Absolute and Comparative Performance of Cache Consistency Algorithms

Jeffrey D. Gee
Alan Jay Smith

Department of Electrical Engineering and Computer Science
Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

As more computer systems turn to multiprocessing for improved performance, additional research is needed to evaluate and improve the performance of cache consistency protocols. In this study, we use trace-driven simulation to examine the performance of several consistency protocols, including some new adaptive protocols which have not been examined in prior research. This study uses a wider variety of traces than have been previously analyzed, including some production applications from a vector mini-supercomputer system, and presents a wider variety of analyses than have been previously presented for a given workload.

We find that the sharing characteristics of application programs have a large bearing on the relative performance of the different protocols. *Update-based* protocols outperform *invalidate-based* protocols when accesses to shared data are highly interleaved among different processors (fine-grain sharing), while invalidate-based protocols are superior if one processor performs all accesses to shared data over long periods of time (coarse-grain sharing). Adaptive protocols provide the best overall performance across all applications; we present a new protocol called Update-Once, which yields the highest average performance. In even the best cases, however, estimated processor utilizations are unacceptably low due to the overhead to maintain consistent caches. To extract good performance from multiprocessor systems, existing application programs must be recoded to reduce sharing between processors.

June 2, 1993

*The authors' research is or has been supported in part by the National Science Foundation under grants MIP-8713274, MIP-9116578 and CCR-9117028, by NASA under Grant NCC 2-550, by the State of California under the MICRO program, and by Mitsubishi Electric Research Laboratories, Sun Microsystems, Philips Laboratories/Signetics, Intel Corporation, International Business Machines Corporation, and Digital Equipment Corporation.

1. Introduction

This study uses trace-driven simulation to evaluate and compare the performance of a large selection of cache consistency protocols. Our purpose is to determine, given the workloads available to us, (a) which existing protocols perform well on these workloads, (b) which protocol features most improve performance, and (c) what new features can be added to further improve existing protocols. This study goes beyond previous research, as is explained below, by using a larger and more varied trace set, including production applications from a mini-vector-supercomputer system, and by presenting a wider variety of results than have been available previously for a given group of traces.

Cache consistency protocols generally fall into one of two categories: (1) *snooping-cache* protocols [Arch88, Good83, Karl86, Katz85, McCr84, Papa84, Sega84, Stew87, Thac87], which broadcast transactions affecting the consistency of shared data to all other processors via a shared bus, and (2) *directory-based* protocols [Arch85, Broo90, Cens78, Chai91, Gupt90, Leno90, Tang76, Okra90], which use point-to-point message passing mechanisms across arbitrary interconnection networks to maintain consistency. Systems based on snooping-cache protocols can only contain a limited number of processors, since the shared bus quickly becomes saturated. Directory-based protocols circumvent this problem by using an interconnection network of sufficient bandwidth. Typically in directory protocols, messages are sent to only those processors with copies of the data; that information is kept in a shared directory.

Snooping and directory-based protocols also differ in the manner by which state is maintained. In snooping protocols, protocol state is distributed among the individual processor caches, whereas for directory protocols some or most state is located in a centralized directory memory. Both types of protocols, however, are free to use the same set of cache states and state transitions (i.e. algorithms) to manage data. We simulate protocols at the algorithmic level, and later attribute appropriate costs to various transactions to evaluate performance in both directory-based and snooping-based environments.

In the remainder of this paper, we present results from simulating a wide selection of protocols using a large number of multiprocessor address traces drawn from real workloads. Efficient stack algorithms [Thom87] were used throughout to evaluate a much larger design space than was possible in prior studies. Our results are compared with earlier work which analyzed the reference and sharing behavior within these same multiprocessor traces [Gee93], and with work which analyzed uniprocessor versions of some of these same applications [Gee92]. This remainder of this paper is organized as follows. Section 2 summarizes previous work in this field. Section 3 describes the address traces and the multiprocessor cache simulator. Section 4

describes the cache consistency protocols that we simulated. The body of our work is contained in Section 5, which discusses and analyzes results from our simulations. Section 6 summarizes and concludes the paper. The results in this paper have been edited to present only the essentials; additional material also appears in [Gee93b].

2. Background

2.1. Previous Work

A number of studies have previously addressed the performance of cache-consistency protocols; in this section we discuss the better known of those studies. One early effort [Arch86] used synthetic multiprocessor traces to measure processor utilization for the snooping-based Write-Once, Synapse, Berkeley, Illinois, Firefly, and Dragon protocols. That study found that the update-based Dragon and Firefly protocols outperformed the invalidate-based Berkeley, Illinois, and Write-Once protocols. Their results should be interpreted with some caution, however, as the nature of the model used to synthesize the traces led to many aspects of the results.

Later studies have utilized real address traces gathered from executing parallel applications on real multiprocessors. Most recently, Vashaw [Vash93] used a hardware monitor to record large trace samples containing both supervisor and user accesses on an 8-processor Encore Multimax. These traces were used to quantify some aspects of data sharing and cache performance, although a thorough examination of consistency protocol performance was beyond the scope of that research.

Agarwal and Gupta [Agar88a] provided one of the first studies of protocol performance using address traces. The traces were of CAD applications running on a 4-processor VAX 8350, and were used to measure snooping bus transactions made by (1) a Write-Through Invalidate protocol, (2) the Write-Once invalidate-based protocol, and (3) the Dragon update-based protocol. Write-Through with Invalidate was the lowest performing of the three protocols, since each write bypasses the local cache and proceeds onto the heavily utilized global bus and main memory. Some of the results in that paper showed the presence of *false sharing*, which occurs when several processors reference separate data items resident in the same cache block.

Three studies by Eggers and Katz [Egge88, Egge89a, Egge89b] also used trace-driven simulation to examine snooping-cache protocol performance. In the first of these studies, four multiprocessor CAD traces taken from Sequent and ELXSI machines were used to simulate the SPUR [Hill86] multiprocessor. Bus transactions and bus cycles for the Berkeley and Firefly protocols were measured, with Berkeley performing better on two traces and Firefly performing better on the other two.

The second study [Egge89a] used the same traces to measure miss ratios and bus utilization for write-invalidate protocols. Miss ratios did not decrease very rapidly with increasing cache size, since invalidations prevent large caches from being fully utilized. Miss ratios did *increase* with increasing block size in some instances, due to the effects of false sharing. Overall, miss ratios and bus utilization were found to be far higher than in uniprocessor systems due to the overhead to maintain cache consistency.

The last study [Egge89b] evaluated extensions to improve the performance of invalidate and update-based snooping protocols. The *read-broadcast* extension [Sega84] allows any cache to receive fresh data from the bus if that cache currently has an invalid copy of the data. The *competitive snooping* extension [Karl86] enables a protocol to switch from updating data to invalidating data after some number of updates (two in this case) have been performed without the data being used by another processor. This puts an upper limit on consistency overhead when processors are no longer using shared data in their caches. This study found that neither of the two extensions improved performance significantly.

In addition to snooping protocols, directory-based protocols have also been extensively studied in the literature. Tang [Tang76] proposed a scheme where the central directory is essentially a copy of all the individual cache tags. Censier and Feautrier [Cens78] described a more compact central directory containing one presence bit for each individual cache and a single dirty bit if the data is modified (in which case only one of the presence bits would be set). Directory storage can be reduced still further if only a subset of all presence bits, or no presence bits at all, are maintained in the central directory [Arch85, Agar88b].

In terms of performance, [Agar88b] found that directory-based schemes are competitive with snooping-based schemes in terms of average cycles per transaction, and concluded that directories should be attractive in systems too large to implement snooping protocols. Research has also found that schemes with a limited number of presence bits can perform nearly as well as fully-mapped directories [Broo90, Okra90].

2.2. Trace Characterization

In [Gee93a], we characterized in some detail the sharing behavior within the same multiprocessor address traces used in this work. The traces, described briefly in the following section, and in far more detail in [Gee93a], are from three different workloads: (1) 4-processor vector scientific traces that we collected on Ardent Titan multiprocessors, (2) 16-processor scalar CAD traces gathered at Stanford on a VAX multiprocessor using trap-bit (T-bit) tracing, and (3) 64-processor scalar scientific traces used at MIT and collected from IBM and Encore multiprocessors.

Our earlier study analyzed the processor locality, temporal locality, and contention within these traces over a range of block sizes, and found that the amount and type of sharing present in the traces depend heavily on workload. A much larger fraction of Ardent references were found to be to shared data, relative to the MIT and Stanford workloads. In addition, *processor locality*, the sharing behavior where only one processor uses shared data over long periods of time, is very strong in the Ardent vector workload but much less evident in the two scalar workloads.

Figure 1 (from [Gee93a]) illustrates the processor locality within a set of applications. The figure shows average *write run* lengths for three trace workloads across block sizes ranging from 8 to 64 bytes. A *write run* [Egge88] is a string of successive writes to a data block by one processor, which may also contain reads to that block by that same processor but no references by other processors. The figure clearly shows that the Ardent workload contains more processor locality than the other two workloads for large block sizes.

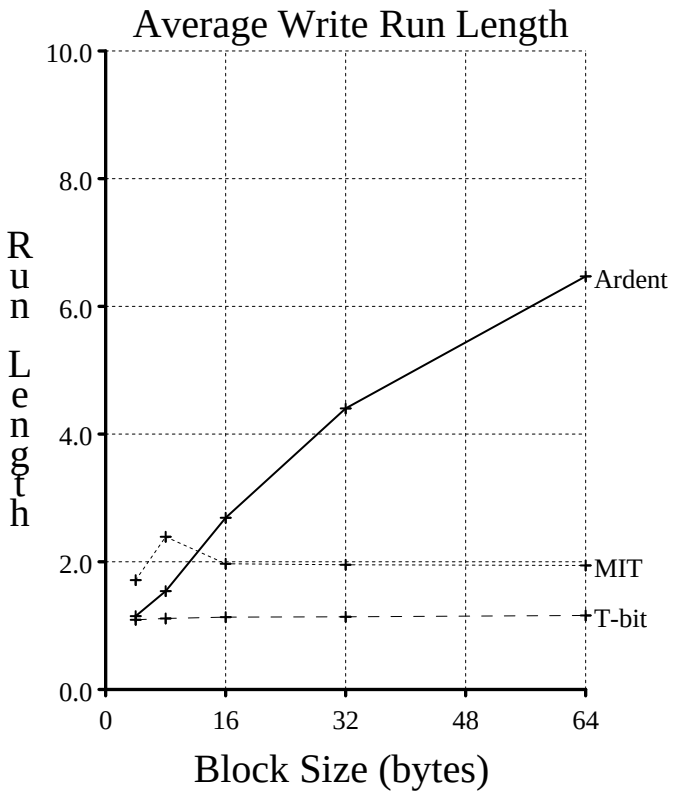

Figure 1: Average write run lengths

In [Gee93a] we analyzed the temporal characteristics of shared data references. Consecutive references to shared data by the same processor tend to occur at short time intervals, while time intervals between references to data by different processors are somewhat longer. For some applications, a data write followed by a read from a different processor is serialized by locks. We also encountered applications which do not serialize such accesses to shared data. Sequentially

consistent protocols, which require that all processors have a consistent view of memory, will remain necessary for this latter class of application.

Based on the reference characteristics that we observed, we had speculated that adaptive protocols may provide consistently good performance across the wide range of sharing behavior observed in our traces. We proposed a protocol which updates data once, and invalidates the data on the next write if the first update had not been used by other processors. The idea behind this protocol stems from Figure 1, as write run lengths tend to fall into one of two extremes: short in the scalar workloads, or very long in the vector workload with large block sizes. The adaptive protocol supports short write runs by initially updating data, and supports long write runs by invalidating data if earlier updates are not used by other processors.

3. Methodology

3.1. The Multiprocessor Cache Simulator

We use efficient simulation algorithms [Thom87] which enable the evaluation of more protocols over a larger design space than was possible in previous studies; the Thomson work extends stack algorithms [Matt70] to multiprocessor system simulations. Using these algorithms, we can measure miss ratios and bus transactions for all cache sizes in one pass through an address trace. The key to these algorithms is the *principle of inclusion*. Blocks that are *valid*, *dirty*, or *shared* in one cache size are also valid, dirty, or shared in any larger cache size.

The simulator operates by maintaining two important data structures. The first is a set of LRU stacks representing the current contents of each processor cache. The position of a cache block in an LRU stack determines the smallest cache size in which a cache hit occurs. The second data structure is a global directory which contains *sharing* and *dirty* levels for every block resident in a processor cache. The *sharing level* represents the smallest cache size where a block is *thought* to be shared and must be updated or invalidated on a data write. The *dirty level* represents the smallest cache size where a block is dirty. At cache sizes greater than the dirty level, a block request must be satisfied by a cache rather than by main memory.

The algorithms which maintain data structures and count events are particularly complex, and we refer the reader to [Thom87] for a complete and detailed description. We extended the algorithms to differentiate between block requests serviced by memory and requests serviced by other caches. This distinction is necessary to analyze time performance (i.e. bus cycles and processor utilization), as requests serviced by other caches may complete more quickly than requests serviced by the slower main memory.

3.2. The Multiprocessor Traces

The input to the simulator consists of three sets of multiprocessor address traces identical to those used in our earlier trace characterization effort [Gee93]. The first set of traces were collected from production applications running on 4-processor Ardent Titan vector machines, and were generated by using an object code profiler to instrument compiled programs. As the instrumented object code executes, all four Titan processors deposit reference addresses to a single, shared trace file. This file is protected by locks to allow only one process access to this file at any time, and this locking guarantees a (not "the") valid trace. The Ardent traces are based on a 64-bit memory interface. Data references can be either 32-bit or 64-bit quantities, whereas instructions are always 32-bit quantities.

At Stanford, 16-processor traces were gathered using the trap-bit tracing method on VAX-series computers [Webe89]. Setting the trap bit on a VAX interrupts a process after each instruction, allowing a trap handler to examine the instruction and generate a trace record of its memory references. To generate multiprocessor traces, a *master process* controls the execution of a number of *slave processes*, which represent the execution of individual processors. After a slave process executes an instruction, it traps back to the master which records its memory references, saves the slave process state, and schedules a different slave process to run, usually in a round-robin fashion.

The 64-processor traces *fft64*, *weather64*, and *speech64* were used at MIT to evaluate the performance of various directory-based protocols [Chai90]. The *fft64* and *weather64* traces were generated from IBM uniprocessor traces using a postmortem scheduling technique. Parallel programs are first executed on a uniprocessor to generate traces containing *tasks* (indivisible units of work assigned to a processor) and synchronization information. These traces were then postprocessed into parallel traces by scheduling these tasks on a number of processors. The *speech64* trace was generated using compiler-aided techniques to insert tracing code into the instruction stream. This technique is similar to the method used to generate the Ardent traces, but operates at compile time rather than after link time. The compiler-based scheme executes on an Encore Multimax under a modified Mul-T (a variant of Multilisp) programming environment.

Table 2 lists the fraction of instruction, synchronization, private-read, private-write, shared-read, and shared-write references in each set of traces (results for individual traces are in the attached Appendix and in [Gee93b]). Like [Agar88a], we consider a reference to be a *shared* reference if it accesses data used by more than one processor during the trace. In contrast, [Egge88] considers references to be shared references if they access memory addresses statically allocated to shared data, and [Egge88] identifies shared references by postprocessing the traces using information from object code header files. Each memory reference in our traces, whether a 4-byte single-precision or an 8-byte double-precision quantity, is counted once in Table 2.

Since our simulations cover block sizes of eight bytes and larger, no special processing is required for double-precision references.

Application Summary			
Program	Machine	Language	Description
arc3d	Ardent Titan	Fortran	3D fluid dynamics
bmk1	Ardent Titan	Fortran	monte carlo simulation
bmk11a	Ardent Titan	Fortran	particle in a cell
flo82	Ardent Titan	Fortran	transonic flow past airfoil
lapack	Ardent Titan	Fortran	linear equations (BLAS level 3)
simple	Ardent Titan	Fortran	2D hydrodynamic/thermal fluid behavior
wake	Ardent Titan	Fortran	free wake of rotor (vortex box panel)
mp3d	VAX T-bit	C	3-d particle simulator for rarefied flow
p-thor	VAX T-bit	C	parallel logic simulator
locus route	VAX T-bit	C	global router for VLSI standard cells
fft64	IBM 370	Fortran	radix-2 fast fourier transform
weather64	IBM 370	Fortran	finite difference weather analysis
speech64	Encore Multimax	Mul-T	lexical decoding of spoken language

Table 1: Trace application summary

Reference Characteristics								
Trace Set	Avg Refs Per Trace (millions)	Inst	Locks	Data	Private Read	Private Write	Shared Read	Shared Write
		(fraction of all references)						
Ardent	20.0	0.654	0.002	0.344	0.067	0.046	0.167	0.064
T-bit	7.3	0.538	0.000	0.462	0.303	0.085	0.061	0.013
MIT	19.4	0.428	0.064	0.508	0.382	0.085	0.028	0.013

Table 2: Workload Reference characteristics

This table lists the average number of references for programs in each set of traces, along with the average fraction of instruction, synchronization, private-read, private-write, shared-read, and shared-write references. A reference is a *shared* reference if it accesses data used by more than one processor during the trace. Averages for the MIT workload do not include *speech64*, as speech64 lacks instruction references due to limitations in the Mul-T tracing environment.

Shared references account for nearly 70 percent of all data references in the Ardent traces, but only about 20 percent for the T-bit and MIT traces. Prior studies [Agar88a, Egge88] have found shared data references to be roughly one-third of all data references, in agreement with the T-bit and MIT trace results. The Ardent vector applications behave differently due to the fact that they make extensive use of large, shared data structures that are operated on by all

processors in parallel. The other applications, captured from scalar workloads, share lesser amounts of data on a finer granularity.

Address Space Breakdown							
Trace Set	Total Kbytes	Inst Kbytes	Data Kbytes	*Percent (%) of Data Bytes*			
				Private Read	Private Write	Shared Read	Shared Write
Geometric Averages							
Ardent	437.3	15.0	385.2	0.0	17.0	0.0	64.5
VAX T-bit	324.6	4.3	318.5	35.4	41.0	6.6	9.2
MIT	543.0	0.0	539.1	2.9	19.7	0.0	0.0
Arithmetic Averages							
Ardent	1036.3	33.3	1003.0	0.1	27.4	1.0	71.5
VAX T-bit	353.1	4.6	348.5	42.2	39.9	8.3	9.6
MIT	1043.5	1.6	1041.9	19.6	59.5	15.1	5.8

Table 3: Address space breakdown in kilobytes

This table lists geometric and arithmetic means for instruction, data, and total address space in kilobytes. Data space is also broken down (percentages) into Private Read, Private Write, Shared Read, and Shared Write categories. Write data is defined as data written during the trace. Values in the upper half of the table are geometric means; thus columns for that data do not sum to 100 percent.

Table 3 provides both geometric and linear averages of the address space size referenced by our traces. On average our traces are quite large. The average Ardent and MIT trace references over one megabyte of data, with much of the data being shared in an Ardent trace. By comparison, the applications used in [Agar88a] reference less than 100 kilobytes of shared data. The programs in [Egge88] appear larger, as shared data regions range from 20 kilobytes to over 25 megabytes. These regions, however, represent statically-allocated shared data, and not the amount of data referenced in the traces.

4. The Cache Consistency Protocols

The efficient algorithms used in this research were designed to evaluate a class of compatible snooping-cache protocols known as the *MOESI protocols* [Swea86]. These protocols are compatible with the control lines and signals on the IEEE Futurebus+ [P896.1a, P896.1b], in the sense that any cache in a multiprocessor system can dynamically perform any action permitted by any of the protocols (in that state, and for that input) without violating cache consistency [Swea86]. The complete MOESI protocol, a superset of all compatible protocols, incorporates alternative actions in various situations, giving designers the flexibility to choose between ease of implementation or potentially higher performance. Among existing protocols, Berkeley

[Katz85], Illinois [Papa84], Dragon [McCr84], Firefly [Stew87], and Write-Once [Good83] can all be supported on Futurebus+, either as defined or with slight modifications. All of the above protocols utilize a subset of the five MOESI states described below:

M: Dirty blocks present only in this cache.

O: Dirty blocks that are potentially *shared*, i.e. cached by more than one processor. Data can only be in state O in one cache, while other caches would hold the data in state S.

E: Clean data present only in this cache.

S: Data that is potentially shared with other caches and is considered clean in this cache. The same data may be present in other caches in either shared clean (S) or shared dirty (O) states.

I: Invalid data that is currently not in this cache.

To support the operation of the various snooping-cache protocols, the IEEE Futurebus+ provides signal lines for the bus master to indicate its intentions, and signal lines for bus slaves to respond to these transactions if necessary; see [Swea86] for a detailed explanation. The bus master asserts the Cache Consistency (**CC**) signal to broadcast its intent to retain a copy of the referenced data (a caching vs. non-caching bus master). A slave cache which contains an exclusive copy of data observes the asserted **CC** signal and changes the state of its data to a shared state. The bus master also asserts the Intent to Modify (**IM**) signal if it intends to write the referenced data, as well as the Broadcast (**BC**) signal if that write will be broadcast to other caches to allow them to update their copies. Invalidate-based protocols do not assert **BC**, forcing all other caches to discard their copies of the data when they observe an asserted **IM** signal.

Slave response signals include the Cache Status (**CS**) line, which is asserted to notify the master processor that a slave cache also intends to retain a copy of the data. Also known as the *shared line*, this signal is monitored by the bus master to determine whether the data that it has referenced should be cached in an exclusive state (M,E) or in a shared state (O,S). The Select (**SL**) signal is also asserted by slave caches which update their copies of shared data when the bus master asserts the IM and BC signals. The Data Intervention (**DI**) signal is asserted by a bus slave when the slave has the latest copy of the data and will supply it data in place of main memory. Finally, the Data Acknowledge **DK** signal, when asserted, states that memory is *not* to be updated when one cache is supplying data to another cache. If **DK** is not asserted, the data is dirty and is written back to memory at the same time it is being transferred to the requesting (master) cache. This process, known as *reflection*, is used in protocols which disallow the

sharing of dirty data.

Tables 4 through 8 contain state diagrams for various MOESI-compatible protocols. These diagrams are in the form of transition matrices showing cache block *result states* and *actions taken* given a *current state* for the cache block and some *local* or *external event* which triggers a transition from that initial state. Figure 2 presents some sample table entries and describes how these entries should be interpreted. In addition to the bus master and bus slave signals shown in the tables, operations **R** and **W** represent reads and writes placed on the bus by the bus master. As we mentioned earlier, these protocols can be implemented in both snooping or directory-based environments. In a snooping environment, each cache maintains the state of its own data (M,O,E,S,or I) and broadcasts consistency transactions to all processors. In a directory-based system, a processor reads the complete set of cache states from the central directory and sends messages to those processors which must take action.

All of these MOESI-compatible protocols are a subset of the complete MOESI protocol shown in Table 11. The complete protocol incorporates various alternative actions in certain situations, giving designers a large degree of flexibility in implementing a cache-consistency protocol. Unlike the more specific protocols outlined in Tables 4 through 8, the complete MOESI protocol utilizes all five MOESI states and allows for cache-to-cache transfers of data whenever possible. These extra features may be useful in a high-performance system, while a modest system would likely include only those features needed to maintain consistency. Tables 9 and 10 show update-based and invalidate-based flavors of the complete MOESI protocol. Both versions use all five MOESI states, disallow reflection, and allow direct cache-to-cache transfers of data whenever possible.

In addition to the MOESI protocols, we present two adaptive protocols which technically do not fall into the MOESI classification, since both require additional states beyond the five MOESI states. Nevertheless, these protocols are MOESI-compatible in the sense that (a) they use the same signals, and (b) a processor-cache combination could implement one of these adaptive protocols and operate in conjunction with other MOESI-compatible protocols without violating cache consistency. These adaptive protocols update data to maintain consistency, but allow slave processors to discard their copies if they believe that they are currently not using the data. By monitoring the shared (CS) line, the processor writing the data can detect when all slave processors have relinquished their copies, and from that point can cache data in an exclusive state.

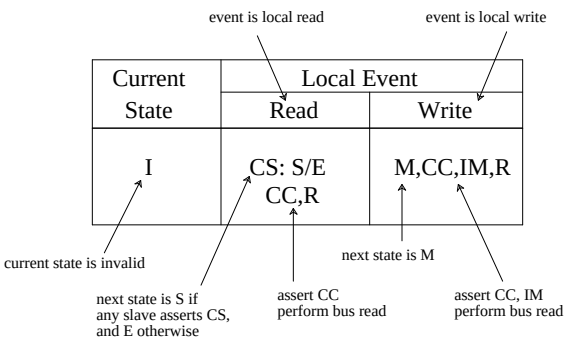

Figure 2: Sample state diagram entry

Berkeley Protocol				
Current State	*Local Event*		*External Event*	
	Read	Write	CC	CC,IM
M	M	M	O,CS,DI,DK	I,DI,DK
O	O	M,CC,IM	O,CS,DI,DK	I,DI,DK
S	S	M,CC,IM	S,CS	I
I	S,CC,R	M,CC,IM,R	I	I

Table 4: State diagram for the Berkeley invalidate-based protocol

Illinois Protocol				
Current State	*Local Event*		*External Event*	
	Read	Write	CC	CC,IM
M	M	M	S,CS,DI	I,DI,DK
E	E	M	S,CS,DI,DK	I
S	S	M,CC,IM	S,CS,DI,DK	I
I	CS:S/E CC,R	M,CC,IM,R	I	I

Table 5: State diagram for the Illinois invalidate-based protocol

Write-Once Protocol				
Current State	*Local Event*		*External Event*	
	Read	Write	CC	CC,IM
M	M	M	S,CS,DI	I,DI,DK
E	E	M	S,CS	I
S	S	E,CC,IM,W	S,CS	I
I	S,CC,R	M,CC,IM,R	I	I

Table 6: State diagram for the Write-Once invalidate-based protocol

Dragon Protocol				
Current State	*Local Event*		*External Event*	
	Read	Write	CC	CC,IM,BC
M	M	M	O,CS,DI,DK	—
O	O	CS:O/M CC,IM,BC,W	O,CS,DI,DK	S,SL,CS
E	E	M	S,CS	—
S	S	CS:O/M CC,IM,BC,W	S,CS	S,SL,CS
I	CS:S/E CC,R	Read>Write	I	I

Table 7: State diagram for the Dragon update-based protocol

Firefly Protocol				
Current State	*Local Event*		*External Event*	
	Read	Write	CC	CC,IM,BC
M	M	M	S,CS,DI	—
E	E	M	S,CS,DI,DK	—
S	S	CS:S/E CC,IM,BC,W	S,CS,DI,DK	S,SL,CS
I	CS:S/E CC,R	Read>Write	I	I

Table 8: State diagram for the Firefly update-based protocol

Full MOESI Update Protocol				
Current State	*Local Event*		*External Event*	
	Read	Write	CC	CC,IM,BC
M	M	M	O,CS,DI,DK	—
O	O	CS:O/M CC,IM,BC,W	O,CS,DI,DK	S,SL,CS
E	E	M	S,CS,DI,DK	—
S	S	CS:O/M CC,IM,BC,W	S,CS,DI,DK	S,SL,CS
I	CS:S/E CC,R	Read>Write	I	I

Table 9: State diagram for the full MOESI update protocol

Full MOESI Invalidate Protocol				
Current State	*Local Event*		*External Event*	
	Read	Write	CC	CC,IM
M	M	M	O,CS,DI	I,DI
O	O	M,CC,IM	O,CS,DI	I,DI
E	E	M	S,CS,DI	I
S	S	M,CC,IM	S,CS,DI	I
I	CS:S/E CC,R	M,CC,IM,R	I	I

Table 10: State diagram for the full MOESI invalidate protocol

Complete MOESI Protocol				
Current State	*Local Event*			
	Read	Write	Pass	Flush
M	M	M	E,CC,BC?,W	I,BC?,W
O	O	CS:O/M CC,IM,BC,W or M,CC,IM	CS:S/E CC,BC?,W	I,BC?,W
E	E	M	—	I
S	S	CS:O/M CC,IM,BC,W or M,CC,IM or S,IM,BC,W* or S,IM,W*	—	I
I	CS:S/E CC,R or S,CC,R* or I,R**	M,CC,IM,R or Read>Write or I,IM,BC,W*,** or I,IM,W*,** or Read>Write	—	—

Complete MOESI Protocol						
Current State	*External Event*					
	CC	CC,IM	BC?	CC,IM,BC	IM	IM,BC
M	O,CS,DI,DK or S,CS,DI	I,DI,DK or I,DI	M,DI,DK,CS?	—	M,DI,DK,CS?	M,SL,CS?
O	O,CS,DI,DK or S,CS,DI	I,DI,DK	CS:O/M DI,DK	S,SL,CS or I	O	O,DI,DK,CS?
E	S,CS or S,CS,DI,DK	I	E,CS?	—	I	E,SL,CS? or I
S	S,CS or S,CS,DI,DK	I	S,CS	S,SL,CS or I	I	S,SL,CS or I
I	I	I	I	I	I	I

Table 11: State diagram for the complete MOESI protocol

Archibald Adaptive Protocol				
Current State	*Local Event*		*External Event*	
	Read	Write	CC	CC,IM,BC
M	M	M	O,CS,DI,DK	—
O	O	CS:O/M CC,IM,BC,W	O,CS,DI,DK	RW1,SL,CS
E	E	M	S,CS,DI,DK	—
S	S	CS:O/M CC,IM,BC,W	S,CS,DI,DK	RW1,SL,CS
RW1	S	CS:O/M CC,IM,BC,W	RW1,CS,DI,DK	RW2,SL,CS
RW2	S	CS:O/M CC,IM,BC,W	RW2,CS,DI,DK	if CS: RW2,SL else: I
I	CS:S/E CC,R	Read>Write	I	I

Table 12: State diagram for the Archibald adaptive protocol

The *Archibald* adaptive protocol [Arch88], which we describe first, contains two additional states beyond the five original MOESI states. The first extra state, *RW1*, represents data that has been updated once by an external processor without being used locally since the external update. The second extra state, *RW2*, represents data that has been updated two or more times without being used by the local processor. Data in states RW1 and RW2 will make transitions back to one of the MOESI states once the local processor references the data. The Archibald protocol specifies that a local processor invalidate blocks in state RW2 when that processor observes yet another external update to that block, unless the *shared* line shows that a third processor is also retaining a copy of the block.

We also evaluate a second adaptive protocol which is very similar to the Archibald protocol, except that invalidates are allowed to occur after the data has reached state *RW1*. We proposed this protocol, which we call the *Update Once* (UpOnce) protocol, in [Gee93] after examining the results from that trace characterization effort. In most of those programs, data was written by one processor either (a) many times in succession, or (b) only once. Given case (a), the Update Once protocol performs only one extra update before making the correct decision and invalidating the data. In case (b), the Update Once protocol always makes the correct decision by broadcasting the write to all slave processors. The state diagram for the Update Once protocol is given in Table 13. As in Archibald, the state of the shared line is monitored before a cache invalidates data it holds.

Update Once Protocol				
Current State	*Local Event*		*External Event*	
	Read	Write	CC	CC,IM,BC
M	M	M	O,CS,DI,DK	—
O	O	CS:O/M CC,IM,BC,W	O,CS,DI,DK	RW1,SL,CS
E	E	M	S,CS,DI,DK	—
S	S	CS:O/M CC,IM,BC,W	S,CS,DI,DK	RW1,SL,CS
RW1	S	CS:O/M CC,IM,BC,W	RW1,CS,DI,DK	if CS: RW1,SL else: I
I	CS:S/E CC,R	Read>Write	I	I

Table 13: State diagram for the Update Once protocol

Both of the adaptive protocols evaluated here are similar to an adaptive protocol described by Karlin, et al [Karl86]. That protocol, however, left it to a writing processor to track the number of consecutive updates that it has performed, and to then invalidate data after some threshold number of updates has been reached. Our protocols are slave-based, but like the Karlin protocol require N consecutive writes by a single master processor to cause slaves to be invalidated. The Karlin proposal originally specified a threshold level where the cost of *N* updates is equal to the cost of an invalidation cache miss. This limits cache consistency overhead to no more than twice that of an optimal protocol which always makes the correct decision. Eggers and Katz [Egge89b] simulated the Karlin protocol where N was set to two updates. Since slave-based and master-based adaptive protocols behave similarly, we would expect the protocol simulated in [Egge89b] and the Archibald protocol to perform comparably.

5. Simulation Results

This section contains results of simulating each of the protocols described in the previous section using the traces described in Section 3. For each simulation run, the following events were counted:

(1) *Cache misses resulting in a block transfer from memory.* The data requested by a processor is not present in its own cache, and is either (a) not present in any other cache or is (b) clean in other caches and the protocol disallows cache-to-cache transfers of clean data.

(2) *Cache misses resulting in a block transfer from another cache.* The data requested by a processor is either (a) dirty in another cache or is (b) clean in other caches and the underlying protocol does support cache-to-cache transfers of clean data.

(3) *Write updates to shared data.* A processor writes a cached item of data that it believes to be shared, and broadcasts the write to all other caches so that they may update their copies.

(4) *Write invalidates to shared data.* A processor writes a cached item of data that it believes to be shared, and notifies all other caches to invalidate their copies.

(5) *Write backs of dirty blocks.* A cache miss triggers a replacement of a dirty block, which is then written back to main memory.

These event counts are then translated into *cache miss ratios*, *data bytes transferred per reference*, *bus cycles*, and *processor utilization*. The first two metrics are time-independent measures which have been used extensively to analyze cache performance in uniprocessor systems. These two metrics are also independent of whether the protocol uses snooping or directories to maintain consistency. In uniprocessor systems, the *cache miss ratio* accurately indicates the performance loss due to cache misses. For multiprocessor systems, however, the first two metrics are not sufficient because the former does not account for consistency traffic (other than misses) and the latter does not reflect non-linear queueing delays, which are additionally a function of bus idle cycles. Bus cycles and processor utilization are direct measures of parameters of interest.

We decided to simulate only data references, because (a) all of the various consistency protocols behave identically for read-only instruction references, (b) some modern computer systems utilize split instruction and data caches to increase memory bandwidth, which minimizes the effect of instructions on protocol performance, and (c) a great deal of simulation time was saved by eliminating instruction references, which represent over half of all memory references. However, omitting instruction references does eliminate network and memory system load due to instruction cache misses, and will make our results for bus traffic and processor utilization somewhat optimistic.

In all simulations, the number of processors simulated conforms to the number of processors in each trace: 4 processors for the Ardent traces, 16 processors for the T-bit traces, and 64 processors for the MIT traces. All traces were simulated over block sizes ranging from 8 to 64 bytes, and for caches up to 1 Mbyte in size. We simulated the Ardent traces under the assumption that cache and memory interfaces are 64-bits wide, equivalent to the bus width in most vector machines. The T-bit and MIT traces were simulated under the assumption that datapaths are 32-bits wide, a figure more typical of general purpose machines. We divided our traces into two

workloads, to reduce the number of data points and simplify the analysis: (1) a *vector workload* consisting of the six Ardent traces, and (2) a *scalar workload* consisting of the six T-bit and MIT traces. Measurements were actually made for each individual trace, and some specific figures appear in [Gee93a,93b]; here we present workload averages only.

5.1. Cache Miss Ratios

The *cache miss ratio* is the fraction of memory references not satisfied by the cache of the referencing processor. It is calculated by dividing the total number of block transfers by the total number of memory references. Figures 3 and 4 show cache miss ratios for the vector (Ardent) and scalar (T-bit and MIT) workloads, respectively. Miss ratios for the Illinois, Write-Once, and full-MOESI invalidate protocols are not shown, as they are identical to miss rates for the Berkeley protocol. Similarly, miss ratios for the Firefly and full-MOESI update protocols are not shown, since they are identical to miss ratios for the Dragon protocol. Please note, as mentioned above, that miss ratios for multiprocessor systems do not directly measure performance, since they do not include consistency traffic.

Results for both workloads are compared against design target [Smit87], SPEC [Gee91], and *uniprocessor* versions of those same workloads. The *design target* miss ratios represent the expected cache performance of multiprogrammed workloads running on uniprocessor systems, while the *SPEC* miss ratios are measurements of a well-known benchmark suite running on RISC-based workstations. Uniprocessor results for the vector workload were generated in a separate study [Gee92], while uniprocessor results for the scalar workload were generated by using the same traces (minus synchronization references) in a single-processor simulation. Miss ratios are fairly consistent across both workloads. Write-invalidate protocols such as Berkeley, Illinois, and Write-Once have the largest miss ratios, as these protocols purge data from other caches to enforce consistency. Miss ratios for write-update protocols such as Dragon and Firefly are lowest, since these protocols maintain shared data in as many caches as possible, thereby decreasing the likelihood of a cache miss. Miss ratios for the Archibald and UpOnce adaptive protocols fall between miss ratios for the invalidate and update-based protocols. Of the two adaptive protocols, Archibald has lower miss ratios, because it allows an additional update to be performed before it switches to invalidating data.

Multiprocessor miss ratios are significantly larger than design target, SPEC, and corresponding uniprocessor miss ratios. This is especially true for larger cache sizes, which do not reduce multiprocessor miss ratios by the factors observed in uniprocessor workloads. This effect is most noticeable for Berkeley and other invalidate-based protocols, because invalidations prevent larger caches from being filled to capacity. Increasing cache block size does substantially decrease miss ratios for the multiprocessor vector workload, but yields only minor

reductions for the multiprocessor scalar workload.

For comparison, [Egge89a] measured miss ratios for write-invalidate protocols as a function of cache size and block size. That study found multiprocessor miss rates to be much higher than uniprocessor miss rates, and also noted the limited improvement in miss rate with larger cache and block sizes. For caches larger than 64 Kbytes combined with a 32-byte block size, [Egge89a] measured miss rates ranging from 0.4 to 1.5 percent, much lower than the 7 to 8 percent that we measured in our programs.

Two factors contribute to the larger miss ratios observed in this study: (a) our traces are larger, referencing several times as much data, and (b) some of our workloads contain many more processors (up to 64 vs. a maximum of 12 in [Egge89a]).

5.2. Data Bytes Transferred per Reference

We also measured the average number of data bytes transferred per memory reference. Data traffic arises from: (a) block transfers due to cache misses, (b) write backs of dirty blocks when they are replaced, and (c) write updates. Data traffic is a better measure of performance than miss ratio, since it includes items (b) and (c), but it still does not accurately reflect performance since it does not measure address cycles or bus idle cycles.

In general, the average number of bytes transferred per reference must be well below the bus width to realize any improvement over a non-caching implementation. Data traffic for the vector and scalar workloads are shown in Figures 5 and 6, while Figure 7 shows data traffic numbers for both workloads as a function of cache block size. All protocols described in Section 4 are represented except for full-MOESI update and full-MOESI invalidate, since their results are identical to results for the Dragon and Berkeley protocols, respectively. All figures also contain results from the design target, SPEC, and uniprocessor versions of the respective vector and scalar workloads.

As expected, data traffic decreases with increasing cache size, due to fewer cache misses. Large caches also reduce data traffic by reducing block replacements and the number of block write-backs. Data traffic does increase with block size, for two reasons: (a) not all of the data fetched in large blocks is used, and (b) large blocks increase consistency overhead due to false sharing. The large increase in traffic vs. block size is most noticeable within the various invalidate-based protocols (Illinois, Berkeley, WriteOnce) in Figure 7, since write-sharing forces entire blocks to be repeatedly invalidated and later read back.

Like miss ratios, data traffic is far higher in multiprocessor workloads relative to uniprocessor workloads. This is easily seen by comparing data traffic for the two multiprocessor workloads against traffic for uniprocessor versions of the same workloads. Of even more interest is

the fact that data traffic in multiprocessor systems remains high, even at very large cache sizes, due to cache consistency overhead.

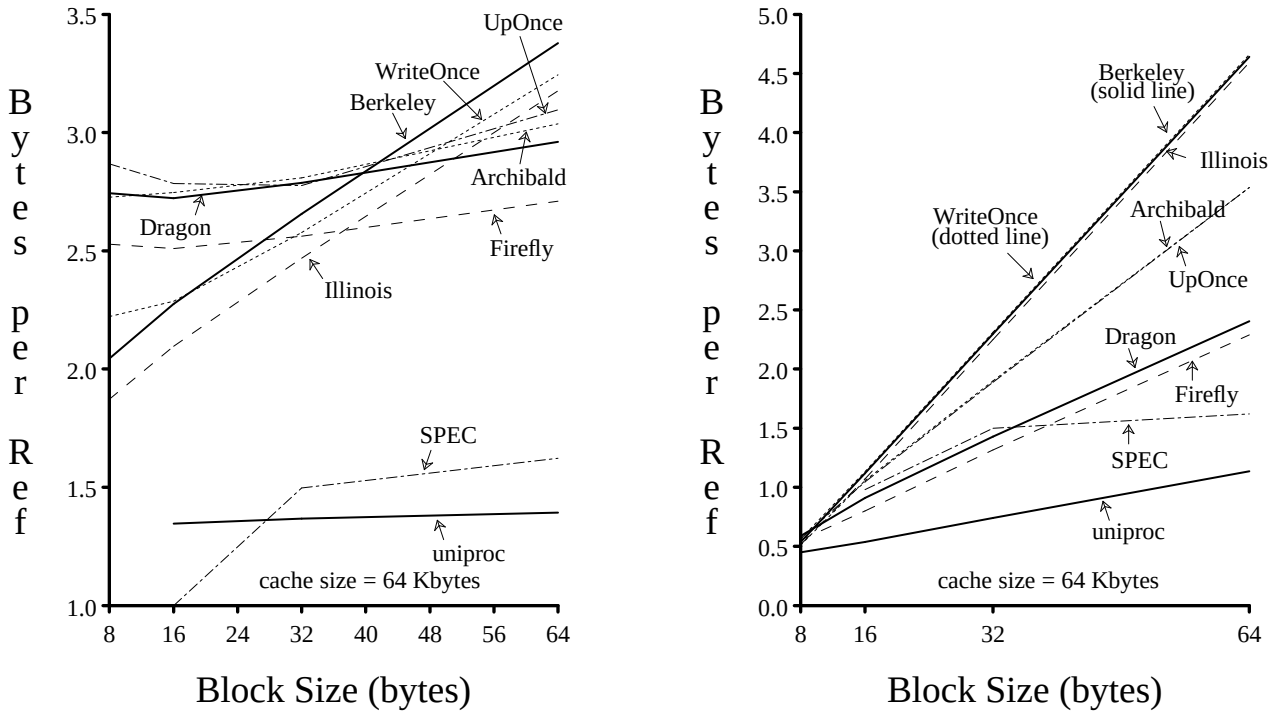

Figure 7: Data traffic vs. block size for vector (left) and scalar (right) workloads.

5.3. Bus Cycles

A more direct and useful measure of protocol performance than data traffic is the average number of *bus cycles* generated by each memory reference. This takes into account address cycles and bus idle cycles (the time between the address cycle and the data transfer). A given processor is idle during the bus cycles it generates, and other processors are prevented from using the bus during that time as well.

The bus transactions measured by our simulator consist of (1) memory-to-cache block transfers, (2) cache-to-cache block transfers, (3) write invalidates, (4) write updates, and (5) write backs of dirty blocks. Several of our protocols also update memory when dirty data is placed on the bus, a process known as *reflection*. Both Illinois and Firefly reflect cache-to-cache block transfers of dirty data, and Firefly also reflects write updates. We assume that write buffers are present to hide the memory latency for reflected transfers, but we also approximately factor in some delays for when write buffers become full.

Table 14 provides costs for these transactions from real implementations and also those assumed in two research studies. Of these implementations, the Berkeley SPUR (Berkeley protocol) and the DEC Firefly (Firefly protocol) are research machines, while the other machines are commercial products. Among commercial machines, the SGI 4D/240 implements the Illinois write-back protocol, the Sequent Balance and Ardent Titan implement Write-Through with Invalidate, and the Intergraph CLIPPER implements Write-Through with Invalidate for shared pages and write-back for private pages. All numbers in Table 14 are for snooping-based implementations.

The timing for block transactions consists of three components: (1) an address cycle, which may take considerable time in snooping environments where slave processors must perform lookups in their local caches before acknowledging receipt of the address (as in Futurebus+), (2) latency cycles before the first item of data arrives, and (3) data transfer cycles proportional to the length of the cache block. Cycle counts for block transfers are thus of the form $T0 + B * T1$, where *T0* cycles are due to address and transfer latency and $B * T1$ cycles is the transfer time for the cache block. The timing for transactions not involving entire cache blocks, such as write invalidates and updates, are independent of block size. Our design parameters are shown in Table 15 for both snooping and directory based implementations, and are based on the following assumptions:

- Split-transaction protocols are not used, forcing the processor initiating the transaction to be delayed for the duration of the transaction.
- Our directory implementation is based on a crossbar interconnection network, allowing message transactions to be sent to all processors simultaneously if needed. Directory lookups require 3 cycles, and must complete before any cache-to-cache transactions can be initiated. These lookups can proceed in parallel with memory-to-cache block transfers, adding no additional cycles to these transactions.
- Snooping write invalidates require 3 cycles: one for the master cache to broadcast the write address on the bus, another for slave processors to receive the address and perform local cache lookups, and a third cycle for all slaves to acknowledge receipt of the broadcast transaction. Directory invalidates require 5 cycles: three for the directory lookup, one cycle to send outgoing invalidation messages to the proper processors, and one cycle to receive incoming acknowledgements.
- Write updates require one cycle more than a write invalidate for both snooping and directory protocols since a data cycle is needed after the address cycle.

- Snooping cache-to-cache block transfers take 3 + B cycles: one for the master cache to broadcast the requested block address, one cycle for slave processors to perform local cache lookups, a third cycle for slaves to acknowledge receipt of the address and cancel the main memory request, and B cycles for the slave to transfer the cache block. Directory cache-to-cache block transfers take 5 + B cycles: three for the directory lookup, one to send the message to the slave processor holding the data, one cycle cache latency for the slave, and B cycles for the slave to transfer the block.
- Memory-to-cache block transfers in both implementations take 8 + B cycles: one for the master to send the address, seven cycles for main memory latency, and B cycles for memory to transfer the block. We selected these parameters to correspond roughly to the memory latency on an Ardent Titan. As this latency is becoming somewhat low relative to current and future machines, we will also discuss results for a 30 cycle memory latency.
- Write updates and cache-to-cache block transfers reflected to memory nominally execute as fast as unreflected transfers, since we assume the existence of write buffering. To account for occasional buffer stalls, we estimate that buffers will overflow roughly 25% of the time (see [Smit79] for some relevant results). Each overflow stalls the machine for four cycles, as we would expect a buffer to become free within half the total memory latency. The frequency of stalls, multiplied by the delay per stall, adds an additional cycle to each reflected transfer.
- Write backs of dirty blocks require 1 + B cycles, one cycle for address transfer and B cycles for data transfer.

Table 15 also lists bus-busy cycles for uncached reads and writes. Completing a data read from memory takes nine cycles, one for address transfer, seven for memory latency, and one for data transfer. For data writes, we assume that write buffers will be provided to hide memory latency, reducing the cost per uncached write to two cycles for address and data transfer. Even with write buffers, however, the nine cycle memory latency will create buffer stalls and increase the average time to perform a write [Smit79].

To simplify the analysis, we ignore the effect of queueing delays on the number of cycles per transaction. As the bus and main memory are expected to be heavily loaded, these queueing delays could be quite large, and could significantly increase the number of cycles per transaction. For an M/M/1 queue, the mean number of requests awaiting service is $r/(1-r)$, where r is the load measured by dividing the mean arrival rate by the mean service rate.

Cycles Per Transaction						
Study or Machine	Memory to Cache	Cache to Cache	Cache to Cache*	Inv.	Update	Update*
[Arch86]	3+B	B	3+B	1	1	4
[Agar88b]	3+B	2+B	2+B	1	2	2
Berkeley SPUR [Wood87]	10 + B	10 + B	-	12	-	-
DEC Firefly [Thac87]	3 + B	3 + B	3 + B	-	-	4
SGI 4D/240 [Leno90]	6 + B	6 + B	6 + B	1-2	-	-
Sequent Balance [Thak88]	6 + B	-	-	2	-	-
Ardent Titan [Died88]	8+B	-	-	1	-	-
Intergraph CLIPPER [Cho86]	4+B	-	-	4	-	-

Table 14: Timing parameters taken from various studies and implementations. For block transfers, **B** is the block size in words. Asterisks denote transactions reflected to main memory.

Cycles Per Bus Transaction		
Event	Cycles (snooping)	Cycles (directory)
write invalidate	3	5
write update	4	6
write update (reflected)	5	7
cache/cache transfer	3 + B	5 + B
cache/cache transfer (reflected)	4 + B	6 + B
memory/cache transfer	8 + B	8 + B
write back	1 + B	1 + B
uncached read	9	9
uncached write	2	2

Table 15. Costs per bus transaction (B = block size in words)

From the timing parameters in Table 15, we generated bus cycles for each combination of workload and cache-consistency protocol. These cycle counts, divided by the number of references simulated, yield the average number of bus cycles per reference. Figures 8 and 9 present results for snooping-based implementations of each protocol; corresponding figures for directory-based implementations are left to the Appendix. Figures 8 and 9 also contain *design target memory delays* from [Smit87], adjusted for the seven cycle memory latency in our model. The design targets represent the average delay per memory reference in general-purpose uniprocessor systems. In Table 16 below, we list bus cycles per reference for various multiprocessor

caching implementations, and compare them to the same workloads running on (a) multiprocessor implementations without data caching and (b) uniprocessor implementations with data caching.

Our results in the figures and in Table 16 clearly show the need for private caches in shared-memory multiprocessor systems. The average number of bus cycles per memory reference is roughly seven cycles without caches, but only about one cycle with large coherent caches. However, the average number of bus cycles per reference remains far higher than levels observed in uniprocessor systems, because of the overhead to maintain cache consistency. This sharing overhead must somehow be reduced in order to extract satisfactory performance from multiprocessor systems.

Average Bus Cycles Per Reference				
Protocol	Cache Size	Block Size	Cycles per Reference	
			Vector Wkld	Scalar Wlkd
mp without caching	**-**	**-**	**6.70**	**7.50**
Berkeley (snooping)	16KB	32	0.96	0.98
Berkeley (directory)	16KB	32	1.06	1.11
Dragon (snooping)	16KB	32	1.16	0.78
Dragon (directory)	16KB	32	1.38	0.93
uniprocessor	**16KB**	**32**	**0.62**	**0.79**
Berkeley (snooping)	128KB	32	0.60	0.92
Berkeley (directory)	128KB	32	0.74	1.07
Dragon (snooping)	128KB	32	0.92	0.67
Dragon (directory)	128KB	32	1.26	0.82
uniprocessor	**128KB**	**32**	**0.28**	**0.20**
Berkeley (snooping)	1MB	32	0.54	0.92
Berkeley (directory)	1MB	32	0.69	1.06
Dragon (snooping)	1MB	32	0.82	0.65
Dragon (directory)	1MB	32	1.18	0.79
uniprocessor	**1MB**	**32**	**0.10**	**0.14**

Table 16. Average bus cycles per memory reference for workloads on multiprocessor implementations with and without caching and in uniprocessor implementations with caching (8 cycle memory latency).

Based on our results, no one consistency protocol outperforms all other protocols across all workloads, as has been noted in previous studies. In the vector workload, the *full-MOESI invalidate* protocol minimizes bus cycles, followed closely by *Illinois* and *Berkeley*, the two other invalidate-based protocols. *UpOnce* and *Archibald* are also effective, while the update-based *full-MOESI update*, *Dragon*, and *Firefly* protocols are least effective. Results for the scalar workload are quite the opposite, as the *full-MOESI update* protocol minimizes bus cycles, followed closely by *Firefly*, *Dragon*, *UpOnce*, and *Archibald*. The three invalidate-based protocols are least effective in the scalar workload as they are unsuited to the fine granularity of sharing in these programs.

We also measured bus cycles for a memory latency of 30 cycles, to reflect the next generation of faster processors; results are shown in the Appendix. The larger memory latency increases the average number of cycles per reference by 10 to 30 percent, yet the relative performance of the various protocols is largely unchanged compared to the first set of results. There are, however, two new results of significance: (1) protocols supporting cache-to-cache transfers of clean data suffer the least dropoff in performance, while (2) protocols supporting *reflection* are most severely impacted, due to the increased write back delay when write buffers become full. As processors become faster and memory latencies increase, protocols which avoid memory transactions when possible perform much better relative to other protocols.

In Figure 10, we plot the minimum number of bus cycles generated by any of the protocols. Results pertain only to snooping protocols, at cache sizes of 16, 128, and 512 Kbytes, and for a memory latency of 8 cycles. For comparison, we also include results for the uniprocessor version of the vector workload using the same memory system parameters. (Results for the uniprocessor version of the scalar workload are similar to results for the uniprocessor vector workload, and are omitted from the figure to improve clarity). It can be seen in Figure 10 that minimum cycles for both uniprocessor and multiprocessor vector workloads decrease with increasing block size across the entire parameter range. In the multiprocessor scalar workload, cycle counts increase for every increase in block size, due to the much higher presence of false sharing. More importantly, we observe that uniprocessor cycle counts are not only much lower than multiprocessor cycle counts, but also decrease much more rapidly with increasing cache size. *Bigger caches do not improve multiprocessor performance by the same large factors because of the bus traffic overhead to keep multiprocessor caches consistent.* This overhead must be reduced, which can only be accomplished by *recoding existing applications to minimize sharing*. Unless this recoding effort takes place, our results strongly suggest that even the best cache consistency protocols will not provide adequate performance.

Due to space limitations, the data for directory-based versions of these protocols is left to the attached Appendix. Due to longer latencies for most transactions, directory-based protocols

typically generate 10 to 30 percent more bus cycles than their snooping-based counterparts. However, directory-based protocols do not depend on the shared bus needed for snooping protocols. Thus directory transactions may require more time to complete, but need not contend or wait for a single, shared resource, and thus should scale much better.

Prior studies have shown generally similar results, although for smaller trace sets and fewer cases. Unlike prior studies, however, we have available a large number of traces from a wide variety of workloads, rather than only a limited number of highly similar traces. We were able to observe a strong correlation between the type of workload running on a multiprocessor system and the best choice of protocol for that workload. Machines running coarse-grain, scientific programs on a limited number of processors operate best with invalidate-based protocols paired with large block sizes. No prior study had analyzed vectorized traces such as these. We also noticed that machines running fine-grain, scalar applications across many processors should be paired with update-based protocols and small block sizes. Finally, adaptive protocols, although never optimal for a given application, provide consistently good performance across all programs.

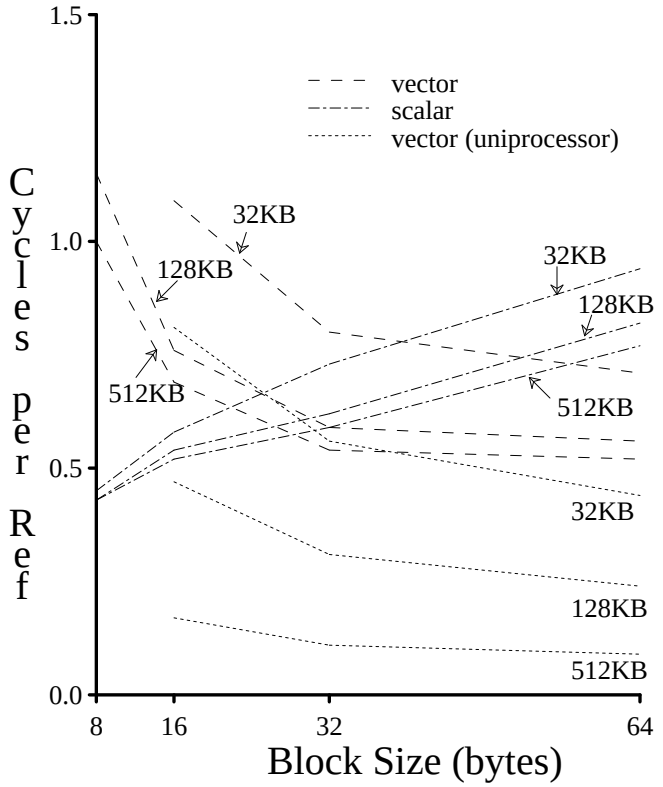

Figure 10: Minimum bus cycles observed in both workloads

It is also instructive to examine protocol performance on *individual traces* as well as on complete workloads. Due to the large number of traces used in this study, we provide only one data point, as shown in Table 17. For each trace, bus cycles per reference were computed for all protocols assuming a snooping implementation, with a cache size of 128 Kbytes and a block size of 32 bytes. These results were then normalized by dividing bus cycles by the minimum number of bus cycles generated by any of the protocols on that trace. The last rows in Table 17 contain geometric means of these ratios for the vector workload, scalar workload, and both workloads combined.

Snooping Protocol Performance on Individual Traces
(ratio of bus cycles to minimum bus cycles for that trace)

Trace	Invalidate Protocols			Update Protocols			Adaptive Protocols	
	MOE.i	Ill	Berk	MOE.u	Drag	Fire	UpOnce	Arch
arc3d	1.00	1.00	1.05	1.39	1.42	1.54	1.08	1.14
bmk1	1.00	1.00	1.00	1.00	1.00	1.50	1.50	1.50
bmk11a	1.00	1.00	1.04	1.68	1.72	1.86	1.34	1.50
flo82	1.00	1.04	1.00	1.61	1.61	2.00	1.09	1.11
lapack	1.08	1.00	1.08	1.35	1.35	1.38	1.20	1.29
simple	1.00	1.05	1.02	1.38	1.40	1.65	1.22	1.32
wake	1.00	1.03	1.00	2.16	2.23	2.61	1.32	1.55
dcsim16	1.14	1.14	1.25	1.04	1.07	1.00	1.00	1.00
lr16	1.00	1.00	1.13	1.00	1.13	1.00	1.00	1.00
mp3d16	1.65	1.74	1.80	1.00	1.14	1.06	1.05	1.00
fft64	2.65	2.65	2.71	1.00	1.06	1.10	1.02	1.02
spch64	1.37	1.43	1.41	1.00	1.05	1.20	1.52	1.61
wthr64	1.23	1.23	1.35	1.05	1.16	1.00	1.05	1.05
Geometric Averages								
vector	1.01	1.02	1.03	1.47	1.49	1.75	1.24	1.33
scalar	1.43	1.45	1.54	1.02	1.10	1.06	1.09	1.10
all	1.19	1.20	1.24	1.24	1.30	1.39	1.17	1.22

Table 17: For each protocol and trace, table entries correspond to the number of bus cycles generated by that protocol, divided by the fewest number of bus cycles observed for that trace (1.00 = best). Cache size is 128 Kbytes. Block size is 32 bytes. Memory latency is 8 cycles.

In six of the seven vector traces, the full-MOESI invalidate protocol generates the fewest number of bus cycles, while the full-MOESI update protocol generates the fewest bus cycles for four of six scalar traces. There are cases where multiple protocols perform equally well. One

observation of note is that protocols which are purely invalidate-based or update-based perform poorly on at least a few traces. In contrast, the adaptive UpOnce protocol performs strongly in all traces, and in many instances performs nearly as well as the best protocol for that trace. This consistent level of performance is clearly evident in the geometric means of the ratios, where UpOnce narrowly edges out MOESI-invalidate as the best performing protocol over all traces.

5.4. Processor Utilization

In this last section, we estimate *processor utilization*, the fraction of program execution time during which processors are performing useful work. Processor time, as noted above, is lost due to memory access delays, time for consistency transactions, and queueing delays to access the memory.

We compute processor utilization by taking the ratio of CPU busy cycles to the number of cycles needed to execute an application. To estimate CPU busy cycles, we first determine the number of instructions in each trace (I), divide this number by the number of processors in the trace (P), and then multiply the result by the number of execution cycles per instruction (CPI). The formula for CPU busy cycles is thus *(I / P) * CPI*.

The number of cycles needed to execute an application is much more difficult to estimate, and is dependent heavily upon the multiprocessor architecture. We are further limited by the output of our simulator, which only measures the frequency of various consistency events. In our approximation, we assume that the number of cycles needed to execute an application is no less than the sum of (a) CPU busy cycles per processor and (b) bus cycles which stall a given processor. To calculate bus cycles which stall a processor, we assume that each processor *waits on* 1/P of all block transfers, *sends* 1/P of all cache-to-cache block transfers, *receives* 1/P of all invalidates (only one processor is invalidated per invalidation), and *receives* all updates (updates can presumably affect all processors).

The above formula should provide a reasonable, albeit optimistic, estimate of processor utilization for directory-based protocols. The directory-based results could be less optimistic had we chosen to factor in the effect of queueing delays. We cannot ignore queueing for snooping protocols, however, as processors wishing to initiate consistency transactions will often wait for the shared bus to become available. Under such conditions, the first formula does not hold. What we can do is calculate the total number of bus cycles generated by *all* processors in a snooping implementation. Since we know that no application can complete in less time than that number of cycles, processor utilization for snooping protocols becomes the lesser of (1) the directory-based estimate:

(1) (CPU Busy Cycles) / (CPU Busy Cycles + per CPU Bus Cycles)

or (2) a snooping-based estimate which assumes bus saturation:

(2) (CPU Busy Cycles) / (Total Bus Cycles)

The only unknown parameter is CPI, which depends upon the type of machine used to generate each trace. For the vector traces, we use a fairly low CPI of 1.6 cycles, as the Ardent Titan is a RISC-based machine with a quoted performance of 10 MIPS at a 16 MHz clock rate [Died88]. The scalar traces were collected on CISC processors, which generally have a much larger CPI than RISC processors. Clark [Clar88] analyzes the CPI for a VAX 8800 in detail, and quotes an average of 6.8 execution (non-memory) cycles per VAX 8800 instruction. We use this figure for the scalar traces.

We now have sufficient information to estimate processor utilization. As mentioned beforehand, our results will be somewhat optimistic because we omit instruction references from the simulations and only approximate queueing. Figure 11 displays the average processor utilization across all traces in the scalar and vector workloads. Results are shown for snooping-based and directory-based implementations of the Update Once protocol, and we also provide results for uniprocessor versions of both workloads.

For the scalar workload, processor utilization is always higher in a directory-based environment, and is often double the utilization observed in a snooping environment. The scalar traces contain references from 16 to 64 processors, which is sufficient to saturate any shared-bus implementation. For the vector workload, directory protocols outperform snooping protocols at small cache sizes, because the increased frequency of misses at these sizes saturates the shared snooping bus. Snooping protocols eventually outperform directory protocols for larger cache sizes, as one would expect from a workload tuned to a small-scale, shared bus multiprocessor. Even so, the performance improvement from snooping protocols is modest, on the order of 25 percent.

As expected, processor utilization is highest for the vector workload for a 64-byte block size, and for the scalar workload at an 8-byte block size. Cache size has an interesting effect on the results, as there are instances where utilizations actually decrease slightly with increasing cache size. Large caches increase the amount of nominal sharing, and by doing so force additional broadcast updates.

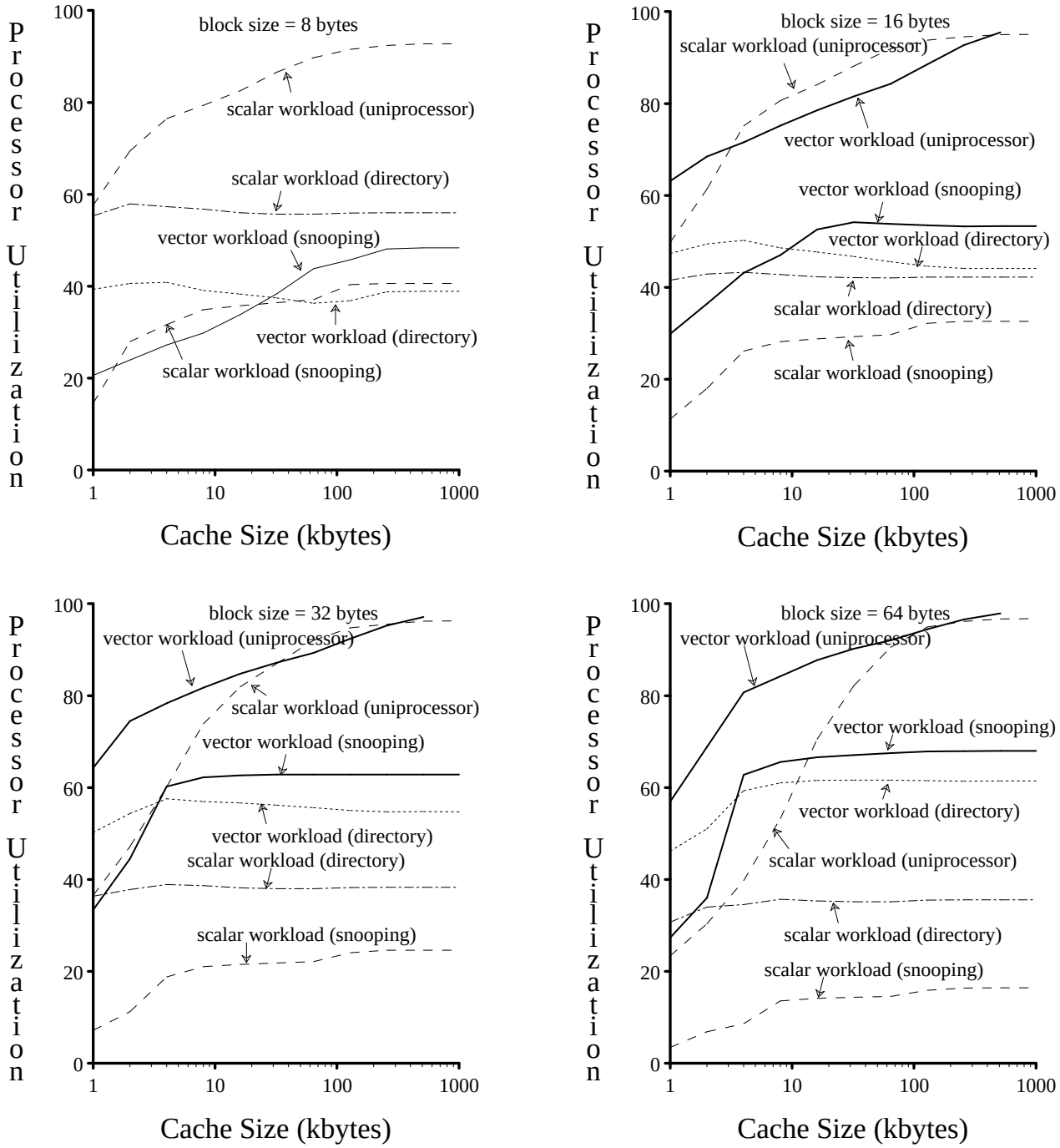

Figure 11: Processor utilizations (%) for the UpOnce protocol. Results are provided for both workloads in snooping and directory environments, and for uniprocessor versions of both workloads. Memory latency is 8 cycles.

A final observation is that processor utilization in multiprocessor systems barely reaches or exceeds 60 percent for either workload, and these are for optimistic estimates with instruction references omitted. Corresponding utilizations for uniprocessor versions of both workloads approach 90 percent. Furthermore, increasing memory latency from 8 cycles to 30 cycles results in as much as a 50 percent drop in processor utilization (see Appendix). This drop occurs despite the fact that the Update Once protocol is quite efficient and avoids memory and bus transactions whenever possible. These results confirm our earlier finding that shared-memory multiprocessors are of dubious utility unless changes are made to reduce sharing in current applications.

6. Conclusions

We have examined in detail the performance of a large number of cache consistency protocols, four of which (MOESI invalidate, MOESI update, Update Once, Archibald) have not been examined in previous simulation studies. We have focused on identifying the best-performing protocols and the protocol features most useful to high performance. Efficient simulation algorithms [Thom87] were employed to facilitate the exploration of a large design space. The input to the multiprocessor simulator consisted of a wide selection of multiprocessor address traces, representing one production vector workload and one scalar workload. The thirteen traces that we used for our trace driven simulations are far more in number and variety than traces used in any previous simulation study.

We analyzed performance in terms of (a) cache miss ratios, (b) data traffic, (c) bus cycles per memory reference, and (d) processor utilization. Bus cycles and processor utilization were measured in both snooping and directory-based environments, providing results applicable to a wide range of multiprocessor architectures.

Although update-based protocols minimize miss ratios and usually minimize data traffic, no protocol clearly outperforms all others in terms of bus cycles and processor utilization. Protocols which invalidate data provide the best performance for our vectorized workload, while update-based protocols provide the best performance for our scalar workload. One interesting result is that adaptive protocols, while never optimal for a given workload, perform best on the average. When ranking individual protocols, the best protocol among invalidate-based protocols is *MOESI Invalidate*, the best among update-based protocols is *MOESI Update*, and the best among adaptive protocols is *Update Once*.

All three of these protocols utilize the full suite of MOESI states, and also include performance enhancing features such as allowing cache-to-cache transfers of clean data and the sharing of dirty data (no reflection). These features minimize the number of transactions involving shared memory, which is the slow path in any system. As processors become faster and memory latencies continue to increase, these features will become even more desirable. For an

snooping-based multiprocessor with an 8 cycle memory latency, allowing dirty data to be shared reduces average consistency cycles by 1 percent for invalidate-based protocols (Table 17, MOESI invalidate vs. Illinois) and by 11 percent for update-based protocols (Table 17, Firefly vs. MOESI update). Allowing cache-to-cache transfers of clean data decreases bus cycles by 5 percent for both invalidate and update-based protocols (Table 17, Dragon vs. MOESI update, Berkeley vs. MOESI invalidate).

Although we have focused on identifying the highest performing protocols, a major conclusion of our work is that *even the best protocols generate much larger miss ratios, data traffic, and memory access delays than results observed on uniprocessor systems*. Large caches and/or larger block sizes alleviate the problem somewhat, but the overhead to maintain cache consistency continues to consume large numbers of cycles. Processor utilization barely exceeds 60 percent in even the best circumstances, and actually can decrease slightly as cache size (and thus sharing) increases. Corresponding utilizations for uniprocessor systems approach 90 percent. To extract good performance out of shared-memory multiprocessor systems, *programs and algorithms must be redone to avoid interprocessor sharing and communication as much as possible*. Only then will performance reach acceptable levels, and we would then expect a pure invalidate-based protocol such as MOESI invalidate to provide the best performance among all protocols.

Bibliography

[Agar88a] A. Agarwal and A. Gupta, ‘‘Memory Reference Characteristics of Multiprocessor Applications under Mach,’’ *Proc. ACM Sigmetrics*, May, 1988, Santa Fe, NM, pp. 215-225.

[Agar88b] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, ‘‘An Evaluation of Directory Schemes for Cache Coherence,’’ *Proc. 15th Annual Int’l Symp. Comp. Arch.*, May, 1988, Honolulu, HI, pp. 280-289.

[Arch85] J. Archibald and J. L. Baer, ‘‘An Economical Solution to the Cache Coherence Problem,’’ *Proc. 12th Int’l Symp. Comp. Arch.*, June, 1985, Boston, MA, pp. 355-362.

[Arch86] J. Archibald and J. L. Baer, ‘‘Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model,’’ *ACM Trans. on Comp. Sys.*, November, 1986, pp. 273-298.

[Arch88] J. Archibald, ‘‘A Cache Coherence Approach For Large Multiprocessor Systems,’’ *Proc. 1988 Int’l Conf. Supercomputing*, July, 1988, St. Malo, France, pp. 337-345.

[Broo90] E.D. Brooks and J.E. Hoag, ‘‘A Scalable Coherent Cache System with Incomplete Directory State,’’ *Proc. 1990 Conf. on Parallel Processing*, August, 1990, St. Charles, IL, pp. I-553 to I-554.

[Cens78] L. Censier and P. Feautrier, ‘‘A New Solution to Coherence Problems in Multicache Systems,’’ *IEEE Trans. on Comp.*, C27, 12, December, 1978, pp. 1112-1118.

[Chai90] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, ‘‘Directory-Based Cache Coherence in Large Scale Multiprocessors,’’ *IEEE Computer*, June, 1990, pp. 49-58.

[Chai91] D. Chaiken, J. Kubiatowicz, and A. Agarwal, ‘‘LimitLESS Directories: A Scalable Cache Coherence Scheme,’’ *Proc. ASPLOS-IV*, April, 1991, Santa Clara, CA, pp. 224-234.

[Cho86] J. Cho, A.J. Smith, and H. Sachs, ‘‘The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLLPPER Processor,’’ U.C. Berkeley Technical Report No. UCB/CSD 86/289, April, 1986.

[Clar88] D.W. Clark, P.J. Bannon, J.B. Keller, ‘‘Measuring VAX 8800 Performance with a Histogram Hardware Monitor,’’ *Proc. 15th ISCA*, May,

1988, Honolulu, Hawaii, pp. 176-185.

[Died88] T. Diede, C. Hagenmaier, G. Miranker, J. Rubinstein, and W. Worley, ‘‘The Titan Graphics Supercomputer Architecture,’’ *Computer*, September 1988, pp. 13-30.

[Egge88] S. Eggers and R. Katz, ‘‘A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation,’’ *Proc. 15th ISCA*, May, 1988, Honolulu, Hawaii, pp. 373-382.

[Egge89a] S. Eggers and R. Katz, ‘‘The Effects of Sharing on the Cache and Bus Performance of Parallel Programs,’’ *Proc. ASPLOS III Conference*, April, 1989, Boston, Mass., pp. 257-270.

[Egge89b] S. Eggers and R. Katz, ‘‘Evaluating the Performance of Four Snooping Cache Coherency Protocols,’’ *Proc. 16th ISCA*, June, 1989, Jerusalem, Israel, pp. 2-15.

[Egge90] S. Eggers and T. Jeremiassen, ‘‘Eliminating False Sharing,’’ University of Washington, Seattle, Technical Report No. 90-12-01.

[Gee91] J. Gee, M.D. Hill, D. Pnevmatikatos, and A.J. Smith, ‘‘Cache Performance of the SPEC Benchmark Suite,’’ U.C. Berkeley Tech. Report No. UCB/CSD 91/648 and University of Wisconsin Madison Tech. Report No. 1049, September 1991.

[Gee92] J. Gee and A.J. Smith, ‘‘Vector Processor Caches,’’ U.C. Berkeley Tech. Report No. UCB/CSD 92/707, October, 1992.

[Gee93a] J. Gee and A.J. Smith, ‘‘Analysis of Multiprocessor Memory Reference Behavior,’’ paper in preparation, 1993.

[Gee93b] J. Gee, ‘‘Analysis of Cache Performance in Vector Processors and Multiprocessors’’, Ph.D. dissertation, University of California, Berkeley, April, 1993.

[Ghar91] K. Gharachorloo, A. Gupta, and J. Hennessy, ‘‘Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors,’’ *Proc. ASPLOS-IVR, April, 1991, Santa Clara, CA, pp. 245-257.*

[Good83] J. Goodman, ‘‘Using Cache Memory to Reduce Processor- Memory Traffic,’’ *Proc. 10’th ISCA*, and *Sigarch Newsletter*, 11, 3, June, 1983, pp. 124-131.

[Gupt90] A. Gupta, W. Weber, and T. Mowry, ‘‘Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes,’’ *Proc. 1990 Int’l Conf. on Parallel Processing*, August, 1990, St. Charles, IL, pp. I-312 to I-321.

[HePa90] Hennessy, J.L., and Patterson, D.A., *Computer Architecture: A Qualitative Approach*, Morgan Kaufman, 1990.

[Hill86] M. Hill, S. Eggers, Larus, Taylor, Adams, Bose, Gibson, Hensen, Keller, Kong, Lee, Pendleton, Ritchie, Wood, Zorn, Hilfinger, Hodges, Katz, Ousterhout, Patterson, ‘‘Design Decisions in SPUR,’’ *IEEE Computer*, November, 1986, pp. 8-22.

[Karl86] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator, ‘‘Competitive Snoopy Caching,’’ *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, Toronto, Canada, October, 1986, pp. 244-254.

[Katz85] R. Katz, S. Eggers, D. Wood, C.L. Perkins, and R.G. Sheldon, ‘‘Implementing a Cache Consistency Protocol,’’ *Proc. 12th Int’l Symp. Comp. Arch.*, June, 1985, Boston, MA, pp. 276-283.

[Leno90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, ‘‘The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor.’’ *Proc. 17th Int’l Symp. Comp. Arch.*, May, 1990, Seattle, WA, pp. 148-159.

R.L. Mattson, J. Gecsei, D.R. Slutz, and L.L. Traiger, ‘‘Evaluation Techniques for Storage Hierarchies,’’ *IBM Systems Journal*, 2, 1970, pp. 78-117.

[McCr84] E. McCreight, ‘‘The Dragon Computer System, An Early Overview,’’ Xerox PARC Technical Report, June, 1984.

[Okra90] B. O’Krafka and A. Newton, ‘‘An Empirical Evaluation of Two Memory-Efficient Directory Methods,’’ *Proc. 17th Int’l Symp. Comp. Arch.*, May, 1990, Seattle, WA, pp. 138-147.

[P896.1a] *Futurebus P896.1: A Backplane Bus Specification for Multiprocessor Architectures* (Draft 7.5a), IEEE, New York, June, 1987.

[P896.1b] *Futurebus+ P896.1: Logical Layer Specifications* (Draft 8.2), IEEE, New York, January, 1990.

[Papa84] M. Papamarcos and J. Patel, ‘‘A Low Overhead Coherence Solution for Multiprocessing with Private Cache Memories,’’ *Proc. 11th Int’l Symp. Comp. Arch.*, June, 1984, Ann Arbor, Michigan, *Sigarch Newsletter*, 12, 3, June, 1984, pp. 348-354.

[Sega84] Z. Segall and L. Rudolph, ‘‘Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor,’’ *Proc. 11th Int’l Symp. Comp. Arch.*, June, 1984, pp. 340-347.

[Smit79] A.J. Smith, ‘‘Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write Through,’’ *Journal of the ACM*, vol. 26, no. 1, January, 1979, pp. 6-27.

[Smit82] A.J. Smith, ‘‘Cache Memories,’’ *ACM Computing Surveys*, vol. 14, no. 3, September, 1982, pp. 474-529.

[Smit87] A.J. Smith, ‘‘Line (Block) Size Selection in CPU Cache Memories,’’ *IEEE Trans. Comp.*, C-36, 9, September, 1987, pp. 1063-1075.

[Stew87] L. Stewart, ‘‘Firefly, A Small VAX Multiprocessor,’’ presentation slides, 1987.

[Stun91] C. Stunkel, B. Janssens, and W.K. Fuchs, ‘‘Address Tracing for Parallel Machines,’’ *IEEE Computer*, January, 1991, pp. 31-38.

[Swea86] P. Sweazey and A.J. Smith, ‘‘A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus,’’ *Proc. 13th Int’l Symp. Comp. Arch.*, Tokyo, Japan, June, 1986, pp. 414-423.

[Tang76] C.K. Tang, ‘‘Cache Design in the Tightly Coupled Multiprocessor System,’’ *AFLPS Conference Proc., Nat’l Comp. Conf.*, June, 1976, New York, NY, pp. 749-753.

[Thac87] C. Thacker, and L. Stewart, ‘‘Firefly: A Multiprocessor Workstation,’’ *Proc. 2’nd ASPLOS*, October, 1987, Palo Alto, CA, pp. 164-172.

[Thak88] S. Thakkar, P. Gifford, and G. Fielland., ‘‘The Balance Multiprocessor System,’’ *IEEE MICRO*, February, 1988, pp. 57-69.

[Thom87] J. Thompson, ‘‘Efficient Analysis of Caching Systems,’’ Technical Report UCB/CSD 87/374, October, 1987, Computer Science Division, UC Berkeley.

[Tore90] J. Torellas and J. Hennessy, ‘‘Estimating the Performance Advantages of Relaxing Consistency in a Shared-Memory Multiprocessor,’’ *Proc. 1990 Int’l Conf. on Parallel Processing*, August, 1990, St. Charles, IL, pp. I: 26-34.

[Vash93] Bart Vashaw, ‘‘Address Trace Collection and Trace Driven Simulation of Bus Based, Shared Memory Multiprocessors’’, Carnegie Mellon University Dept. of Electrical and Computer Engineering Research Report CMUCDS-93-4, March, 1993.

[Webe89] W. Weber and A. Gupta, ‘‘Analysis of Cache Invalidation Patterns in Multiprocessors,’’ *Proc. ASPLOS-III*, Boston, MA, April 1989, pp. 243-256.

[Wood87] D.A. Wood, S.J. Eggers, and G.A. Gibson, ‘‘SPUR Memory System Architecture,’’ Technical Report No. UCB/CSD 87/394, University of California, Berkeley, December, 1987.