*To appear in: "ACM Transactions on Computer Systems," November 1993.*
*Reprinted by permission of ACM.*

# Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors*

Mark D. Hill, James R. Larus, Steven K. Reinhardt, David A. Wood
University of Wisconsin–Madison†

November 1, 1993

**Abstract**

We believe the paucity of massively-parallel, shared-memory machines follows from the lack of a shared-memory programming performance model that can inform programmers of the cost of operations (so they can avoid expensive ones) and can tell hardware designers which cases are common (so they can build simple hardware to optimize them). *Cooperative shared memory*, our approach to shared-memory design, addresses this problem.

Our initial implementation of cooperative shared memory uses a simple programming model, called *Check-In/Check-Out (CICO)*, in conjunction with even simpler hardware, called $Dir_1SW$. In CICO, programs bracket uses of shared data with a `check_out` directive marking the expected first use and a `check_in` directive terminating the expected use of the data. A *cooperative prefetch* directive helps hide communication latency. $Dir_1SW$ is a minimal directory protocol that adds little complexity to message-passing hardware, but efficiently supports programs written within the CICO model.

## 1 Introduction

The rapid, continual advance in microprocessor technology—which has led to 64-bit processors with large caches and fast floating-point units—provides an effective base for building massively parallel computers. Until now, message-passing computers have dominated this arena.

*A preliminary version of this paper appeared in the *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V).* The performance measurements in Section 4 were previously reported in [31].

This work is supported in part by NSF PYI Awards CCR-9157366 and MIPS-8957278, NSF Grants CCR-9101035 and MIP-9225097, Univ. of Wisconsin Graduate School Grant, Wisconsin Alumni Research Foundation Fellowship and donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, Thinking Machine Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

†Authors' address: Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706, wwt@cs.wisc.edu

Shared-memory computers are rare and currently lag in both number and speed of processors. Their absence is due, in part, to a widespread belief that neither shared-memory software nor shared-memory hardware is *scalable* [25]. Indeed, many existing shared-memory programs would perform poorly on massively-parallel systems because the programs were written under a naive model that assumes all memory references have equal cost. This assumption is wrong because remote references require communication and run slower than local references [19]. For example, a remote memory reference on Stanford DASH costs at least 100 times more than a local reference [23, 24].

To compound matters, existing shared-memory hardware either does not extend to highly parallel systems or does so only at a large cost in complexity. *Multis* [5] are a successful small-scale shared-memory architecture that cannot scale because of limits on bus capacity and bandwidth. An alternative is directory-based cache-coherence protocols [2, 17, 23]. These protocols are complex because hardware must correctly handle many transient states and race conditions. Although this complexity can be managed (as, for example, in the Stanford DASH multiprocessor [23]), architects must expend considerable effort designing, building, and testing complex hardware rather than improving the performance of simpler hardware.

Nevertheless, shared memory offers many advantages, including a uniform address space and referential transparency. The uniform name space permits construction of distributed data structures, which facilitates fine-grained sharing and frees programmers and compilers from per-node resource limits. Referential transparency ensures object names (i.e., addresses) and access primitives are identical for both local and remote objects. Uniform semantics simplify programming, compilation, and load balancing, because the same code runs on local and remote processors.

The solution to this dilemma is not to discard shared memory, but rather to provide a framework for reasoning about locality and hardware that supports locality-exploiting programs. In our view, the key to effective, scalable, shared-memory parallel computers is to address software and hardware issues together. Our approach to building shared-memory software and hardware is called *cooperative shared memory*. It has three components:

- A shared-memory programming model that provides programmers with a realistic picture of which operations are expensive and guides them in improving program performance with specific performance primitives.

- Performance primitives that do not change program semantics, so programmers and compilers can aggressively insert them to improve the expected case, instead of conservatively seeking to avoid introducing errors.

- Hardware designed, with the programming model in mind, to execute common cases quickly and exploit the information from performance primitives.

Underlying this approach is our *parallel programming model*, which is a combination of a *semantic model* (shared memory) and a *performance model*. The component that has been absent until now is the performance model, which aids both programmers and computer architects by providing insight into programs' behavior and ways of improving it. The performance model provides a framework for identifying and reasoning about the communication induced by a shared-memory system. Without this understanding, discerning—let alone optimizing—common cases is impossible.

Our initial implementation of cooperative shared memory uses a simple programming performance model, called *Check-In/Check-Out (CICO)*, and even simpler hardware called $Dir_1SW$. CICO provides a metric by which programmers can understand and explore alternative designs on any cache-coherent parallel computer. In the CICO model (Section 2), programs bracket uses of shared data with `check_out` annotations that indicate whether a process expects to use a datum and `check_in` annotations that terminate an expected use. CICO's new approach encourages programmers to identify intervals in which data is used repeatedly, rather than focusing on isolated uses, and to explicitly acknowledge when data can be removed from local buffers. An additional *cooperative prefetch* annotation allows a program to anticipate an upcoming `check_out` and hide communication latency by arranging for data forwarding.

$Dir_1SW$ (Section 3) is a minimal directory protocol that requires a small amount of hardware (several state bits and a single pointer/counter field per block), but efficiently supports programs written under the CICO model. The pointer/counter either identifies a single writer or counts readers. Simple hardware entirely handles programs conforming to the CICO model by updating the pointer/counter and forwarding data to a requesting processor. No cases require multiple messages (beyond a single request-response pair) or transient states. The CICO annotations can be passed to $Dir_1SW$ hardware as memory system directives to improve performance. Programs not conforming to the CICO model or using CICO directives run correctly, but trap to system software that performs more complex operations (in a manner similar to MIT Alewife [8]).

Measurements of eight programs running on a 32-processor system show that $Dir_1SW$'s performance is comparable to a more complex protocol ($Dir_{32}NB$) and that the CICO directives improve performance (Section 4). Finally, Section 5 discusses related work.

## 2 CICO Programming Performance Model

The performance component of our programming model is named *Check-In/Check-Out (CICO)*. It serves two roles. First it provides *annotations*, which can be used as a framework to reason about the communication caused by a cache-coherence protocol in any shared-memory computer. In addition, a program can execute these annotations as *memory-system directives* to pass information to the hardware about upcoming program behavior as a way of improving a cache-coherence protocol's performance. This section describes CICO and illustrates how it identifies communication. The next section (Section 3) shows that hardware can also exploit CICO directives to further speed execution.

### 2.1 CICO Annotation

CICO begins with programmer-supplied annotations that elucidate a program's memory references. These annotations describe a program behavior and do not affect its semantics, even if misapplied. CICO uses three sets of annotations to delimit a portion of a program in which a memory location is kept in a processor's cache. The first annotations mark the beginning of an interval in which a processor expects to use a block:

| | |
|---|---|
| `check_out_X` | Expect exclusive access to block |
| `check_out_S` | Expect shared access to block |

`check_out_X` asserts that the processor performing the `check_out` expects to be the only processor accessing the block until it is checked-in. `check_out_S` asserts that the processor is willing to share (read) access to the block. As an optional directive, the `check_out` annotations fetch a copy of the block into the processor's cache, as if the processor directly referenced the block.

The next annotation, `check_in`, marks the end of an interval in which a processor uses a block. An interval ends either because of cache placement or because another processor accesses the checked-out block.

| | |
|---|---|
| `check_in` | Relinquish a block |

As an optional directive, the `check_in` annotation flushes the block from the processor's cache, as if the processor had replaced it upon a cache miss.

Communication cannot be eliminated from parallel programs. An important step to reduce the impact of communication is to overlap it with computation by prefetching data:

| | |
|---|---|
| `prefetch_X` | Expect exclusive access to block in near future |
| `prefetch_S` | Expect shared access to block in near future |

`prefetch_X` (`prefetch_S`) asserts that a processor performing a prefetch is likely to the block for exclusive (shared) access in the near future. As an optional directive, this annotation brings the block into its cache while the processor continues the computation.

`check_in`'s in the CICO model permit a concise description of a producer-consumer relationship. On any machine, prefetches execute asynchronously and can be satisfied at any time. In the CICO model, a *cooperative prefetch* is satisfied when the prefetched block is checked-in. If the prefetch is issued when the block is checked-out, the response is delayed until the block is checked-in. Cooperative prefetch couples a producer and consumer by forwarding fully computed data. The rendezvous is blind: neither the prefetching processor nor the processor checking-in the block know each other's identity. This form of prefetch abstracts away from machine-specific timing requirements and, as shown in Section 3, has an efficient implementation.

## 2.2 CICO Performance Model

After annotating a program with CICO annotations, a programmer can use the *CICO cost model* described in this section to compute the cost of shared-memory communication. This model attributes a communication cost to CICO annotations. By analyzing a program to determine how many times an annotation executes, a programmer can determine the cumulative communication cost of the annotation. If the annotation accurately models the cache's behavior, the cost attributed to the annotation equals the communication cost of the memory references that the annotation summarizes.

In the CICO cost model, the communication cost of CICO annotations is modeled with the aid of an automaton with three states: *idle*, *shared*, or *exclusive* (see Figure 1). Each block has its own automaton. Initially, every block is *idle*, which means that no cache has a copy. Transitions between states occur at CICO annotations. Edges in the automaton are labeled with the process and annotation that caused them. For example, if a block is *idle*, a `check_out_X` changes the block's state to *exclusive*. The processor causing a transition incurs the communication cost associated with an arc.

Communication costs can be modeled in three ways. The first uses values from an actual machine. The advantage of this approach is that the costs accurately model at least one machine. However, in many cases, these values are too machine- and configuration-specific. A more general approach is to use values that asymptotically represent the bandwidth or latency for a large class of machines. Operations that execute asynchronously, such as prefetches or `check_in`'s, are unit cost. Operations that require a synchronous message exchange, such as `check_out`'s, require time proportional to a round-trip message latency: $O(f(P))$, where $f$ is a function, such as

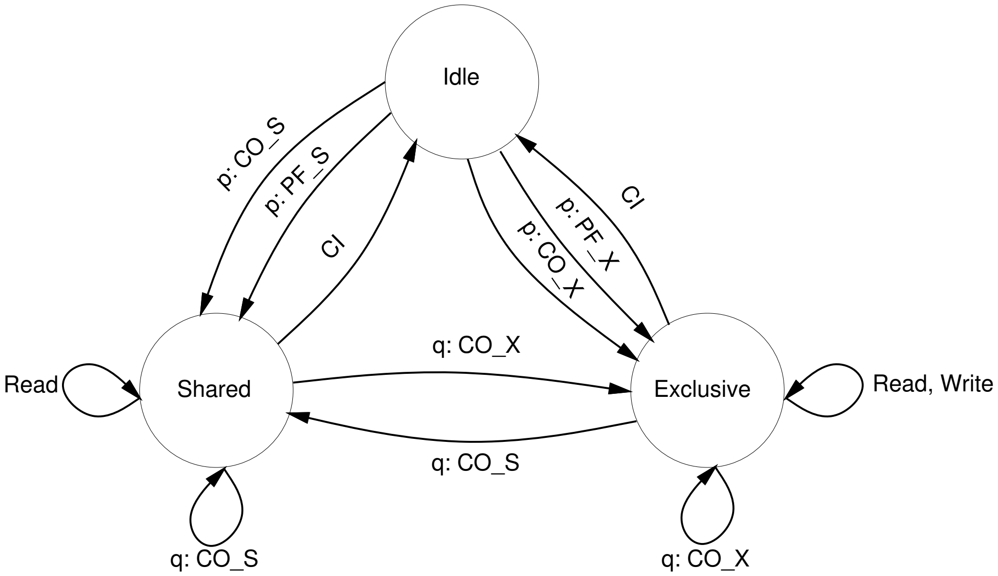


Figure 1: CICO performance model. A block can be in one of three states: *idle*, *shared*, or *exclusive*. Transitions between states occur at CICO annotations and are labeled with the annotation (CO is `check_out`, CI is `check_in`, PF is `prefetch` ) and processor ($p$ or $q$) that caused the transition. In the diagram, processor $p$ obtains a block in the idle state and a distinct processor $q$ operates on the block when it is not idle. A block becomes idle when the last shared or only exclusive copy is checked-in. The processor that causes a transition incurs a communication cost.

$\log_2$, that relates the communication cost to $P$, the number of processors. Finally, the transition *shared* $\rightarrow$ *exclusive* has worst-case cost proportional to $O(P)$ since all extant copies must be invalidated by explicit messages or a broadcast, which requires bandwidth proportional to the number of processors. The final model, which suffices for many purposes, attributes a unit cost to each transition that requires synchronous communication. Table 1 compares the three models.

### 2.3 Synchronization

Synchronization is communication that orders two or more program events in distinct processes. Ordering events with shared memory requires at least two accesses to a location, where one access modifies the location and the other reads it (and perhaps modifies it). The CICO directives described above are unsuitable for synchronization constructs, which require competitive (i.e., unordered and unpredictable) memory access. Rather than extend CICO with directives for unsynchronized accesses, we assume the existence of simple synchronization constructs such as locks and barriers. Section 3.3 shows that Mellor-Crummey and Scott's locks and barriers coexist easily with simple hardware for CICO.

| Initial State | Action | Final State | Concrete Cost | Asymp Cost | Unit Cost |
|---|---|---|---|---|---|
| Idle | `prefetch_X` | Exclusive | 8 | O(1) | 0 |
| | `check_out_X` | | 242 | $O(\log_2 P)$ | 1 |
| | `prefetch_S` | Shared | 8 | O(1) | 0 |
| | `check_out_S` | | 242 | $O(\log_2 P)$ | 1 |
| Exclusive | `check_in` | Idle | 16 | O(1) | 0 |
| | `check_out_X` | Exclusive | 996 | $O(\log_2 P)$ | 1 |
| | `check_out_S` | Shared | 996 | $O(\log_2 P)$ | 1 |
| Shared | `check_in` | Idle — Shared | 8 | O(1) | 0 |
| | `check_out_X` | Exclusive | 1285 | $O(P)$ | 1 |
| | `check_out_S` | Shared | 242 | $O(\log_2 P)$ | 1 |

Table 1: Costs for transitions in the CICO performance model for directory-based shared-memory computers. The concrete costs are best-case values for the $Dir_1SW$ protocol.

### 2.4 Compilers and CICO

Compilers, as well as programmers, can apply the CICO model to both analyze and optimize program behavior. Shared-memory compilers generally have not used accurate memory-access cost models. CICO provides these compilers with an easily-applied metric for evaluating compilation strategies. This metric can be applied either to restructure sequential programs or optimize explicitly parallel programs.

CICO directives are well-suited to compiler analysis since they do not affect a program's semantics. The analysis to employ a directive need not be conservative. Instead, a compiler can optimize the expected case without considering the effects of directives on other possible executions. By contrast, software cache coherence holds a compiler to a much higher standard of ensuring that a program executes correctly, regardless of its dynamic behavior. Because compiler analyses are inherently imprecise, software cache coherence requires a compiler to always err on the conservative side and insert memory system operations to avoid the worst case. This bias results in correct programs that communicate too much.

### 2.5 Discussion

CICO provides shared-memory programmers with a performance model that identifies the communication underlying memory references and accounts for its cost. Message passing also provides programmers with a clear performance model. A common message-passing model attributes a fixed cost to each message independent of its length and destination. When necessary, this model is elaborated to account for an underlying network's topology and transmission cost. Unlike CICO, message-passing models need not detect communication, only account for its cost.

Unfortunately, applying the message-passing model to improve a program's performance is difficult, precisely because communication is explicitly and inextricably linked with functionality. The linkage is so tight that a message-passing program cannot be successfully developed without continual consideration of performance implications because refinements are difficult to incorporate after a program is written. Every communication must be evaluated twice—once in the sender, once in the receiver—to determine if it should be optimized and how the program should change to accomplish this goal. A small change can cause a cascade of modifications as control and data dependences within a process force the reordering of other communications.

CICO is an easier model for a compiler or programmer to apply for the following reasons:

- CICO directives are unnecessary for correct execution. Programmers can incrementally employ them to understand and optimize time-critical routines.
- The directives can be used aggressively to optimize expected program behavior since they do not affect program semantics. The other cases still function correctly.
- The directives do not change a datum's address. A programmer can optimize one routine without changing all routines that interact with it.
- The directives never introduce functional bugs. Using them never breaks correct programs.

## 3 $Dir_1SW$ Hardware

The CICO model can be used by computer architects to simplify hardware and improve its performance. CICO is the abstraction through which programmers and architects communicate, much as instruction sets have been the fundamental abstraction in uniprocessor design. As the analogy suggests, a good abstraction enables programmers to understand and optimize their programs and helps architects design fast, effective computers. RISCs have shown that fast, cost-effective hardware requires hardware designers to identify common cases and cooperate with programmers to find mutually-agreeable models that can be implemented with simple hardware [18]. This combination permits hardware designers to devote their attention to making common cases run fast. Message-passing computers, which are based on a simple model, are built from simple, scalable hardware. Shared-memory multiprocessors, which currently lack a unifying performance model, typically use complex cache-coherence protocols to accommodate

| Message from Processor i | Current State | Next State | Trap? | Data Action | Pointer/ Counter |
|---|---|---|---|---|---|
| Msg_Get_X | Dir_Idle | Dir_X | | send to i | pointer ← i |
| | Dir_X | Dir_X | yes | | |
| | Dir_S | Dir_S | yes | | |
| Msg_Get_S | Dir_Idle | Dir_S | | send to i | counter ← 1 |
| | Dir_S | Dir_S | | send to i | counter += 1 |
| | Dir_X | Dir_X | yes | | |
| Msg_Put | Dir_X | Dir_Idle | | store in | |
| | Dir_S | Dir_S /Dir_Idle | | | counter −= 1 |

Table 2: State Machine for Base $Dir_1SW$ Coherence Protocol.

`Msg_Get_X` and `Msg_Get_S` obtain exclusive and shared copies of a block, respectively. `Msg_Put` returns a copy of a block. Blank entries in action columns indicate no-ops, all traps set the trap bit, all state transitions not listed are hardware errors (e.g., send a `Msg_Put` to `Dir_Idle` block); and all hardware errors trap.

---

all programming styles. By contrast, $Dir_1SW$ relies on the CICO model to describe program behavior and uses simple hardware to effectively support it.

### 3.1 $Dir_1SW$

The hardware base of our cooperative shared-memory machine is the same as a message-passing machine. Each processor node contains a microprocessor, a cache, and a memory module. The nodes are connected with a fast point-to-point network.

Each processor node also has a small additional amount of hardware that implements our directory protocol, $Dir_1SW$, which associates two state bits, a pointer/counter, and a trap bit with each block in memory.[1] In addition, each memory module is addressed in a global address space. In a slightly simplified (base) form, a directory can be in one of three states: `Dir_X`, `Dir_S`, and `Dir_Idle`. State `Dir_X` implies that the directory has given out an exclusive copy of the block to the processor to which the pointer/counter points. State `Dir_S` implies that the directory has given out shared copies to the number of processors counted by the pointer/counter. State `Dir_Idle` implies that the directory has the only valid copy of the block.

Table 2 illustrates state transitions for the base $Dir_1SW$ protocol. `Msg_Get_X` (`Msg_Get_S`, respectively) is a message to the directory requesting an exclusive (shared) copy of a block. `Msg_Put` is a message that relinquishes a copy. Processors send a `Msg_Get_X` (`Msg_Get_S`) message when a local program references a block that is not in the local cache or performs an explicit

[1] We derived the name $Dir_1SW$ by extending the directory protocol taxonomy of Agarwal, et al. [2]. They use $Dir_iB$ and $Dir_iNB$ to stand for directories with $i$ pointers that do or do not use broadcast. The $SW$ in $Dir_1SW$ stands for our *S*oft*W*are trap handlers.

| Message from Processor i | Current State | Next State | Trap? | Data Action | Pointer/ Counter |
|---|---|---|---|---|---|
| Msg_Prefetch_X | Dir_Idle | Dir_X | | send to i | pointer ← i |
| | Dir_X | Dir_X_Pend | | | pointer ← i |
| | Dir_S | Dir_S | | | |
| | Dir_X_Pend | Dir_X_Pend | | | |
| Msg_Get_X | Dir_X_Pend | Dir_X_Pend | yes | | |
| Msg_Get_S | Dir_X_Pend | Dir_X_Pend | yes | | |
| Msg_Put | Dir_X_Pend | Dir_X | | send to prefetcher | |

Table 3: $Dir_1SW$ State Machine Extensions for Cooperative Prefetch.
$Dir_1SW$ supports cooperative prefetching with a new message, `Msg_Prefetch_X` (cooperative prefetch for an exclusive copy), and a new state, `Dir_X_Pend` (cooperative prefetch exclusive pending). As the top block of the table illustrates, `Msg_Prefetch_X` obtains an exclusive copy of an idle block, records the prefetching processor's identity in a `Dir_X` block's pointer field (so the subsequent `Msg_Put` forwards the block), and is a no-op otherwise. The following three blocks show how the base protocol interacts with the new state. `Msg_Get_X` and `Msg_Get_S` trap if a prefetch is pending. A `Msg_Put` forwards data to a prefetching processor (pointed to by the pointer/counter).

---

`check_out`. In the common case, a directory responds by sending the data. The `Msg_Put` message results from an explicit `check_in` or a cache replacement of a copy of a block.

Several state transitions in Table 2 set a trap bit and trap to a software trap handler running on the directory processor (not the requesting processor), as in MIT Alewife [8]. The trap bit serializes traps on the same block. The software trap handlers will read directory entries from the hardware and send explicit messages to other processors to complete the request that trapped and to continue the program running on their processor. Traps only occur on memory accesses that violate the CICO model. Thus, programs conforming to this model run at full hardware speed. Note that protocol transitions implemented in hardware require at most a single request-response pair. State transitions requiring multiple messages are always handled by system software. Shifting the burden of atomically handling multiple-message requests to software dramatically reduces the number of transient hardware states and greatly simplifies the coherence hardware.

For programs that trap occasionally, the incurred costs should be small. These costs can be further reduced by microprocessors that efficiently support traps [21] or by adopting the approach used in Intel's Paragon computer of handling traps in a companion processor.

### 3.2 Prefetch Support

This section illustrates how $Dir_1SW$ supports cooperative prefetch, which allows communication to be overlapped with computation. $Dir_1SW$ currently supports only prefetching an exclusive copy of a block that is currently idle or is checked-out exclusive.

As Table 3 illustrates, cooperative prefetching requires a new message, `Msg_Prefetch_X` (cooperative prefetch of an exclusive copy), and a new state, `Dir_X_Pend` (cooperative prefetch pending). `Msg_Prefetch_X` completes immediately if it finds the prefetched block in state `Dir_Idle`. The message becomes a pending prefetch if the prefetched block is in state `Dir_X`. Otherwise, the message is a no-op and the block must be fetched by a subsequent `check_out`. A pending prefetch from processor $i$ sets a block's state to `Dir_X_Pend` and its pointer to $i$, so the block can be forwarded to $i$ as soon as it is checked-in. Get messages (`Msg_Get_X` and `Msg_Get_S`) conflict with a pending prefetch and trap.

`Msg_Prefetch_X` works well for blocks used by one processor at a time, called *migratory data* by Weber and Gupta [30]. It is also straightforward to augment $Dir_1SW$ to support a single cooperative prefetch of a shared copy—providing the block is idle or checked-out exclusive. It is, however, a much larger change to $Dir_1SW$ to support in hardware multiple concurrent prefetches of a block. We are investigating whether this support is justified.

### 3.3 Synchronization Support

Mellor-Crummey and Scott's locks and barriers are efficient if a processor can spin on a shared-memory block that is physically local [26]. A block is local either because it is allocated in a processor's physically local, but logically shared, memory module or because a cache-coherence protocol copies it into the local cache. The current $Dir_1SW$ is unsuitable for synchronization because the interactions do not fit the CICO model and the protocol traps on common cases, ruining performance. We currently support the first alternative with the addition of non-cacheable pages and a `swap` instruction. Both are easily implemented because they are supported by most microprocessors. We are also investigating further extensions to $Dir_1SW$ to support synchronization.

### 3.4 Discussion

$Dir_1SW$ is easier to implement than most other hardware cache-coherence mechanisms. The fundamental simplification comes from eliminating race conditions in hardware, not from reducing the number of state bits in a directory entry. Race conditions, and the myriad of transient states they produce, make most hardware cache-coherence protocols difficult to implement correctly. For example, although Berkeley SPUR's bus-based Berkeley Ownership coherence protocol [22] has only six states, interactions between caches and state machines within a single cache controller produce thousands of transient states [32]. These interactions make verification extremely difficult, even for this simple bus-based protocol. Furthermore, most directory protocols, such as Stanford DASH's [23] and MIT Alewife's [8], are far more complex than any bus-based protocol. They require hardware to support transitions involving $n$ nodes and $2n$ messages, where $n$ ranges from 4 to the number of nodes or clusters.

By contrast, the base $Dir_1SW$ protocol (without prefetching) is simpler than most bus-based cache-coherence protocols. All hardware-implemented transitions involve at most two nodes and two messages. Most bus-based protocols require transitions among at least three nodes and require more than two messages. Furthermore, adding prefetch makes $Dir_1SW$'s complexity, at most, comparable to bus-based protocols. This complexity, however, is modest compared to other directory protocols.

The principal benefit of $Dir_1SW$'s simplicity is that it allows shared-memory to be added to message-passing hardware without introducing much additional complexity. Hardware designers can continue improving the performance of this hardware and make the common cases execute quickly, rather than concentrating on getting the complex interactions correct.

The principal drawback of $Dir_1SW$ is that it runs slowly for programs that cause traps. Although programmers could avoid traps by reasoning directly about $Dir_1SW$ hardware, the CICO model provides an abstraction that hides many hardware details but still allows a programmer to avoid traps. CICO and $Dir_1SW$ are designed together so programs following the CICO model do not trap.

Traps on shared blocks are much more onerous than traps on exclusive blocks since the former requires broadcasts to force `check_in`'s of the shared copies. Programs that cannot substantially eliminate these traps will not scale on $Dir_1SW$ hardware. One solution is to extend the hardware to $Dir_iSW$, which maintains up to $i$ pointers to shared copies. $Dir_iSW$ traps on requests for more than $i$ shared copies (like Alewife [8]) and when a `check_out_X` request encounters any shared copies (unlike Alewife, which sometimes handles this transition in hardware). Like $Dir_1SW$ (and unlike Alewife), $Dir_iSW$ never sends more than a single request/response pair, since software handles all cases requiring multiple messages.

A secondary drawback of $Dir_1SW$ is its lack of hardware support for multiple concurrent prefetches of a block. Although $Dir_1SW$ efficiently supports single-producer, single-consumer relations with cooperative prefetch, multiple consumers cannot exploit cooperative prefetch. $Dir_1SW$ cooperative prefetch only records one prefetch exclusive request. Multiple consumers must obtain updated values with explicit `check_out`'s. We are unsure as to whether this behavior is a serious performance problem or of the extent to which blocking can reduce the number of consumers, so we are unwilling to extend the $Dir_1SW$ protocol yet.

| Name | Input Data Set | Cycles (billions) |
|---|---|---|
| *barnes* | 2048 bodies, 10 iter. | 3.3 |
| *ocean* | 98 × 98 2 days | 1.5 |
| *sparse* | 256 × 256 dense | 2.5 |
| *pthor* | 5000 elem, 50 cycles | 20.9 |
| *cholesky* | bcsstk15 | 21.0 |
| *water* | 256 mols, 10 iter | 9.8 |
| *mp3d* | 50000 mols, 50 iter | 24.6 |
| *tomcatv* | 1024 × 1024, 10 iter | 8.5 |

Table 4: Application Programs

This table lists the benchmarks used in this paper. *Sparse* is a locally-written program that solves $AX = B$ for a sparse matrix $A$. *Tomcatv* is a parallel version of the SPEC benchmark. All other benchmarks are from the SPLASH benchmark suite [29].

# 4 Evaluation

This section evaluates the performance of CICO and $Dir_1SW$ with a collection of eight benchmarks running on a 32-processor system. The evaluation compares the programs' performance, both with and without `check_out` and `check_in` directives, against their performance under $Dir_nNB$. This protocol maintains a record of all outstanding copies of a cache block and never broadcasts an invalidate. Unlike our earlier paper [20], we compare the programs' simulated execution times, not the number of directory events.

The measurements were collected by executing applications programs—hand-annotated with CICO directives—on a Thinking Machines CM-5 augmented with an additional layer of software to simulate $Dir_1SW$ and other protocols such as $Dir_nNB$. The combination of CM-5 hardware and a software layer is called the *Wisconsin Wind Tunnel (WWT)* [28]. WWT runs a parallel shared-memory program on a parallel message-passing computer (a CM-5) and concurrently calculates the program's execution time on the proposed system. We call WWT a *virtual prototype* because it exploits similarities between the system under design (the target) and an existing evaluation platform (the host)[7]. In WWT, the host directly executes all target instructions and memory references that hit in the target cache. *Direct execution* means that the host hardware executes target program operations—for example, a floating-point multiply instruction runs as a floating-point multiply. Simulation is only necessary for target operations that a host machine does not support. Direct execution runs orders of magnitude faster than software simulation.

WWT calculates target program execution times on the parallel host with a distributed, discrete-event simulation algorithm. WWT manages the interactions among target nodes by dividing execution into lock-step quanta that ensure all events originating on a remote node that affect a node in the current quantum are known at the quantum's beginning. On each node, WWT orders all events in a quantum and directly executes the process up to the next event.

Figure 2 shows the performance of the eight benchmarks. The base case, against which the other measurements are normalized, is $Dir_1SW$ without CICO directives. We did not enable prefetch to facilitate comparison of the basic protocols. The two bars for each benchmark show the performance of $Dir_1SW$ with `check_out` and `check_in` directives enabled and the $Dir_nNB$ protocol against the base case. In all cases, enabling the CICO directives improved performance, sometimes by as much as 30%. The largest performance improvement was in *sparse*, which was written by a programmer familiar with CICO instead of adding CICO directives to an existing program.

The figure also shows the performance of the benchmarks under the $Dir_{32}NB$ protocol. In many cases, $Dir_1SW$ with CICO directives performed nearly as well as this protocol. The three programs with large performance disparities (*ocean*, *pthor*, and *mp3d*) shared data without synchronization. This behavior is costly under $Dir_1SW$, both because it causes additional traps and broadcasts and because it makes the CICO directives difficult to employ effectively.

To examine this phenomena, we modified *mp3d* to eliminate some unsynchronized sharing and increase cache reuse. Unlike previous attempts, which rewrote the program [12], our changes were minor. The major problem is that unsynchronized races for the space cell data structure increased communication. We reduced conflicts by having processors lock the cells they operate on. To avoid spinning, processors back off from a locked cell and return to it after they have finished the other cells. The bars labeled *rmp3d* are normalized against the performance of *mp3d* and show the large effect of this minor change. The modified version ran 50% faster under $Dir_1SW$ and 13% faster under $Dir_{32}NB$ . Equally as important, the performance difference between $Dir_1SW$/CICO and $Dir_{32}NB$ decreased significantly in the restructured program.

The measurements in this section show that for a moderate-sized computer, $Dir_1SW$ performs comparably to a significantly more complex directory protocol and that the gap can be further narrowed by using the CICO annotations as memory system directive.

# 5 Related Work

Inserting CICO directives is superficially similar to inserting coherence primitives in software cache-coherent systems [13, 10, 27]. Software coherence schemes invalidate far more data than

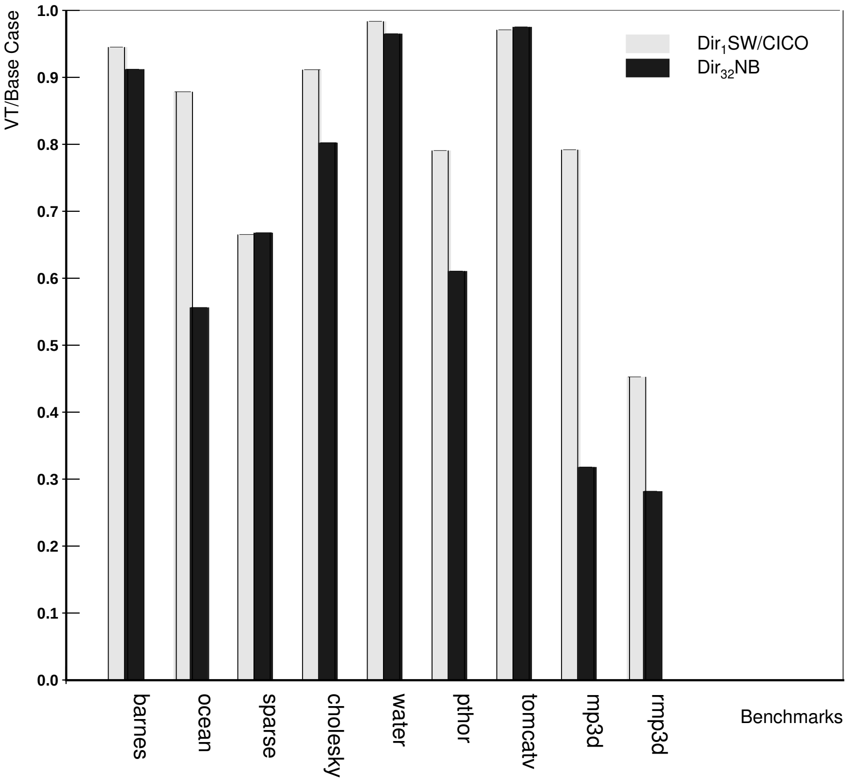


Figure 2: This graph shows the performance of each benchmark with and without CICO directives and running under the $Dir_n NB$ protocol. No prefetching is done to facilitate comparison of the basic protocols. Every bar except those labeled *rmp3d* are normalized against the benchmark's performance under $Dir_1 SW$ without CICO directives. *rmp3d* is a restructured version of *mp3d* and is normalized against *mp3d* to show the improvement. Bars labeled $Dir_1 SW$/CICO show the performance with CICO directives enabled. The bars labeled $Dir_{32} NB$ show the programs' performance without CICO directives under $Dir_{32} NB$ . A height of less than one indicates that a program ran faster than it did under $Dir_1 SW$ .

dynamically necessary for two reasons not shared by CICO [1]. First, correctness requires invalidates along all possible execution paths—even those that will not occur dynamically. Second, correctness requires conservative static analysis, which makes worst-case assumptions. $Dir_1SW$ leaves the burden of correctness with hardware, while providing software with the ability to optimize performance.

CICO's hierarchy of performance models has similar goals to Hill and Larus's models for programmers of multis [19]. CICO's models, however, provide richer options for reasoning about relinquishing data and initiating prefetches.

Many researchers have investigated using directory protocols for hardware cache coherence in large-scale shared-memory systems [2]. Stanford DASH [23] connects $n$ clusters ($n \leq 64$) with a mesh and the processors within a cluster with a bus. It maintains coherence with a $Dir_nNB$ protocol between clusters and snooping within a cluster. Each multiprocessor in Stanford Paradigm [11] connects $n$ clusters ($n \leq 13$) with a bus and uses a two-level bus hierarchy within a cluster. It uses a $Dir_nNB$ protocol between clusters and a similar protocol within each cluster. IEEE Scalable Coherent Interface (SCI) [16] allows an arbitrary interconnection network between $n$ nodes ($n < 64K$). It implements a $Dir_nNB$ protocol with a linked-list whose head is stored in the directory and other list elements are associated with blocks in processor caches. MIT Alewife [8] connects multithreaded nodes with a mesh and maintains coherence with a LimitLESS directory that has four pointers in hardware and supports additional pointers by trapping to software.

$Dir_1SW$ shares many goals with these coherence protocols. Like the other four protocols, $Dir_1SW$ interleaves the directory with main memory. Like the DASH, SCI and Alewife protocols, it allows any interconnection network. Like the SCI and Alewife protocols, $Dir_1SW$ directory size is determined by main memory size alone (and not the number of clusters). $Dir_1SW$ hardware is simpler than the other four protocols, because it avoids the transient states and races that they handle in hardware (see Section 3.1). $Dir_1SW$ relies on a model (CICO) to ensure that expensive cases (trapping) are rare. If they are common, $Dir_1SW$ will perform poorly. All four other protocols, for example, use hardware to send multiple messages to handle the transition from four readers to one writer. $Dir_1SW$ expects the readers to check-in the block and traps to software if this does not happen.

Both Baylor et al. [4] and $Dir_1SW$ use a counter to track extant shared copies. On a write, invalidations are sent to all processors, but only acknowledged by processors that had copies of the block (Chaiken [9] calls this approach a *notifying* implementation of $Dir_iB$). Unlike $Dir_1SW$, Baylor et al. do not discuss returning a block to the idle state when all copies are returned, probably because this is unlikely to occur without CICO.

We are aware of two other efforts to reduce directory complexity. Archibald and Baer [3] propose a directory scheme that uses four states and no pointers. As mentioned above Alewife [8] uses hardware with four pointers and traps to handle additional readers. Both are more complex than $Dir_1SW$, because both must process multiple messages in hardware. Archibald and Baer must send messages to all processors to find two or more readers, while Alewife hardware uses multiple messages with 1–4 readers. $Dir_1SW$'s trapping mechanism was inspired by Alewife's.

$Dir_1SW$ supports software-initiated prefetches [6, 15] that leave prefetched data in cache, rather than registers, so data prefetched early do not become incoherent. $Dir_1SW$'s cooperative prefetch support also reduces the chance that data are prefetched too early since a prefetch remains pending until a block is checked-in. This avoids having the block ping-pong from the prefetcher to the writer and back. Similar, but richer support is provided by QOSB [14], now called QOLB. QOLB allows many prefetchers to join a list, spin locally, and obtain the data when it is released. $Dir_1SW$ supports a single prefetcher (per block) with much simpler hardware than QOLB, but it does not provide good support for multiple concurrent prefetchers (for the same block). Finally, cooperative prefetch always maintains naive shared-memory semantics, whereas a process issuing a QOLB must ensure that it eventually releases the block.

# 6 Conclusions

Shared memory offers many advantages, such as a uniform address space and referential transparency, that are difficult to replicate in today's massively-parallel, message-passing computers. We believe the absence of massively-parallel, shared-memory machines follows from the lack of a programming performance model that identifies both the common and expensive operations so programmers and hardware designers can improve programs and hardware.

In our view, the key to effective, scalable, shared-memory parallel computers is to address the software and hardware issues together. Our approach to building shared-memory software and hardware, called *cooperative shared memory*, provides programmers with a realistic model of which operations are expensive; programmers and compilers with performance primitives that can be used aggressively, because they do not change semantics; and hardware designers with a description of which cases are common.

Our initial implementation of cooperative shared memory uses a simple programming performance model, called *Check-In/Check-Out (CICO)*, and even simpler hardware called $Dir_1SW$. CICO provides a metric by which programmers can understand and explore alternative designs on any cache-coherent parallel computer. In the CICO model (Section 2), programs bracket uses of shared data with `check_out` directives that indicate whether a process expects to use a datum exclusively and `check_in` directives that terminate an expected use. CICO's new approach encourages programmers to identify intervals in which data is repeatedly used, rather than focusing on isolated uses, and to explicitly acknowledge when data can be removed from local buffers. An additional *cooperative prefetch* directive allows a program to anticipate an upcoming `check_out` and hide communication latency.

$Dir_1SW$ is a minimal directory protocol that adds little complexity to the hardware of a message-passing machine, but efficiently supports programs written within the CICO model. It uses a single pointer/counter field to either identify a writer or count readers. Simple hardware entirely handles programs conforming to the CICO model by updating the pointer/counter and forwarding data to a requesting processor. No case requires multiple messages (beyond a single request-response pair) or hardware transient states. Programs not conforming to the model run correctly, but cause traps to software trap handlers that perform more complex operations.

An evaluation of CICO and $Dir_1SW$ on the Wisconsin Wind Tunnel (WWT) illustrates the effectiveness of the CICO programming model and the competitive performance of the simple $Dir_1SW$ hardware. Furthermore, the results provide strong evidence for the virtual prototyping method, since with less than a person-year of effort we can run $Dir_1SW$ programs and collect statistics at speeds comparable to real machines.

We are seeking to refine cooperative shared memory and enhance WWT [28, 31]. A promising approach, for example, is to sequence directory operations in software to enable higher-level programmer or compiler directives (e.g., vector `check_out`). We are studying cooperative prefetch, non-binding prefetch, and other variants. We will study programs running on kiloprocessor shared-memory systems by extending WWT to simulate multiple nodes on each host node.

# 7 Acknowledgements

Satish Chandra, Glen Ecklund, Rahmat Hyder, Alvy Lebeck, Jim Lewis, Shubhendu Mukherjee, Subbarao Palacharla, and Timothy Schimke helped develop the Wisconsin Wind Tunnel and applications. Dave Douglas, Danny Hillis, Roger Lee, and Steve Swartz of TMC provided invaluable advice and assistance in building the Wind Tunnel. Sarita Adve, Jim Goodman, Guri Sohi, and Mary Vernon provided helpful comments and discussions. Singh *et al.* [29] performed an invaluable service by writing and distributing the SPLASH benchmarks.

# 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] 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.

[3] James Archibald and Jean-Loup Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 355–362, June 1984.

[4] Sandra Johnson Baylor, Kevin P. McAuliffe, and Bharat Deep Rathi. An Evaluation of Cache Coherence Protocols for MIN-Based Multiprocessors. In *International Symposium on Shared Memory Multiprocessing*, pages 230–241, Tokyo, April 1991.

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

[6] David Callahan, Ken Kennedy, and Allan Poterfield. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 40–52, April 1991.

[7] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriguez-Rosell. A Virtual Machine Emulator for Performance Evaluation. *Communications of the ACM*, 23(2):71–80, February 1980.

[8] David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[9] David Lars Chaiken. Cache Coherence Protocols for Large-Scale Multiprocessors. Technical Report MIT/LCS/TR-489, MIT Laboratory for Computer Science, September 1990.

[10] J. Cheong and A.V. Veidenbaum. A Cache Coherence Scheme With Fast Selective Invalidation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 299–307, June 1988.

[11] David R. Cheriton, Hendrick A. Goosen, and Patrick D. Boyle. Paradigm: A Highly Scalable Shared-Memory Multiprocessor. *IEEE Computer*, 24(2):33–46, February 1991.

[12] David R. Cheriton, Hendrik A. Goosen, and Philip Machanick. Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience. In *International Symposium on Shared Memory Multiprocessing*, pages 109–118, April 1991.

[13] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic Management of Programmable Caches. In *Proceedings of the 1988 International Conference on Parallel Processing (Vol. II Software)*, pages 229–238, Aug 188.

[14] 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.

[15] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, June 1991.

[16] David B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(2):10–22, February 1992.

[17] David B. Gustavson and David V. James, editors. *SCI: Scalable Coherent Interface: Logical, Physical and Cache Coherence Specifications*, volume P1596/D2.00 18Nov91. IEEE, November 1991. Draft 2.00 for Recirculation to the Balloting Body.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[19] Mark D. Hill and James R. Larus. Cache Considerations for Programmers of Multiprocessors. *Communications of the ACM*, 33(8):97–102, August 1990.

[20] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 262–273, October 1992.

[21] Douglas Johnson. Trap Architectures for Lisp Systems. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 79–86, June 1990.

[22] Randy H. Katz, Susan J. Eggers, David A. Wood, C.L. Perkins, and R.G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985.

[23] 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.

[24] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[25] Calvin Lin and Lawrence Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing (Vol. II Software)*, pages II–163–170, August 1990.

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

[27] Sang Lyul Min and Jean-Loup Baer. A Timestamp-based Cache Coherence Scheme. In *Proceedings of the 1989 International Conference on Parallel Processing (Vol. I Architecture)*, pages I–23–32, August 1989.

[28] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[29] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[30] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 243–256, April 1989.

[31] 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.

[32] David A. Wood, Garth G. Gibson, and Randy H. Katz. Verifying a Multiprocessor Cache Controller Using Random Case Generation. *IEEE Design and Test of Computers*, 7(4):13–25, August 1990.