values can be packed into a message. Another alternative is Queue on Lock Bit (QOLB) [15], which provides a lock for a cache block, maintains a queue of waiting processors, and ensures that the block is in a processor's cache when it acquires the lock. QOLB, though attractive, does not provide adequate mechanisms for multi-block objects or for accessing portions of large objects such as arrays.

## 3.7 Memory Coherence

Hardware shared-memory systems usually ensure a globally consistent view of memory with a cache-coherence protocol. To achieve higher performance, proponents of alternative memory semantics have suggested weakening consistency guarantees so they hold only at defined points in a program's execution [2]. Message-passing systems rely on a compiler to ensure that a program will not find itself in a situation in which two processors believe that different values are current. Ensuring this property does not appear to be a major concern of current compilers, perhaps because current compilation strategies are simple. For example, the common owner-computes strategy allows only one processor to update a location and consequently avoids reconciling multiple updates. In best-case CISM, a compiler also knows which processors have out-of-date copies and can produce code to update these copies. Updating typically requires less message traffic than invalidation, which is used by directory-systems protocols.

Compilers' lack of success in software-controlled caching [1] suggests that the sophistication of message-passing compilation strategies will be limited by static program analysis. Software-controlled caching requires a compiler to identify interprocessor data dependences and insert code to invalidate an out-of-date cache block before it is accessed. The program analysis for this problem is similar to that required for CISM. In both cases, compilers must produce conservative code that preserves a program's semantics for all possible executions. Because static program analysis is necessarily imprecise, compilers make pessimistic assumptions. With software-controlled caching, conservative analysis resulted in more memory-system traffic and lower performance than cache hardware [1].

## 3.8 Cost Modeling

An important difference between the two forms of shared memory is a programmer's or compiler writer's ability to model the cost of a memory reference. With HISM, a remote memory reference appears identical to a local reference, so it is easy to forget that they have vastly different costs (8 cycles best case vs. 132 cycles worst case in DASH). To further complicate matters, on machines with caches, repeated references to the same location can incur vastly different costs as the location moves in and out of the cache. Cache models are complex and typically only capture capacity, not conflict misses [13,26]. By contrast, on a message-passing computer, a remote reference requires explicit communication, whose high cost is obvious and can be modeled.

## 3.9 Worst-Case Performance

A significant issue for both CISM and HISM is how well do they perform in the worst case and how often does the worst case arise? In CISM, the difference between best and worst case is large because of run-time resolution's and similar techniques' low performance. Unfortunately, this technique is necessary whenever a compiler cannot fully analyze a memory reference. Most compiler work to-date has focused on dense matrix codes. Even for these programs, the analysis is complex and must be applied interprocedurally since partial information does not result in a gradual performance degradation but prevents parallelization [17]. Programs that contain indirect references, dynamic data structures, or pointers cannot be accurately analyzed and so will perform poorly.

Worst case HISM behavior arises because of ineffective cache usage, bad data distribution, and false sharing. It is unclear what is the difference between best and worst case behavior, but it is unlikely to approach the factor of 1,000 between best-case and run-time resolution CISM [17].

## 4.0 Conclusion

We can briefly summarize the main advantages of compiler-implemented shared memory over hardware. In the best case, CISM statically distributes data on processors that access it and can use each processor's local memory as a large, software-controlled cache to hold data. A compiler can arrange to efficiently transfer large quantities of data directly from a producer to consumers, without requiring explicit requests for each packet and without being constrained by a hardware coherence protocol. In addition, message passing effectively combines synchronization with communication. Message passing, in general, offers a compiler greater control and the potential of achieving higher performance when the compiler can accurately analyze a program.

In praising compiler-implemented shared address spaces, it is important not to lose sight of hardware's advantages. Hardware enables programmers and compilers to quickly and easily run a program on a parallel computer and it effectively services memory requests in programs whose sharing patterns are not analyzable. Caches provide an efficient, dynamic mechanism to improve locality. These advantages cannot be duplicated effectively by software on a message-passing computer.

These two approaches are, in many ways, complementary. The challenge for shared-memory designers is to provide new primitives and mechanisms that enable compilers to efficiently transfer large quantities of data (for example [24]), make better use of the local memory on each processing node, and integrate communication and synchronization in a shared address space. The challenge for message-passing system designers is to provide mechanisms that enable programs to rapidly respond to dynamic memory accesses and to quickly transfer small quantities of data. The goal in both case can be the same: direct control of low-level hardware mechanisms when static analysis suffices and effective hardware support to fall back on when it does not.

## Acknowledgments

Douglas Berger, David Callahan, Satish Chandra, Trishul Chilimbi, Mary Hall, Mark Hill, Brad Richards, Anne Rogers, Madhusudhan Talluri, and David Wood provided many helpful comments on this paper.

## References

[1] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 298–308, June 1991.

[2] 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.

[3] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, June 1990.

[4] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[5] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, June 1990.

[6] C. Gordon Bell. Multis: A New Class of Multiprocessor Computers. *Science*, 228:462–466, 1985.

[7] Zeik Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, Sanjay Ranka, and Min-You Wu. Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers. Technical Report SCCS-444, Syracuse University, March 1993.

[8] David Callahan and Ken Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[9] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic Array Alignment in Data-Parallel Programs. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 16–28, January 1993.

[10] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Thread Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 164–175, April 1991.

[11] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Nuth, Scott Wills, Paul Carrick, and Greg Flyer. The J-Machine: A Fine-Grain Concurrent Computer. In G. X. Ritter, editor, *Proc. Information Processing 89*. Elsevier North-Holland, Inc., 1989.

[12] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–382, 1988.

[13] Dennis Gannon, William Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[14] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessor Systems.* PhD thesis, Rheinischen Friedrich-Wilhelms-Universität, 1989.

[15] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 64–77, April 1989.

[16] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K.Y. Wang, and M. Burke. Ptran II - A Compiler for High-Performance Fortran. In *4th Workshop on Compilers for Parallel Computers*, page ?, Delff, Netherlands, December 1993.

[17] Mary W. Hall, Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines. In *Proceedings of Supercomputing 92*, pages 522–534, November 1992.

[18] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):?, November 1993. Preliminary version appeared in ASPLOS V, Oct. 1992.

[19] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, C-38(12):1612–1630, December 1989.

[20] W. Daniel Hillis and Lewis W. Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993.

[21] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[22] Kendall Square Research. Kendall Square Research Technical Summary, 1992.

[23] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting Shared Data Structures on Distributed Memory Architectures. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 177–186, March 1990.

[24] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

[25] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 63–74, April 1991.

[26] James R. Larus, Satish Chandra, and David A. Wood. CICO: A Shared-Memory Programming Performance Model. In Jeanne Ferrante and Tony Hey, editors, *Portability and Performance for Parallel Processors*, chapter ?, page ? John Wiley & Sons, Ltd., 1993. To appear.

[27] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[28] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

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

[30] Dror E. Maydan, Sman P. Amarasinghe, and Monica S. Lam. Array Data-Flow Analysis and its Use in Array

Privatization. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 2–15, January 1993.

[31] Keshav Pingali and Anne Rogers. Compiling for Locality. In *Proceedings of the 1990 International Conference on Parallel Processing (Vol. II Software)*, pages II–142–146, August 1990.

[32] J. Ramanujam and P. Sadayappan. Compile-Time Technique for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[33] Randall Rettberg and Robert Thomas. Contention is no Obstacle to Shared-Memory Multiprocessing. *Communications of the ACM*, 29(12):1202–1212, December 1986.

[34] Anne Marie Rogers. Compiling for Locality of Reference. Technical Report TR 91-1195, Department of Computer Science, Cornell University, March 1991. PhD thesis.

[35] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-Time Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[36] Jaswinder Pal Singh, Truman Joe, Anoop Gupta, and John L. Hennessy. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessor. In *Proceedings of Supercomputing 93*, page ?, November 1993.

[37] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[38] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993.

[39] Hans Zima and Barbara Chapman. Compiling for Distributed-Memory Systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.