Appears in: *Proceedings of the 20th Annual International Symposium on Computer Architecture,* May 1993.

# Mechanisms for Cooperative Shared Memory*

David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus,
Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, Steven K. Reinhardt
wwt@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA

## Abstract

This paper explores the complexity of implementing directory protocols by examining their *mechanisms*—primitive operations on directories, caches, and network interfaces. We compare the following protocols: $Dir_1B$, $Dir_4B$, $Dir_4NB$, $Dir_nNB$ [2], $Dir_1SW$ [9] and an improved version of $Dir_1SW$ ($Dir_1SW^+$). The comparison shows that the mechanisms and mechanism sequencing of $Dir_1SW$ and $Dir_1SW^+$ are simpler than those for other protocols.

We also compare protocol performance by running eight benchmarks on 32 processor systems. Simulations show that $Dir_1SW^+$'s performance is comparable to more complex directory protocols. The significant disparity in hardware complexity and the small difference in performance argue that $Dir_1SW^+$ may be a more effective use of resources. The small performance difference is attributable to two factors: the low degree of sharing in the benchmarks and Check-In/Check-Out (CICO) directives [9].

**Keywords:** Shared-memory multiprocessors, memory systems, cache coherence, directory protocols, and hardware mechanisms.

*This work is supported in part by NSF PYI Awards CCR-9157366 and MIPS-8957278, NSF Grant CCR-9101035, Univ. of Wisconsin Graduate School Grant, Wisconsin Alumni Research Foundation Fellowship and donations from A.T.&T. Bell Laboratories and Digital Equipment 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.



## 1 Introduction

Directory protocols are a technique used to implement cache coherence on large-scale shared-memory parallel computers [2]. Directory protocols logically associate a directory entry with each aligned block in main memory. This entry records that the block is idle (no cached copies), one writable copy exists, or one or more read-only copies exist. We only consider write-invalidate protocols that invalidate outstanding copies of a block in other processors when a processor wishes to write into it. To facilitate invalidations, each directory entry also contains logical pointers to some or all of the processor(s) that hold copies of the block. Agarwal et al. [2] use the notation $Dir_iB$ to denote protocols that explicitly record the $i$ processors that share a block and rely on broadcasts to invalidate more than $i$ processors. $Dir_iNB$ denotes protocols that avoid broadcast by preventing more than $i$ processors from sharing a block.

We examine $Dir_nNB$, $Dir_4B$, $Dir_4NB$, $Dir_1B$, $Dir_1SW$, and $Dir_1SW^+$. The Stanford DASH project and IEEE Scalable Coherent Interface (SCI) implement $Dir_nNB$ [13, 8]. DASH uses a bit vector pointing to a maximum of 16 clusters, while SCI uses a linked-list whose head is stored in the directory and other list elements are associated with blocks in a maximum of 64K processor caches.

$Dir_4B$ and $Dir_4NB$ were inspired by empirical data suggesting that, in many sharing patterns, the number of readers is lower than four, regardless of the system size [17]. MIT Alewife's [4] LimitLESS directory contains four hardware pointers and uses software to record additional pointers. LimitLESS's software can implement $Dir_4B$, $Dir_4NB$, $Dir_nNB$, and other alternatives.

$Dir_1B$, and our protocols, $Dir_1SW$ and $Dir_1SW^+$, record only one writer or reader. The limited state reduces implementation complexity, but can cause many broadcasts. $Dir_1SW$ [9] (reviewed in Section 2) and $Dir_1SW^+$ (introduced in Section 3.6.3) count the readers so they can return the directory to idle when all readers return the block, thereby avoiding an unnecessary broadcast. Programmers or compilers can also produce more desirable sharing patterns by reasoning about the shared-memory communication in a program with the *Check-In/Check-Out (CICO)* programming model. Furthermore, CICO primitives also serve as memory system directives that improve performance. We review this approach—*cooperative shared memory*—in Section 2 [9].

Many directory protocols are complex and require considerable hardware, which reduces the attractiveness of shared-memory machines. A directory protocol *policy* describes its response to program events, such as loads and stores, and the interactions among directories and caches on different processors. At the next lower level of abstraction, these policies are implemented with *mechanisms*—operations on directories, caches, and network interfaces—such as updating a directory pointer, replacing a cache block, and sending a point-to-point message. Describing a directory protocol at the mechanism level exposes disparties in protocol complexity that are not apparent at the policy level. Most protocols, for example, have policy transitions from many readers to one writer and from one writer to another writer. The shared-exclusive transition is more complex to implement than the exclusive-exclusive transition. The difference becomes clear at the mechanism level. Most systems synthesize the shared-exclusive transition by sending a sequence of invalidate and acknowledgement messages. An implementation must (a) sequence through a large number of message sends, (b) count the acknowledgements, (c) ensure concurrent requests to the same directory entry are serialized, and (d) guarantee that the interaction of these messages with messages for the node's processor, cache, and other directory entries cannot cause network deadlock. On the other hand, for an exclusive-exclusive transition, the directory only sends a single invalidation, which greatly simplifies these considerations.

The first contribution of this paper is to explore the complexity of $Dir_1B$, $Dir_4B$, $Dir_4NB$, $Dir_nNB$ [2], and $Dir_1SW$ [9] at the mechanism level of abstraction (Section 3). The mechanisms and mechanism sequencing of $Dir_1SW$ are significantly simpler than these other protocols because the shared-to-exclusive transition is not handled by hardware (MIT LimitLESS is more complex that $Dir_1SW$, but much simpler than the other protocols). $Dir_1SW$'s mechanisms can also be used to implement a protocol with higher performance. We call the best extended protocol $Dir_1SW^+$.

However, this comparison is an academic exercise if simpler protocols perform poorly. Several papers have examined directory protocol performance. Agarwal et al. [2] presented event counts for four-processor VAX traces less than two million instructions long (half-million per processor). Weber and Gupta [17] used five benchmarks, more processors (up to 16), and longer traces (4 million instructions). A major contribution of their paper is a classification of shared objects (into read-only, migratory, synchronization, mostly-read, and frequently read/written). In addition, for migratory data, they suggested flushing data from a cache. Their second paper [7] switched to the MIPS architecture, ran the applications to completion (up to 2–48 million references—instruction counts not given), and extended results to 32 processors. This change may be due to longer traces and different synchronization assumptions. Lenoski et al. [13] presented speedup measurements from the Stanford DASH prototype running three applications. Since prototypes of other directory systems were unavailable, the paper could not compare DASH against alternatives. Chaiken [5] compared LimitLESS against $Dir_4NB$ and $Dir_nNB$, using several applications on 16 and 64 processors with 7 to 30 million references per application. His principal result was that LimitLESS's performance is comparable to $Dir_nNB$ and better than $Dir_4NB$ (even though he assumes read-only data is handled separately). Hill et al. [9] measured the performance of $Dir_1SW$ by recording event counts, but did not compare their results with other protocols.

The second contribution of this paper is a comparison of directory protocol performance that extends previous work in three ways (Section 4). First, our results come from executing billions, not millions, of instructions. Second, we evaluate performance with execution time, not event counts. Third, we present results for $Dir_1B$, $Dir_4NB$, $Dir_4B$, $Dir_nNB$, $Dir_1SW$, and $Dir_1SW^+$ together. Our simulations show that $Dir_1SW^+$'s performance is similar to more complex directory protocols for seven of eight benchmarks on a system of 32 processors ($Dir_nNB$ preforms better on *mp3d* due to unscalable, unsynchronized sharing). If this result holds for other applications and larger systems, the significant disparity in hardware complexity and the small difference in performance argue that $Dir_1SW^+$ may be a more effective use of resources. The small performance difference between $Dir_1SW^+$ and the more complex protocols is attributable to two factors. First, as Weber and Gupta's measurements show, the number of outstanding shared copies is typically close to one and rarely much greater [17]. This small amount of sharing means that directory protocols that track many outstanding copies provide functionality that is not fully utilized and their additional hardware rarely improves performance. Second, CICO memory system directives reduce sharing even further.

After the principal results in Sections 3 and 4, Section 5 discusses the implication of technology trends and directions for future work, while Section 6 draws conclusions.

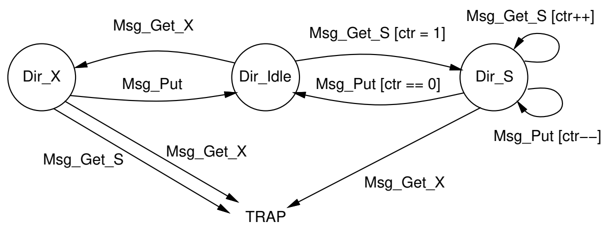


Figure 1: Base $Dir_1SW$ Protocol

## 2 CICO and $Dir_1SW$

This section reviews cooperative shared memory, CICO and $Dir_1SW$, originally presented in Hill et al. [9]. The *Check-In/Check-Out (CICO)* programming performance model allows a programmer both (1) to reason about the communications caused by shared-memory references and (2) to pass performance directives to the memory system. Neither the programming model or the directives are specific to $Dir_1SW$. Elsewhere, we demonstrate that the annotations can be used to improve program performance by increasing cache reuse and reducing program sharing [12]. This paper examines the effect on directory protocol behavior of using CICO annotations as memory system directives. We do not discuss the cooperative prefetch mechanism.

In CICO, programmers 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. In programs conforming to the model, processors coordinate access to exclusive (writable) cache blocks to avoid expensive invalidates. The primary effect of using CICO annotations as memory system directives is to have `check_in`'s flush cache blocks back to memory.

The base $Dir_1SW$ protocol associates two state bits, a trap bit, and a pointer/counter with each block in memory. A directory entry can be in one of three states: `Dir_X`, `Dir_S`, and `Dir_Idle`. State `Dir_X` implies that the directory has given an exclusive copy of the block to the processor pointed to by the pointer/counter. State `Dir_S` implies that the directory has given out $N$ shared copies, where $N$ is the number in the pointer/counter. State `Dir_Idle` implies that the directory owns the only valid copy of the block.

Figure 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 relinquishing 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. A processor sends a `Msg_Put` message on an explicit `check_in` or a cache replacement of the block.

Several state transitions in Figure 1 set a trap bit and trap to a software trap handler running on the directory processor (not the requesting processor), as in MIT LimitLESS [4]. The trap bit serializes traps from multiple references to a block. The software trap handler reads directory entries from the hardware and sends explicit messages to other processors to complete the request that trapped and then restarts the program that faulted. Traps only occur on memory accesses that violate the CICO model. Thus, programs conforming to this model run at full hardware speed. Traps on blocks in state `Dir_X` interact with one processor, while *traps in state `Dir_S` must broadcast to recall all read-only copies.* While broadcast cannot be used in infinitely large systems, it is acceptable in finite systems if the frequency of broadcast times the cost of the broadcast is small.

## 3 Directory Mechanisms

The hardware base of cache-coherent shared memory is similar to a message-passing machine. Each processor node contains a microprocessor, a cache, and a memory module. Nodes are connected with a fast point-to-point network. Shared memory differs because each memory module is addressed in a global address space and each processor node contains additional hardware to implement a directory protocol. Moreover, many directory protocols are complex and require considerable hardware, which reduces the attractiveness of shared-memory machines.

A directory protocol can be decomposed into three levels of abstraction. *Policy* describes its response to program events, such as loads and stores, and the interactions among directories and caches on different processors. At the next lower level of abstraction, policies are implemented with *mechanisms*—operations on directories, caches, and network interfaces—such as updating a directory pointer, replacing a cache block, and sending a point-to-point message. Mechanisms are further decomposed into primitive operations on a particular hardware *implementation*, which is the lowest level of abstraction. For example, $Dir_1SW$ requires a mechanism to increment the directory's pointer/counter and has a policy to increment this counter on a `Msg_Get_S` message that finds a block in state `Dir_S`. This mechanism may, in turn, be implemented as an atomic sequence of primitive hardware operations that read, add one to, and write the counter.

Policy and mechanisms can be implemented in either hardware or software. Most directory protocols implement both policy and mechanisms in hardware. However, both LimitLESS [4] and $Dir_1SW$ [9] implement policy with a combination of software and hardware.

Previous work has concentrated on developing new protocols, that is, policies. This section focuses, instead, on the mechanisms required to implement these protocols. Describing a directory protocol at the mechanism level exposes disparities in protocol complexity that are not apparent at the policy level. Most protocols, for example, have policy transitions from many readers to one writer and from one writer to another. When examined at the mechanism level, the shared-exclusive transition is clearly harder to implement than the exclusive-exclusive transition. Most systems synthesize the shared-exclusive transition by sending a sequence of invalidate and acknowledgement messages. An implementation must (a) sequence through a large number of message sends, (b) count the acknowledgements, (c) ensure concurrent requests to the same directory entry are serialized, and (d) guarantee that the interaction of these messages with messages for the node's processor, cache, and other directory entries cannot cause network deadlock. On the other hand, for an exclusive-exclusive transition, the directory only sends a single invalidation, which greatly simplifies these considerations. By examining protocols' mechanisms, we can compare the cost and complexity of implementing different protocols and explore the appropriate boundary between hardware and software.

## 3.1 Message-Passing Hardware

All parallel machines provide message-passing mechanisms. Message-passing machines, such as the Intel Paragon, simply expose these mechanisms directly to the programmer. Shared-memory machines, such as Stanford DASH and the Kendell Square KSR1, use these mechanisms to implement shared memory but hide the underlying mechanisms from the programmer. We believe that future shared-memory systems will expose the underlying message passing, as done in MIT Alewife [4]. Some statically-partitionable codes achieve maximum performance through explicit message passing. Agarwal, et al., have demonstrated that other codes achieve better performance with a combination of shared-memory and message-passing than by using one or the other alone [11].

Consequently, we assume base hardware includes support to explicitly send and receive messages. Messages contain a 4-bit message type and are sent to an explicitly-specified destination node $p$. The messages are large enough to contain at least one cache block and an address. The network interface is memory-mapped and resides on the memory bus. A limited DMA capability allows contiguous data to be fetched (stored) directly from (to) memory. When a message arrives at a destination node, it can either wait for an explicit receive operation (i.e., polling) or interrupt the processor and invoke a software trap handler.

The network interface and routers constitute a significant fraction of a parallel machine's design. Our focus in this paper is supporting shared memory without greatly increasing the overall design effort.

## 3.2 General Directory Mechanisms

This section identifies the primary directory mechanisms needed to implement other protocols: $Dir_iB$, $Dir_iNB$, and $Dir_nNB$ (collectively called $Dir_iX$). Rather than formally describing the protocols, we abstract these mechanisms from several recently proposed machines that use these protocols or minor variants of them. Where the published literature lacks details, we made reasonable design choices. We also concentrate exclusively on directory mechanisms since these protocols require identical cache mechanisms.

The $Dir_iX$ directory protocols require numerous additions to the underlying message-passing mechanisms, as Table 1 illustrates. The fundamental change is that some messages, based on the message type, invoke directory operations. The basic directory mechanisms are:

1. Send a single point-to-point message from a directory controller to a processor cache controller.
2. Read/write a pointer field.
3. Increment/decrement/zero a counter.
4. Test for counter equal to zero.

| MsgP | $Dir_i X$ | $Dir_1 SW$ | **Mechanisms** | **Description** |
|---|---|---|---|---|
| | | | Message Receive | |
| • | • | • | poll | Wait for processor to poll for message |
| • | • | • | interrupt | Interrupt processor and invoke software trap handler |
| | • | • | directory | Invoke directory operation |
| | | | Message Send | Send message to processor |
| • | • | • | explicit send | Memory mapped interface for explicit sends |
| | • | • | implicit send | Integrated support for directory controller |
| • | • | • | dest = $p$ | Send message to node $p$ |
| | • | • | dest = PTR | Send message to node in pointer/counter |
| | $i$ (NB) | 1 | max. messages | Maximum messages sent in response to single message |
| | $n$ (NB) | | | |
| | | | Update State | Update directory state field with new value |
| | • | • | new_state | New state value |
| | | | Update PTR/CTR | Update directory entry pointer/counter |
| | • | • | op = Incr | Increment counter by one |
| | • | • | op = Decr | Decrement counter by one |
| | • | • | op = Zero | Reset counter to zero |
| | • | • | op = Set | Set pointer to source node id |
| | | | Test PTR/CTR | |
| | • | • | CTR = 0 | Test if counter equals 0 |
| | • | | PTR = $p$ | Test if pointer points to node $p$ |
| | • | | PTR is valid | Test if field contains a valid pointer |
| | | | Sequence through PTR/CTR | Sequence through pointer/counter field |
| | • | | test | Test pointer/counter field |
| | • | | update | Update pointer/counter field |
| | • | | send | Send message to node in pointer field |
| | $i < n$ (NB) | | Select victim for replacement | Select pointer field to be invalidated |

Table 1: Mechanisms Summary

This table summarizes the mechanisms need for underlying message-passing hardware (MsgP), the general directory protocols $Dir_i X$, and our protocol $Dir_1 SW$. The parameters for each mechanism are listed below it. A • in the appropriate column indicates when a particular protocol requires a mechanism. A directory is invoked in response to a message originating at a processor cache controller (possibly the local one).

When $Dir_i X$ protocols send invalidation messages, they must keep track of acknowledgements in order to maintain sequential consistency (or weaker models). Although a counter is not strictly required (one could invalidate a pointer at each acknowledgement and test for no valid pointers), a counter is far easier to implement.

In general, $Dir_i X$ protocols also need the following mechanisms:

1. Identify valid pointer fields.
2. Compare pointer fields against a node ID.
3. Sequence through the pointers.

$Dir_i NB$ protocols, $i < n$, use a replacement policy to select a victim when the $i + 1^{st}$ shared copy is requested. This policy, in turn, requires an additional mechanism.

The mechanisms for $Dir_n NB$ protocols are slightly different because they can employ bit vectors instead of explicit pointers.

1. Decode node ID and test/set/clear bit in vector.
2. Sequence through bit vector.

All $Dir_i X$ protocols for $i > 1$ require the ability to sequence through either a set of pointers or a bit vector and send multiple invalidations.

### 3.3 $Dir_1 SW$ Mechanisms

The $Dir_1 SW$ column of Table 1 lists the subset of directory mechanisms required by $Dir_1 SW$. $Dir_1 SW$ requires mechanisms to update state, send a single message, and test and update a single pointer/counter field. However, because $Dir_1 SW$ has only a single pointer/counter field, it does not need the sequencing logic used by $Dir_i X$ ($i > 1$). Similarly, $Dir_1 SW$ sends at most one message in response to an incoming request; protocol transitions requiring multiple messages are handled by software.

### 3.4 Design Cost

In our view, the ultimate measure of directory protocol complexity is *design cost*—how long a protocol takes to implement. Unfortunately, differences in design teams, tools, and project goals prevent any concrete comparison of design cost.

For this reason, this section considers indirect measures of design cost that arise from sequencing directory mechanisms. A key goal of $Dir_1SW$ was to reduce the cost and complexity of shared-memory hardware by using a protocol where the most frequent policy transitions can be implemented with simple, short sequences of mechanisms (e.g., a single invalidate message). More complex sequencing—involving many messages—is done by system software (trap handlers). Avoiding complex hardware sequencing eliminates the complexity that arises from transient states, ensuring new policy requests are serialized, and avoiding network deadlock.

One indirect measure of protocol complexity that has some value is the number of state/event pairs that must be handled in hardware, where events can be messages or processor actions (e.g., loads and stores). This measure is useful, because it quantifies the number of cases that the designer must consider and test for correctness. By this measure, $Dir_1SW$ is fundamentally simpler than any of the $Dir_iX$ protocols (with the exception of $Dir_1NB$) because it does not require sequencing, sends at most one message in response to any message, and requires only a simple datapath. Since much of this simplicity comes from pushing the complexity into software trap handlers, other hardware/software protocols such as LimitLESS, share this advantage.

All $Dir_iX$ protocols for $i > 1$ require the ability to sequence through either a set of pointers or a bit vector and send multiple invalidations. To implement this mechanism as an atomic sequence, all invalidations must be sent before receiving any other messages. Unfortunately, deadlock avoidance then becomes a major consideration. If the maximum number of messages is bounded by a small constant, as in $Dir_iNB$, deadlock can be avoided with sufficient output buffering. The directory controller simply waits until its output FIFO has room for $i$ messages before sending the first. However, this is not a scalable solution[1] for protocols that may send large numbers of messages, such as $Dir_iB$ and $Dir_nNB$, since the maximum number of messages is proportional to system size.

[1] This solution can be used for any system with a *fixed* maximum size, provided each node has output buffering at least as large as this size.

The alternative is to make this mechanism non-atomic, and process incoming messages between sends. This facilitates deadlock avoidance, however, the sequencer's state becomes an additional, transient part of the cache block's state, greatly increasing the number of state/message interactions. In addition, multiple cache blocks may need to be sequenced simultaneously (in order to avoid deadlock), requiring some form of preemptive scheduling. Although this complexity can be managed, architects must expend considerable effort designing, building, and testing complex hardware rather than improving the performance of simpler hardware.

### 3.5 Manufacturing Cost

Comparing the manufacturing cost of mechanisms is relatively straight-forward. *Manufacturing cost* is ultimately measured in dollars, but is commonly estimated with other measures such as transistor count, bits of memory, datapath width, etc. For directory protocols, the dominant cost is memory overhead: number of bits of state stored per block of memory. All protocols need a small number of bits (e.g., 3 or 4) to represent the block's state. The $Dir_iX$ protocols other than $Dir_nNB$ require $i$ pointers of $log_2n$ bits each; $Dir_nNB$ protocols require $n$ bits. By contrast, $Dir_1SW$ requires only one $log_2n$-bit pointer/counter field. Consider a system that supports up to 1024 nodes and has 32-byte cache blocks. If we assume 4 bits can describe the state of each block, then $Dir_4NB$ incurs a 16% memory overhead (44 bits/256 bits), $Dir_nNB$ incurs a 402% overhead, while $Dir_1SW$ incurs only 5% overhead.

After memory, the next greatest cost is the directory datapath. For the $Dir_iX$ protocols other than $Dir_nNB$, the comparison of a node ID with $i$ pointer fields requires either a wide datapath with $i$ comparators or a sequential search. A $Dir_nNB$ implementation will require an $n$-bit datapath and priority decoder. By contrast, the $Dir_1SW$ state machine requires only a $log_2n$-bit datapath with the ability to increment, decrement, test for zero, and select the ALU result, the message source ID, or a small constant for writing into the pointer/counter.

The absence of sequencing in the $Dir_1SW$ mechanisms also allows a regular structure: in response to each message, the state associated with the cache block is read, modified, and written back, and optionally a single message is sent. Beyond its inherent simplicity, this regularity leads naturally to a pipelined implementation with increased throughput. While other schemes can also be pipelined, as for example, in the Stanford DASH [14], the increased datapath complexity requires additional designer time that could otherwise be spent elsewhere.

## 3.6 Improvements to $Dir_1SW$

The base $Dir_1SW$ protocol described above performs as well as any feasible directory coherence protocol for programs that exactly follow the CICO programming model (see Section 2). However, rigidly adhering to this model is not possible or desirable for all programs. This section examines several extensions to the $Dir_1SW$ protocol that improve its performance for programs that do not conform precisely to CICO. With one exception, these extensions use *exactly* the same mechanisms as base $Dir_1SW$ and require minor changes to the policy implemented in hardware and software. The new mechanism, which is very simple, sets the counter in a directory entry to the value 1.

### 3.6.1 $Dir_1SW$+NPT: No Pairwise Traps

The base $Dir_1SW$ protocol traps to software whenever a CICO violation occurs; that is, whenever the directory receiving a `Msg_Get` message cannot immediately respond with the requested data. However, the $Dir_1SW$ mechanisms permit directory hardware to send a single message to an arbitrary processor in response to a message from another processor. The NPT extension modifies the hardware policy to directly send an invalidation message and forward the block to the requesting processor when the `Msg_Put` message arrives. This extension moves a common, but more complex policy from software to hardware, which may reduce execution time.

### 3.6.2 $Dir_1SW$+RO1: One Shared Copy

The $Dir_1SW$ mechanisms permit a protocol to maintain either a pointer to a processor node or a counter. The base $Dir_1SW$ protocol maintains a pointer for exclusive copies and a count of shared copies. However, many of the shared-to-exclusive state transitions occur when only a single shared copy is outstanding: over 50% for 6 of the 8 applications, and over 85% for 4 of them. An obvious extension of $Dir_1SW$ is to add a new state `Dir_S_One` that maintains a pointer to a single shared copy. The benefit of this change is that it reduces the number of traps that broadcast an invalidate to all processors.

### 3.6.3 $Dir_1SW$+RO1+NPT: $Dir_1SW^+$

This extension, called $Dir_1SW^+$, combines the changes from Section 3.6.2 and Section 3.6.1. In this protocol, the trap handler is only invoked to broadcast an invalidation for a block that was shared by more than one processor.

| 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 2: Application Programs

This table describes 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 [16].

# 4 Directory Performance

This sections presents our experimental methods, compares the performance of $Dir_1SW$ variants, and compares the best $Dir_1SW$ variant with alternative protocols.

## 4.1 Methods

The measurements in this paper came from the eight explicitly parallel programs listed in Table 2 running on the Wisconsin Wind Tunnel (WWT), a virtual prototype for cache-coherent, shared-memory computers [15]. WWT runs parallel shared-memory programs on a parallel message-passing computer (a Thinking Machines CM-5) and uses a distributed, discrete-event simulation to concurrently calculate the programs' execution times on a proposed target machine.

The simulated parallel computer (the target system) used in this paper consists of 32 processor nodes, each containing a processor, shared-memory module, cache, and network interface. Processors execute SPARC binaries. The execution time for each instruction is fixed. Instruction fetches and stack references require no additional cycles beyond the basic instruction time. Other memory locations are cached in a node's cache. A cache hit takes no additional cycles, while a cache miss invokes a coherence protocol that sends messages, accesses a directory entry, etc. Each message, cache or directory transition has a cost. Caches and directories process messages in first-

| | |
|---|---|
| Cache | 256 KB, 4-way set-associative |
| Block size | 32 bytes |
| TLB | 64 entries, fully associative, FIFO replacement |
| Page size | 4 KB |
| Message latency | 100 cycles remote, 10 cycles to self |
| Barrier latency | 100 cycles from last arrival |
| Cache miss | 19 cycles + 5 if block is replaced + 8 if replaced block was exclusive copy |
| Cache invalidate | 3 cycles + 5 if block is invalidated + 8 if invalidated block was exclusive copy |
| `check_out` | Same as cache miss, plus 1 cycle for `check_out` issue |
| `check_in` | Same as cache invalidate, plus 1 cycle for `check_in` issue |
| Directory | 10 cycles + 8 if cache block is received + 5 if message is sent + 8 if cache block is sent |
| Trap | 255 cycles + 5 for each message sent + 8 for each block sent (directory hardware locked out for first 55 cycles) |

Table 3: Baseline System Assumptions

come-first-serve order. Queuing delay is included in the cost of a cache miss. Network topology and contention are ignored, and all messages are assumed a fixed latency. Table 3 lists the basic system parameter values.

| Factor | Var. Due (%) | Mean Effect (%) | 90% Confidence Interval |
|---|---|---|---|
| Benchmarks | 67.45 | n/a | n/a |
| NPT | 24.29 | -15.86 | [-21.77, -9.96] |
| RO1 | 0.14 | -1.19 | [ -7.10, 4.71] |
| NPT+RO1 | 0.04 | -0.68 | [ -6.59, 5.22] |
| /CICO | 6.05 | -7.92 | [-13.82, -2.02] |
| NPT/CICO | 1.94 | 4.48 | [ -1.42, 10.38] |
| RO1/CICO | 0.08 | 0.90 | [ -5.01, 6.80] |
| NPT+RO1/CICO | 0.00 | 0.01 | [ -5.89, 5.92] |

Table 4: Analysis of Variance of $Dir_1SW$ Extensions

These numbers were collected from a full factorial experiment using the benchmarks for replication and NPT, RO1, and CICO as factors [10, Chapter 18]. Column "Variation Due" lists the percent of performance variation in the 64 runs ($8 * 2^3$) caused by benchmark, factors, and interactions between factors. The results show that the benchmark, NPT, and CICO are the most important factors. Column "Mean Effect" gives the relative change in normalized execution time caused by factors and interaction terms. NPT and CICO reduce running time the most. (The benchmark row is marked "n/a" because a replication factor is assumed to have no systematic effect.) The final column gives the 90% confidence intervals for each factor and interaction term. The range of all intervals extends up and down from the mean effect by $t_{[0.95;2^3*(8-1)]} * SSE/(2^3 * 8 * 2^3 * (8-1))$, where $SSE$ is the sum of the squares of the residuals (errors)—the difference between an actual value and the corresponding prediction using the mean effect. Use of the t-distribution is meaningful if these differences are distributed normally with zero mean. A normal quantile-quantile plot of the 64 residuals (not shown) reveals that this assumption is approximately true.

## 4.2 $Dir_1SW$ Variants

This section discusses the performance of $Dir_1SW$ variants on 32 processors for the benchmarks in Table 2. The variants are: no-pairwise-traps (NPT), one-shared-copy (RO1), and CICO directives. We examine all eight combinations ($2^3$). Figure 2 displays execution times for seven cases, normalized to the execution time of the base case (without NPT, RO1, and CICO). Since the normalized execution times are less than 1.0, the extensions all improve performance relative to base $Dir_1SW$. However, the extensions affect the benchmarks by varying amounts. They matter little to *water* and *tomcatv*, for example, because both perform little communication relative to their computation. (The same effect is apparent for the other protocols compared in Figure 3 of Section 4.3.) For *mp3d*, however, NPT and CICO reduce execution time by 52% and 21%, respectively, by mitigating the effect of this program's unsynchronized sharing in its `cell` data structure.

To get more insight from the many numbers in Figure 2, we use an analysis of variance to characterize mean behavior. This aggregation is meaningful only if the eight benchmarks are representative of some interesting workload. Table 4 reports results and the table's caption describes the analysis of variance method in more detail. The results show that most of the variation between runs is caused by the benchmarks themselves. Nevertheless, NPT and CICO caused statistically significant variation. The mean relative improvement from NPT was 16%, while CICO yielded 8%.

Using CICO primitives as memory system directives affects sharing behavior and improves performance. Table 5 examines the effect on sharing behavior of using CICO `check_in`'s to flush cache blocks (rather than allowing them to be replaced or invalidated).[2] A `check_in` improves performance if it enables another processor to find a block at the directory instead of requiring additional messages be sent

[2] We also examined `check_out`'s but found their effect to be small.

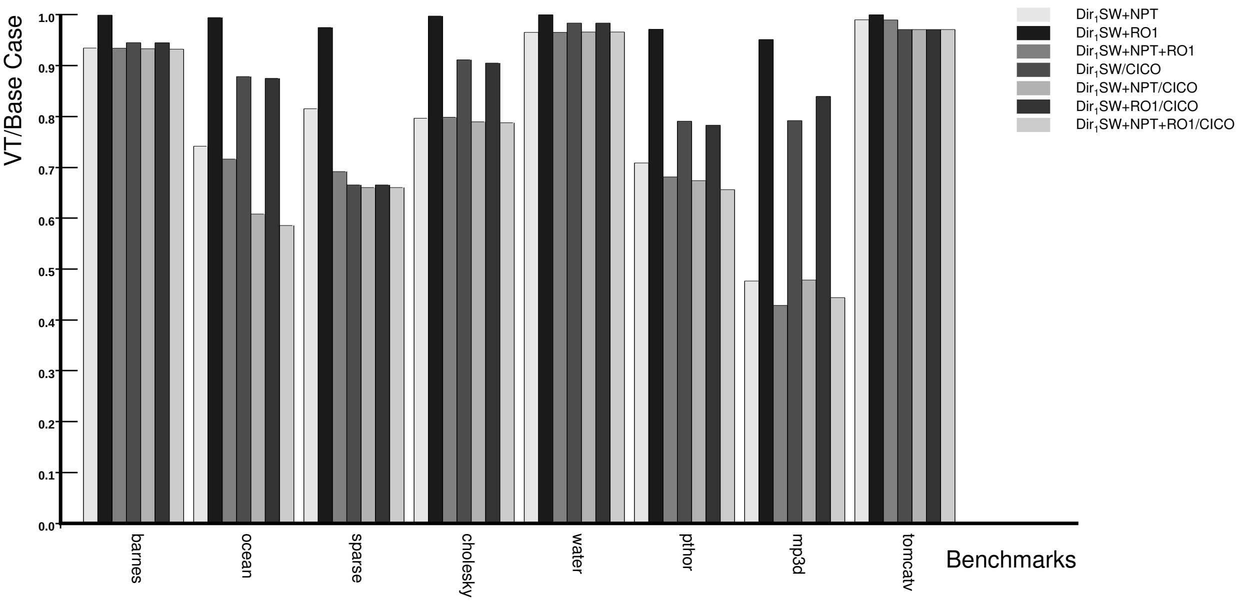


Figure 2: Performance of $Dir_1SW$ Extensions

This figure shows the time to run the benchmarks with and without no-pairwise-traps (NPT), one-shared-copy (RO1), and CICO, relative to the time to run the same program under base $Dir_1SW$ without CICO.

| **Benchmark** | **Indirections Avoided (%)** | **Counter Productive check-in's (%)** |
|---|---|---|
| *barnes* | 84 | 52 |
| *ocean* | 45 | 14 |
| *sparse* | 99 | 51 |
| *pthor* | 61 | 61 |
| *cholesky* | 94 | 27 |
| *water* | 97 | 6 |
| *mp3d* | 74 | 58 |
| *tomcatv* | 100 | 65 |

Table 5: CICO Effects

This table displays the effect on each benchmark's sharing behavior of using `check_in`'s to flush cache blocks (rather than allowing them to be replaced or invalidated). Column "Indirections Avoided" shows the relative reduction in the frequency of indirections. An indirection occurs when a processor cannot obtain a block from the directory, but must send messages to one or more processors. Column "Counter-productive `check_in`'s" gives the fraction of `check_in` for which the same processor is the next user of a checked-in block.

to other processors. The results show that `check_in` reduces the frequency of indirections by 45%–100%. A `check_in` hurts performance if the same processor is the next user of the block, which we found to occur in 6%–65% of the `check_in`'x.

Together, NPT and CICO ran programs 19% faster, implying NPT makes CICO less important. With NPT, CICO has a more modest impact on indirections to previously exclusive blocks (e.g., migratory data). Without NPT or CICO, migrating a block costs four network traversals and two traps. Adding NPT or CICO eliminates the traps, while CICO also reduces the network traversals to two. Thus, at best, adding CICO to NPT improves performance by a factor of two. In practice, the effect is much smaller, because programs do not spend much time migrating data.

Finally, we would like to estimate how the effects of NPT and CICO vary from benchmark to benchmark. To do this, we calculate 90% confidence intervals assuming the residuals—the performance not explained by average effects—are normally distributed with mean zero. This calculation—explained further in the caption of Table 4—reveals $[-22\%, -10\%]$ for NPT and $[-14\%, -2\%]$ for CICO. With eight benchmarks and not-exactly-normally distributed residuals, our confidence intervals are best taken with a grain of salt.

In summary, NPT and CICO improve performance of almost all programs, while RO1 helps a little. Since

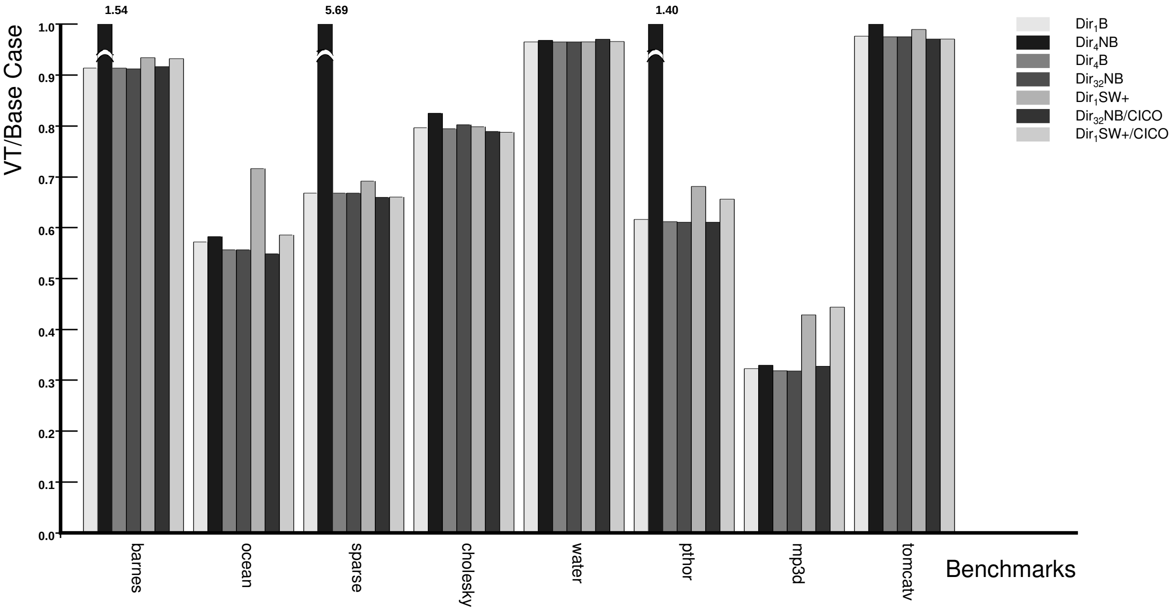


Figure 3: Performance of Other Protocols

This figure displays the normalized execution times (with respect to base $Dir_1SW$ without CICO) for several protocols. Three runs of $Dir_4NB$ are broken bars, because their normalized execution was much longer than $Dir_1SW$'s.

NPT and RO1 use the same mechanisms as the base protocol, we incorporated them in a new protocol called $Dir_1SW^+$.

### 4.3 Comparison to Other Protocols

This section compares $Dir_1SW^+$ without and with CICO, denoted $Dir_1SW^+$ and $Dir_1SW^+$/CICO, against several other protocols. Figure 3 displays normalized execution time for the eight benchmarks running on 32 processors under several protocols. The principal result is that $Dir_1SW^+$ and $Dir_1SW^+$/CICO perform comparably to $Dir_{32}NB$—well within 10%—even when $Dir_{32}NB$ uses CICO, except for *mp3d* with its unscalable, unsynchronized sharing. The data also show that $Dir_4NB$ is an unstable protocol, at least, when no special mechanism handles read-only data.

These conclusions do not seem to be sensitive to the key system parameters of network and directory latency. (We also measured runs with 64 processors, but do not report these results because they did not differ qualitatively.) Table 6 shows the normalized execution time results from varying interconnection network latency from 100 processor cycles (the default) to 400 cycles. A 400-cycle network slowed all protocols by about a factor of two, but it has little effect on the performance difference between $Dir_{32}NB$ and $Dir_1SW^+$.

Increasing the latency of a directory operation to 100 cycles approximates the effect of using an auxiliary processor, rather than a finite state machine, to perform directory operations. Increasing the directory cost from 10 to 100 cycles slowed the benchmarks by an average of 40% with no obvious trends favoring one protocol over another. Finally, we looked at performance with larger values for both network and directory latency. With the slower network, increasing directory latency only decreased performance slightly (15%).

## 5 Discussion

While quantitative results are useful, it is important to step back and look at what they mean. The data shows that:

- Memory system directives, such as CICO, can alter program behavior to make simple directory hardware more attractive.
- Elucidating the mechanisms underlying a coherence protocol can lead to new protocols that per-

| Directory Access Cost | Network Latency | $Dir_{32}NB$ Mean | $Dir_{32}NB$ Deviation | $Dir_1SW^+$ Mean | $Dir_1SW^+$ Deviation | $Dir_1SW^+$/CICO Mean | $Dir_1SW^+$/CICO Deviation |
|---|---|---|---|---|---|---|---|
| 10 | 100 | 0.7261 | 0.2155 | 0.7756 | 0.1759 | 0.7503 | 0.1829 |
| 10 | 400 | 1.3285 | 0.3286 | 1.3777 | 0.3210 | 1.3348 | 0.2914 |
| 100 | 100 | 1.0096 | 0.2035 | 1.0803 | 0.1535 | 1.0332 | 0.1452 |
| 100 | 400 | 1.5403 | 0.4142 | 1.5857 | 0.4076 | 1.5179 | 0.3636 |

Table 6: Varying System Assumptions

This table displays the arithmetic mean and standard deviation of the normalized execution times (with respect to base $Dir_1SW$ without CICO) for eight benchmarks under several protocols with different assumptions of directory cost and network latency. The arithmetic mean listed for each specific system is proportional to the execution time of the eight benchmarks on that system, provided each benchmarks ran for the same amount of time under the base case.

form better without significantly increasing implementation complexity.

- For the system assumptions and benchmarks, most protocols performed similarly. The significant disparity in hardware complexity and the small difference in performance argue that $Dir_1SW^+$ may be a more effective use of resources.

Although our results have immediate import, they also apply to future computers. These machines are moving toward large-scale ($\geq 1K$-processors) systems of fast microprocessors ($\geq$ 1 GIPS). The network latencies of these machines (measured in processor cycles) will be much larger than today's machines. The data in Table 6 for 400 cycle network latency shows that larger networks do not affect $Dir_1SW$ more than other protocols such as $Dir_nNB$ (assuming that programs infrequently cause broadcasts).

A perhaps more important implication of the data is that performance in machines with long network latencies is not sensitive to directory latency. This suggests that moving protocol sequencing to software running on a node's main processor, an auxiliary processor (as in the Intel Paragon), or a processor in the network interface may be practical [1]. The obvious drawback of this approach is that a processor sequences a protocol slower than a hardware finite state machine. A secondary drawback is that slower directories increase directory contention. The data shows that increasing directory latency from 10 to 100 cycles degrades execution time by 15%. This degradation can be mitigated or reversed by reducing directory contention (e.g., with greater interleaving) and by using protocols that send fewer messages.

On the other hand, software sequencing offers many advantages and opportunities:

- System design time can be reduced because less hardware must be designed. In addition, field-upgrades of protocols are possible. Thus, the design time and hardware for shared-memory machines could be similar to message-passing computers.

- Protocols can adapt to dynamic program behavior since buffering and analyzing recent behavior is practical in software.

- Protocols can be statically tailored by compilers, program libraries, or application programs to behave differently for different objects [3]. For example, update protocols could distribute widely-used data (e.g., the vector $x$ in $x := Ax + b$) and help in synchronization (a barrier wakeup) [6].

- Protocols can support higher-level operations such as fetching an entire row of a matrix or a scatter-gather operator.

- Collecting information for performance monitoring is much easier.

Regrettably, we leave evaluation of these ideas to future work. Our benchmarks were written for small scale systems. Running these programs on more than 32 or 64 processors exposes bottlenecks and yields poor speedup. We plan to use the CICO programming model [12] to construct programs that manage communication more effectively and use these programs to evaluate these ideas.

# 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. The key to effective, scalable, shared-memory parallel computers is to address the software and hardware issues together.

This paper explored the complexity of implementing directory protocols by examining their *mechanisms*—primitive operations on directories, caches, and network interfaces. We compare the following protocols: $Dir_1B$, $Dir_4B$, $Dir_4NB$, $Dir_nNB$ [2], $Dir_1SW$ [9] and an improved version of $Dir_1SW$ ($Dir_1SW^+$). The comparison shows that the mechanisms and mechanism sequencing of $Dir_1SW$ and $Dir_1SW^+$ are simpler than those for other protocols. Simulation results for eight benchmarks on 32-processor systems show that $Dir_1SW^+$'s performance is comparable to more complex directory protocols. The small performance difference between $Dir_1SW^+$ and the more complex protocols is attributable to two factors: the small degree of sharing in programs and CICO directives. The significant disparity in hardware complexity and the small difference in performance argue that $Dir_1SW^+$ may be a more effective use of resources.

As network latencies increase, the performance effect of directory operation overhead decreases, which provides the opportunity to sequence directory operations in a processor rather than a state machine. This change, in turn, permits high-level directory operations that have the potential to hide more of the increased communication cost. Evaluating these alternatives for kiloprocessor systems will require new benchmarks and an evaluation platform that simulate more processors than current machines contain.

# 7 Acknowledgements

Dave Douglas, Danny Hillis, Roger Lee, and Steve Swartz of Thinking Machines provided invaluable advice and assistance in building the Wisconsin Wind Tunnel. Glen Ecklund and Alain Kägi helped develop the Wisconsin Wind Tunnel and applications. Singh et al. [16] wrote and distributed the SPLASH benchmarks.

# References

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

[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] John K. Bennett, John B. Carter, and Willy Zwanepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 168–176, February 1990.

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, 1991.

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

[12] James R. Larus, Satish Chandra, and David A Wood. CICO: A Shared-Memory Programming Performance Model. Submitted for publication., January 1993.

[13] 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.

[14] Daniel E. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor.* PhD thesis, Stanford University, February 1992. CSL-TR-92-507.

[15] 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.

[16] 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.

[17] 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.