Appears in: "Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)," October 1992. 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

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA

Abstract

We believe the absence 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* annotation marking the expected first use and a *check-in* annotation terminating the expected use of the data. A *cooperative prefetch* annotation 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.

*This work is supported in part by NSF Presidential Young Investigator Awards CCR-9157366 and MIPS-8957278, NSF Grant CCR-9101035, a University of Wisconsin Graduate School Grant, a Wisconsin Alumni Research Foundation Fellowship and by donations from A.T.&T. Bell Laboratories, Cray Research Foundation and Digital Equipment Corporation. Our department's Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School.

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. 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* [22]. Indeed, many existing shared-memory programs would perform poorly on massively-parallel systems because the programs were written under the predominate naive model, which assumes all memory references have equal cost. This assumption is wrong because remote references require communication and run slower than local references [16]. For example, a remote memory reference on Stanford DASH costs at least 100 times more than a local reference [20].

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* [4] 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, 14, 20]. 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 [20]), 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 programs' 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 exclusively and *check-in* annotations that terminate an expected use. CICO's new approach encourages programmers to identify intervals in which data is repeatedly used, rather than focusing than 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 forwarding of data.

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. Programs not conforming to the CICO model run correctly, but trap to system software that performs more complex operations (in a manner similar to MIT Alewife [6]).

Measurements of programs from the SPLASH benchmark suite [25] illustrate the effectiveness of the CICO model (Section 4). Finally, Section 5 discusses related work.

2 CICO Programming Performance Model

The performance component of our shared-memory programming model is named *Check-In / Check-Out (CICO)*. It serves two roles. The descriptive role is to identify points in a program's execution at which communication will occur in any cache-coherent shared-memory system. The prescriptive role is to describe how a programmer or compiler can reduce the cost of this communication. This section describes CICO and illustrates how it identifies communication and ways to reduce its cost. The next section (Section 3) shows that hardware can also exploit CICO annotations to further speed execution. CICO has two aspects. The first is a set of annotations that bracket uses of shared data. The second is a hierarchy of models that identify and assign cost to communication.

2.1 CICO Annotation

CICO begins with programmer-supplied annotations that elucidate a program's memory references. These annotations describe a program's behavior and do not affect its semantics, even if misapplied.

CICO assumes memory is divided into aligned, fixed-sized blocks. The base model provides a programmer with three performance annotation:

`check_out_X`	Expect exclusive access to block
`check_out_S`	Expect shared access to block
`check_in`	Relinquish a block

`check_out_X` asserts that the processor performing a 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. Exclusive access is not synonymous with write access. Shared access is fundamentally more difficult to manage in hardware, so a single reader should check-out a block exclusive. Finally, `check_in` asserts that the processor performing the check-in expects another processor to access the checked-out block next. *These annotations are advisory and do not imply any mechanism to ensure exclusive access.*

The annotations play a descriptive role by identifying points in a shared-memory program at which communication occurs and by identifying program behavior that causes unnecessary communication. This communication occurs in all cache-coherent, shared-memory systems, not just Dir_1SW. By making a program conform to the CICO model, a programmer or compiler can improve the program's performance by reducing unintended communication caused by ping-ponging and false sharing. The CICO model can also be employed to improve conforming programs by identifying and further reducing communication.

All communication, however, cannot be eliminated from parallel programs. The next step is to reduce the impact of communication by overlapping it with computation. The first extension to the model provides two additional, *cooperative prefetch* annotations:

`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 will likely check-out the block for exclusive (shared) access in the near future. Prefetches encourage a programmer or compiler to identify, in advance, data that a processor will use and consider how to obtain it while computation is still progressing.

Check-in's in the CICO model permit a concise description of this 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 arrives 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 Model

CICO provides a hierarchy of performance models that elucidate an annotated shared-memory program's communication. The collection of models enables a programmer to apply an appropriate model to each part of a program. Time-critical sections can be analyzed in depth, while less important code, such as initialization or error-handlers, can be understood with naive shared memory assumptions.

Model Level 0. The simplest model ignores CICO annotations and corresponds to naive shared memory.

Model Level 1. The first real CICO model considers only communications induced by *check-out's* and *check-in's*. A parallel program's shared-memory cost is proportional to the number of communication events along its critical path. A communication event occurs when a processor:

- encounters a `check_out` annotation for a block previously checked-in,
- violates a CICO annotation's exclusivity assumption (by accessing a block while another processor has it `check_out_X` or by writing a block while it is `check_out_S`).

Even this simple model can be applied to reduce communication. An important way is to increase exploitation of temporal locality, so a block is heavily used before being relinquished. Another way is to reduce inter-processor sharing, with its concomitant *check-in*, *check-out* events. Finally, eliminating unsynchronized true sharing (data races) reduces violations of the CICO model. CICO does not provide explicit direction as to how to modify a particular program. Instead, it provides a metric by which a programmer can explore alternative designs.

Model Level 2. An important refinement is prefetching, which can reduce the cost of a `check_out` of a block. Consequently, the first rule must be refined to initiate a communication event only when a processor:

- encounters a `check_out` annotation for a block previously checked-in and which has not been prefetched by the processor executing the `check_out_X` .

Model Refinements. CICO models cache-coherent shared-memory multiprocessors. The physical limitations of real caches—such as multi-word blocks and finite associativity and size—produce an orthogonal set of performance considerations that strongly affect the application of the CICO model to real computers. For example, multi-word blocks may cause CICO violations due to the well-known problem of false sharing, which occurs when processors unwittingly use different locations in the same block for conflicting purposes [10]. Similarly, finite caches cause unintended communication when a cache block is replaced (i.e., implicitly checked-in). Detailed application of CICO must consider the behavior characteristics of real caches [19].

2.3 Example

Consider an example. Figure 1a outlines a stencil algorithm that sets each element of a two-dimensional array to the average of its nearest neighbors. Figure 1b shows the same program with CICO annotations. Each processor computes the average for a contiguous group of columns and shares its neighbors' boundary columns. The loop contains two intervals. In the first, each processor copies its neighbors' boundary columns into local arrays, to avoid data races when the columns are updated. In the second, each processor uses the local arrays to compute averages for its columns.

The Level 1 CICO model points out that communication in this algorithm occurs because of the boundary columns. One approach to reducing communication would be to iterate the stencil computation over the local columns several times before reading updated values from neighboring processors.

Another approach is to apply the Level 2 CICO model and use prefetch to hide the communication latency. Figure 2 shows how cooperative prefetch works for this example. All processors begin each interval by prefetching the shared columns that they will access in the subsequent interval. The prefetches remain pending because the columns are already checked-out by the other processor. At the end of an interval, as a processor checks-in its columns, pending prefetches succeed and the columns are checked-out to the other processor and transferred to it, in advance of its explicit check-out.

```
-- Code for processor P
LOOP
   Cleft := A[*, L-1]
   Cright := A[*, U+1]
   BARRIER
   -- Compute stencil on columns L..U
   BARRIER
END LOOP
```

Part (a).

```
-- Initialize prefetch for iteration 1
prefetch_X A[*, L-1], A[*, U+1]
BARRIER
LOOP
   -- Interval 1
   prefetch_X A[*, L], A[*, U]
   check_out_X A[*, L-1], A[*, U+1]
   Cleft := A[*, L-1]
   Cright := A[*, U+1]
   check_in A[*, L-1], A[*, U+1]
   BARRIER
   -- Interval 2
   prefetch_X A[*, L-1], A[*, U+1]
   check_out_X A[*, L], A[*, U]
   -- Compute stencil on columns L..U
   check_in A[*, L], A[*, U]
   BARRIER
END LOOP
```

Part (b).

Figure 1: Example of cooperative prefetch in column-blocked stencil. The example is simplified. If the matrix is large enough, the outer loop would have to be blocked so the columns fit into a processor's cache.

2.4 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 annotations described above are unsuitable for synchronization constructs, which require competitive (i.e., unordered and unpredictable) memory access. Rather than extend CICO with annotations 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.

2.5 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

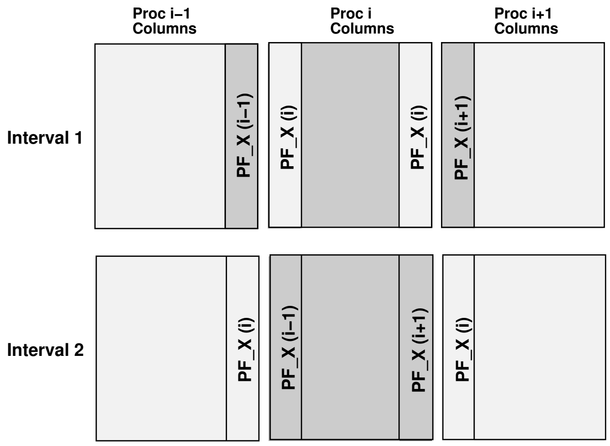

Figure 2: Illustration of cooperative prefetch in column-block stencil. The lightly shaded columns are checked-out by processor $i-1$ or $i+1$. The dark columns are checked-out by processor i. The annotations describe pending prefetches.

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 annotations are well-suited to compiler analysis since they do not affect a program's semantics. The analysis to employ an annotation need not be conservative. Instead, a compiler can optimize the expected case without considering the effects of annotations 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.

2.6 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 annotations are unnecessary for correct execution. Programmers can incrementally employ them to understand and optimize time-critical routines.

- The annotations can be used aggressively to optimize expected program behavior since they do not affect program semantics. The other cases still function correctly.

- The annotations do not change a datum's address. A programmer can optimize one routine without changing all routines that interact with it.

- The annotations 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 [15]. 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 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 1 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 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 1 set a trap bit and trap to a software trap handler running on the directory processor (not the requesting processor), as in MIT Alewife [6]. 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 [17] 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 2 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 [26]. 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 [23]. A block is local

[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_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 1: 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.

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 2: 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).

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-cachable 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 [18] has only six states, interactions between caches and state machines within a single cache controller produce thousands of transient states [27]. These interactions make verification extremely difficult, even for this simple bus-based protocol. Furthermore, most directory protocols, such as Stanford DASH's [20] and MIT Alewife's [6], 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 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 [6]) 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-outs. 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.

4 Preliminary Evaluation

This section contains a preliminary evaluation of CICO and Dir_1SW. For two reasons, the evaluation focuses on the number of traps incurred by applications. First, a trap means that a program does not conform to the CICO model. The measurements below demonstrate that a large body of existing code can be adapted to the CICO model so traps occur infrequently and have little or no effect on program performance. Second, our tools currently record only event counts, which cannot be used to predict parallel speedup or measure prefetch effectiveness. We are extending the metrics so the applications' parallel performance can be accurately characterized.

The measurements were collected by executing applications programs—hand-annotated with CICO annotations—on a Thinking Machines CM-5 augmented with an additional layer of software to simulate Dir_1SW. The combination of CM-5 hardware and a software layer is called the *Wisconsin Wind Tunnel (WWT)* and is an example of a new simulation technique called virtual prototyping. A *virtual prototype* consists of a target machine interface on top of a commercial hardware platform. By using software, or a combination of hardware and software, this interface layer supports the functional machine interface of a target machine. Whenever possible, target system software runs native on the commercial platform. Only target system features not supported by the platform are simulated in software or implemented in custom hardware.

WWT manipulates the CM-5 processor node virtual memory maps and ECC (error-correcting code) bits to trap on memory references to cache blocks that would not be in a target machine's cache. ECC, which is calculated on double words, permits a much finer granularity of sharing than does the page-level protection used in shared virtual memory [21]. ECC errors and page access violations trap to the Dir_1SW simulator, which models the Dir_1SW hardware by sending explicit messages to other processors to obtain cache blocks and maintain coherence. Once brought into a node's cache, a block is accessed by the local processor at full speed, without simulator intervention, just as it would be on Dir_1SW hardware.[2]. WWT captures all non-local memory references and accurately records all transitions of Dir_1SW hardware, regardless of the accuracy or completeness of the CICO annotations. Furthermore, WWT's speed has allowed us to examine large application (a total of almost 80

[2] One exception: unlike the real hardware, all writes to pages containing read-only blocks trap.

Name	Application	Input Data Set	Size (lines)	# CICO Annotations	Instructions (millions)
barnes	Barnes-Hut N-body simulation	2k bodies	2775	112	174
cholesky	Cholesky factorization of sparse matrix	bcsstk15	1888	72	5,522
locus	Standard cell wire routing	Primary1	7001	125	927
mm	Blocked matrix multiply	256×256	395	37	1,533
mp3d	Hypersonic flow simulation	50000 mols	1607	62	3,762
pthor	Digital circuit simulation	risc	9200	758	5,917
tomcatv	Parallel SPEC benchmark	1024×1024 10 iter	404	68	25,296
water	Water molecule simulation	512 mols 10 iter	1466	73	37,479

Table 3: Example applications.

This table describes the applications. With the exception of *mm* and *tomcatv*, they are SPLASH benchmarks [25]. (The SPLASH benchmark *ocean* is written in Fortran, which does not yet run on WWT.) Column *Size* lists the original size of each application, in lines of code. The next column lists the number of CICO annotations added to each program. The final column lists the number of instructions (in millions) executed by each program.

billion instructions) and large data sets.

4.1 Applications

The SPLASH benchmark suite is a collection of explicitly-parallel, shared-memory applications [25]. We added CICO primitives, by hand, to the SPLASH benchmarks (except *ocean*, which is written in Fortran) and two additional programs (*mm*, a blocked matrix multiply, and *tomcatv*, a parallel version of the SPEC benchmark). Table 3 describes the applications.

We annotated these programs by finding memory references that access shared data and inserting an explicit check-out annotation for each accessed block. Note that WWT also implicitly checks-out a block at each memory reference that misses in a processor's cache, so the statistics are accurate, even where our annotations are incomplete. We also added explicit check-in annotations to release shared data before it is used by other processors. Finally, we added some prefetch annotations, although they were not the focus of this study.

In some cases, inserting these annotations required changes to a program to introduce additional synchronization. For example, in *water*, most of the interprocess communication occurs when each processor updates the force vectors between its molecules and other processors' molecules. In the original code, each molecule is protected by a lock that prevents simultaneous updates. This approach does not work well for CICO since the sharing pattern for each molecule is unpredictable and, hence, ill-suited for prefetching. Since conflicting molecule accesses are infrequent (1 of every N/P updates, where N is the number of molecules and P is the number processors) pair-wise locks can be replaced by two general barriers that synchronize conflicting accesses. Immediately before reaching a barrier, a processor prefetches the next (conflicting) molecule and checks-in its molecule. After updating the conflicting molecule, a processor releases it and prefetches back its original molecule. Another barrier synchronizes these prefetches. The processors then update the next N/P conflict-free molecules without synchronization.

4.2 Results

Figure 3 categorizes how check-outs perform with Dir_1SW.[3] Check-outs that reference blocks already in the local cache have minimal cost, while those causing traps are the most expensive. Other check-outs are handled by Dir_1SW hardware at an intermediate cost that depends on the effectiveness of prefetching. The bar labeled *Prefetch* lists the fraction of check-outs that execute after a prefetch. The bar labeled *No Prefetch* gives check-outs whose full latency is seen.

The measurements illustrate the usability of the CICO programming model. By annotating uses of shared data structures in these programs, we were able to almost entirely avoid traps. Table 4 lists the number of traps that occur and calculates their overhead cost under a plausible assumption about trap handler execution cost. The three applications with a non-negligible number of traps (*locus*, *mp3d*, and *pthor*) were written to use unsynchronized sharing (data races) that does not conform to the CICO model. Even so, we were able to reduce the cost of trap handling to a low level. We also believe that trap

[3] We do not report check-ins or prefetches because they do not cause traps and their latency overlaps computation.

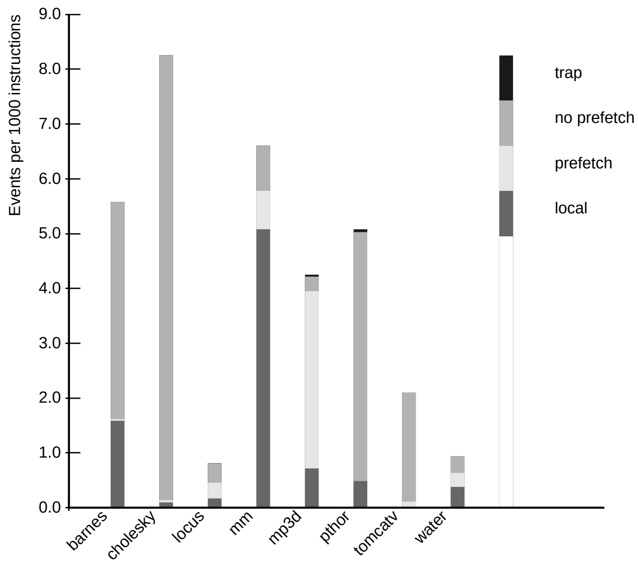

Figure 3: This histogram describes the check-out behavior of Dir_1SW. Each bar describes the state of a block when a check-out executes. Check-outs, which can incur substantial costs, are broken down in the following way. *Local* is the fraction of checked-out blocks that were already in a processor's local cache. *Prefetch* is the fraction of checked-out blocks that were previously prefetched. *No Prefetch* is the fraction of checked-out blocks that were not prefetched. *Traps* is the fraction of check-outs that cause a trap.

Name	**# Traps**	**# Broadcast Traps**		**Overhead (per 1000 inst)**
barnes	509	58	(11%)	0.3
cholesky	21,202	16	(0%)	2.1
locus	3,643	2,093	(57%)	12.3
mm	50	0	(0%)	0.0
mp3d	126,635	19,373	(15%)	43.0
pthor	286,156	2,417	(1%)	27.6
tomcatv	2,060	34	(2%)	0.0
water	0	0	(0%)	0.0

Table 4: Effect of traps.

This table lists the number of traps occurring during each program's execution and the fraction of traps requiring a broadcast. The last column is a calculation of the trap handler overhead cost under the assumption that a non-broadcast trap consumes 500 and a broadcast trap requires 5,000 instruction executions.

overhead will decrease when the programs execute on larger data sets, which will reduce the likelihood of traps.

The results also provide strong evidence in favor of virtual prototyping. Unlike building hardware, we have run large programs after less than a person-year of effort. Unlike pure simulation, our system executes programs and collects statistics at close-to-hardware speeds. The current, untuned system executes the almost 80 billion instructions in the applications in several hours.

5 Related Work

Inserting CICO annotations is superficially similar to inserting coherence primitives in software cache-coherent systems [9, 7, 24]. Software coherence schemes invalidate far more data than 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 [16]. 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 [20] 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 [8] 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) [13] 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 [6] 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.

Dir_1SW shares many goals with these coherence protocols. Like all 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, 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.

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 [6] 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 that leave prefetched data in a cache. Like [5, 12] we do not prefetch into registers, so data prefetched early does not become incoherent. Dir_1SW's cooperative prefetch support also reduces the chance that data is 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 [11], 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* annotations that indicate whether a process expects to use a datum exclusively and *check-in* annotations 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* annotation 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 transient states. Programs not conforming to the model run correctly, but cause traps to software trap handlers that perform more complex operations.

A preliminary evaluation of CICO and Dir_1SW on the Wisconsin Wind Tunnel (WWT) illustrates the effectiveness of the CICO programming model and shows that Dir_1SW traps can be avoided. 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.

7 Acknowledgements

Satish Chandra, Glen Ecklund, Alvy Lebeck, Jim Lewis, 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.* [25] performed an invaluable service by writing and distributing the SPLASH benchmarks. Michael Wolf provided the *mm* benchmark.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Susan J. Eggers and Randy H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 257–270, 1989.

[11] 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.

[12] 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.

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

[14] 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.

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

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.