Analysis of Multiprocessor Memory Reference Behavior*

Jeffrey D. Gee
Alan Jay Smith

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

ABSTRACT

Shared-memory multiprocessors can provide impressive performance at reasonable costs, although private caches are usually needed to alleviate the potential bottleneck at shared memory. These private caches in turn require the use of *cache-consistency (coherency) protocols*, whose performance is a strong function of the reference behavior within multiprocessor applications. In this paper we characterize the memory reference behavior in a wide variety of scalar and vector multiprocessor address traces from production workloads. This analysis is for the purpose of estimating and improving the performance of cache-consistency protocols. Our analysis extends previous results in the literature by performing a wider variety of analyses, and analyzing a larger and more diverse set of multiprocessor traces, including a production vector workload.

We find wide differences between the sharing behavior observed in vector and scalar applications. Compared to scalar programs, vector programs reference shared data more frequently and contain larger amounts of *processor locality*, the tendency for shared data to be used by only one processor over periods of time. Write sharing by different processors over short intervals are infrequent in one workload but frequent in another. This implies that sequentially-consistent programming models will remain necessary unless applications are recoded to avoid such reference patterns.

June 2, 1993

* *The authors' research is supported in part 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

As more and more computer systems use multiprocessing for increased performance, continued research is needed to improve on existing *cache consistency (coherency)* protocols. These protocols perform the crucial function of allowing any processor in a multiprocessor system to privately cache shared data for faster access, while specifying a sequence of actions to guarantee that all cached copies of data are identical. There are two main implementations for consistency protocols: *Bus-based* protocols broadcast writes to shared data to other processors on a shared bus. Other processors caching the same data update or invalidate their versions to maintain consistency; this type of system is sometimes referred to as a ‘‘snooping cache system.’’ *Directory-based* protocols do not require a system-wide shared bus; most of the state information for cached copies is kept in a directory based at the main (shared) memory, and the information to maintain consistency is sent to some or all processors as necessary.

Although a number of cache-consistency protocols have been proposed in the literature ([Arch86b, Swea86, Agar88b] summarize many of these protocols while [Smit91] presents a comprehensive bibliography of research in this area), the effectiveness of any protocol is largely dependent on the memory reference behavior within the multiprocessor system. This paper analyzes the memory reference behavior in multiprocessor applications to (a) determine how data is shared in these applications, (b) estimate which existing protocols would perform best on different workloads, and (c) propose improved protocols.

We examine characteristics such as (a) *processor locality*, the tendency for all accesses to shared data over short periods of time to be made by the same processor, (b) *temporal locality*, the property by which data is likely to be reused once it has been referenced, and (c) *contention*, where several processors require access to an item of data during overlapping intervals. Our analysis is carried out in a protocol independent manner, so that our results pertain to bus-based, directory-based, or even network-based protocols.

The contributions of this paper, beyond what already exists in the literature, come from two aspects of this work. First, we analyze a much larger number and variety of multiprocessor address traces than have been previously studied. These traces come from three different sources: 4-processor Ardent Titan traces, 16-processor VAX T-bit traces, and 64-processor IBM and Encore Multimax traces. The Ardent traces are not only new to the research community, but also contain vector data from large, real-world production applications typically run on supercomputers, making them potentially more useful and representative than traces gathered in academic or research environments. The second important contribution is that our analysis is more thorough and extensive than the previously published work; in particular, we describe the reference process with a Markov model, and we do detailed studies as a function of cache line size.

In summary, we find that all three workloads contain some processor locality, although processor locality in the Ardent vectorized workload increases with block size far beyond levels for the other workloads. Temporal locality is present, as successive references to an item of shared data are closely spaced in time, although this is more true when one processor is reusing data as opposed to several processors sharing data in round-robin fashion. We also find that write-shared data is generally used by only one processor at a time, and is often protected by locks. These two characteristics suggest the use of higher-performing, weaker cache consistency protocols. In some of our traces, however, processors frequently read unprotected data that was recently modified by another processor. Applications sharing data in this manner will continue to require strongly consistent programming models unless they are recoded to explicitly serialize access to shared data.

The remainder of this paper is organized as follows: Section 2 discusses related research and our extensions to the current set of results. Section 3 describes our methodology and workload. Section 4 discusses how reference behavior is characterized and why such characterization efforts are useful. Section 5 presents our results. Section 6 summarizes these results and discusses how they might be used to improve current cache consistency protocols. Some of these improvements are analyzed and evaluated via trace-driven simulation in a companion paper [Gee93].

2. Background

A great deal of research currently exists on the topic of cache consistency. Several studies have previously looked at the memory reference characteristics of multiprocessor applications, while other studies have looked at consistency protocol performance, memory consistency models, and the performance benefits of relaxing consistency. This section briefly summarizes results from a number of these studies and discusses the contributions of our work; a somewhat more extensive summary appears in [Gee93b].

Darema-Rogers, et. al. [Dare87], examined the memory reference behavior of three parallel scientific applications using the IBM PSIMUL tool to simulate an eight-processor system. In all three programs, less than 25% of all data references are to shared data. Although most references are to private data, references to shared data are bursty in nature and require some form of caching to reduce contention for the shared memory.

Agarwal and Gupta [Agar88a] analyzed multiprocessor traces of three parallel scalar programs running on a 4-processor VAX 8350. Two of the traces were collected from CAD applications; the other was collected from a parallel implementation of the OPS5 programming language. References to shared data are approximately 25% of all data references. Agarwal and Gupta introduced the notion of *remote* and *local* references (using the terminology *pinging* and *clinging*), where a remote reference occurs when a processor accessing shared data differs from the last processor to access the data, and a local reference occurs when these processors are identical. Agarwal and Gupta found there is little processor locality for shared data, except for reference runs containing writes, for which locality was moderate.

Eggers and Katz [Egge88] analyzed the reference characteristics of four multiprocessor CAD traces. Roughly 25 to 35% of all data references are to shared data. They found that sharing behavior differed considerably among the traces. Eggers and Katz concluded that write-invalidate protocols should provide superior performance, since (a) write runs can be long, and (b) few processors reread data after being invalidated. Timing simulations to confirm this hypothesis were inconclusive.

Baylor and Rathi [Baly89], like Darema-Rogers et. al., analyzed traces of parallel scientific applications gathered using the IBM PSIMUL tool. For this study the PSIMUL system was configured for 64 processors. As in other studies, references to shared data are roughly 25% of all data references. Baylor and Rathi measured the average time ownership (in cycles) of cache lines, where ownership begins when a processor writes a line and ends when a different processor references the line. Ownership times were found to decrease with increasing line size, (the *false sharing* problem). Lines containing synchronization variables are shared by nearly all processors and owned for very short times.

Weber and Gupta [Webe89,Gupt92], using traces of up to 32 processors, examined cache invalidation patterns to evaluate the scalability of directory-based protocols. Five multiprocessor traces were examined, from areas such as operations research, computational chemistry, logic simulation, and computer-aided design. The average number of invalidations per shared write is often less than one, even in 16 and 32-processor traces.

Vashaw [Vash93] used a hardware monitor to collect large address trace samples from an 8-processor Encore Multimax. In addition to their length, these traces are unique in that they include both supervisor and user state references. The study analyzed the traces, and found that supervisor references generate much larger cache and tlb miss rates than user references. Implications for protocol performance are less clear, as supervisor references contain lesser amounts of sharing and fewer hot spots relative to user references.

In addition to these studies, there are studies addressing the performance of snooping-cache consistency protocols. Archibald and Baer [Arch86b] simulated a multiprocessor system driven by synthetic reference streams, and reported that update-based protocols outperformed invalidate-based protocols. Studies using real trace data were less conclusive [Agar88a,Egge88], finding neither type of protocol to consistently outperform the other. Eggers and Katz

[Egge89b] evaluated the impact of varying cache and block size on multiprocessor cache miss rate and bus utilization, and found that increased invalidation misses due to sharing in larger cache and/or block size often limit performance. Another study by the same authors [Egge89c] examined two extensions to adapt invalidate and update-based protocols to varying reference patterns, neither of which consistently improved performance. Finally, Eggers and Jeremiassen [Egge90] attempted to improve the performance of snooping-cache systems via the elimination of false-sharing.

Directory-based protocols have more recently become the focus of much research, as they offer improved scalability over bus-based protocols and can function in general interconnection networks. Early research by Agarwal, et al. [Agar88b] found that directory-based protocols were competitive with bus-based protocols in terms of performance, and far superior in terms of scalability. O'Krafka and Newton [Okra90] evaluated two space-efficient directory protocols using a detailed Motorola 68020-based timing simulator. They found that caching recently-used directory entries, rather than allocating entries for each block of memory, yields nearly the performance of a full-map directory protocol with only a fraction of the directory overhead. Several other studies have also looked at the problem of reducing directory storage overhead [Broo90,Chai90,Chai91,Gupt90]. Currently we know of at least two research efforts that are actively studying directory-based protocols through hardware implementation [Chai91,Leno90].

Finally, there is the important issue of sequential vs. weak consistency. *Sequential consistency*, as defined by Lamport [Lamp79], requires the result of any parallel execution to be the same as if the operations of all processors were executed in some interleaved order, with the operations of individual processors appearing in program order. More generally, sequential consistency requires that all processors observe the same ordering of memory references, with no processor allowed to execute any of its own memory references out-of-order. These conditions are fairly simple to uphold in small-scale, bus-based systems without write-buffering, as the shared bus insures that all processors observe the effects of a memory reference at the same time.

Weakly-consistent systems [Dubo86], on the other hand, require that memory be consistent only at synchronization points. Within any processor, non-synchronization accesses can be reordered, buffered, and pipelined to improve performance. Synchronization references must be placed around critical regions of a program to serialize access to shared data and force all outstanding references to complete before proceeding further. Recent studies [Zuck92,Ghar91,Tore90] have evaluated the benefits of relaxing consistency and found that performance can improve by 10 to 40 percent. The main disadvantages of relaxing consistency are a more complicated programming model and the potential high cost of recoding existing sequentially-consistent applications to execute correctly.

As mentioned in the Introduction, this paper builds upon earlier research by (a) analyzing reference behavior across many more multiprocessor traces than previously examined, including samples from production vector applications, (b) carrying out a highly detailed analysis with some new metrics, and (c) observing reference and sharing behavior over a range of block sizes. We use our results to predict which protocols are best for different workloads, to propose improvements to current protocols, and to determine whether protocols which detect and correct consistency errors may be a viable alternative to protocols which prevent such errors from occurring in the first place.

3. Methodology

Our work is based primarily on trace-driven simulation. The traces analyzed in this study originate from three sources: 4-processor Ardent Titan [Died88] traces gathered at Ardent Computer, 16-processor VAX T-bit traces collected at Stanford, and 64-processor IBM 370 and Encore Multimax traces used at MIT. The traces from Stanford and MIT have been used in previous studies [Webe89,Chai90], while we collected the Ardent traces to use in this effort.

The Ardent traces were generated using an object code profiler (similar to the MIPS *pixie* facility) to instrument compiled programs. The instrumented object code executes and deposits memory reference addresses from all four Ardent processors into a single, shared trace file. The file is protected with locks to allow only one process to access this file at any time. This tracing method is quite accurate [Stun91], as inherent synchronization within an application ensures that the interleaving of references by different processors can only be affected between synchronization points. Since all possible interleavings are allowed, provided individual processor references appear in program order, our traces represent at least one valid ordering of actual program execution. A recent study [Kold91] confirms the validity of traces generated in a similar manner.

The traces from Stanford [Webe89] were gathered using the trap-bit tracing method on VAX-series computers. 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 for 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 traces *weather64*, *fft64*, and *simple64* used at MIT were generated from uniprocessor traces using a postmortem scheduling technique developed at IBM [Kuma89]; the machine traced was the IBM 370. 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 some number of processors. This form of tracing is somewhat prone to distortion because (a) the trace originates from a uniprocessor system, and (b) the scheduling of work on processors is somewhat arbitrary. In this case however, the traces are of scientific programs that usually perform series of operations on large data structures. Partitioning these data structures into a number of sub-units and creating tasks for each sub-unit corresponds roughly to how a real multiprocessor would execute such programs in parallel.

The *speech64* trace was generated at MIT 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, although the Ardent method inserts tracing code after link time, while this scheme operates at compile time. The compiler-based scheme executes on an Encore Multimax under a modified Mul-T (a variant of Multilisp) programming environment. This environment allows an arbitrary number of processes to be traced, although instruction references currently are not traceable.

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
simple64	IBM 370	Fortran	2-D hydrodynamic behavior of fluids
weather64	IBM 370	Fortran	finite difference weather analysis
speech64	Encore Multimax	Mul-T	lexical decoding of spoken language

Table 1: Trace application summary

Table 1 lists and gives a short description of each trace program. The Ardent Titan traces come entirely from a production scientific workload, as the Titan is a commercial vector machine with multiple processors. Many of these same Ardent applications were used to evaluate vector cache performance in two other studies [Gee92a,Gee92b], although the traces used for that purpose were uniprocessor versions. The T-bit programs *p-thor* and *locus route* are CAD applications performing logic simulation and VLSI routing, respectively, while *mp3d* is a 3-dimensional particle simulator. All three applications were developed as part of various research projects at Stanford University, and continue to be used in research environments. The *fft64*, *weather64*, and *simple64* traces from MIT were taken from scientific applications, while *speech64* is a trace of a research program developed at MIT to perform the lexical decoding of a spoken language.

Reference Characteristics										
Trace	Refs	Inst	Locks	Data	Priv Read	Priv Write	Total Priv	Shd Read	Shd Write	Total Shd
	(M)	*fraction of all refs*			fraction of data refs					
arc3d	20.0	0.652	0.002	0.346	0.147	0.150	0.297	0.539	0.164	0.703
bmk1	20.0	0.740	0.000	0.260	0.457	0.341	0.798	0.198	0.004	0.202
bmk11a	20.0	0.550	0.003	0.447	0.065	0.031	0.096	0.591	0.313	0.904
flo82	20.0	0.626	0.004	0.370	0.113	0.095	0.208	0.585	0.207	0.792
lapack	20.0	0.760	0.001	0.239	0.443	0.150	0.593	0.208	0.200	0.408
simple	20.0	0.649	0.003	0.348	0.138	0.063	0.201	0.609	0.190	0.799
wake	20.0	0.600	0.001	0.399	0.195	0.195	0.390	0.465	0.145	0.610
mp3d	7.0	0.607	0.000	0.393	0.656	0.074	0.730	0.186	0.084	0.270
p-thor	7.1	0.497	0.000	0.503	0.623	0.204	0.827	0.161	0.012	0.173
locus route	7.7	0.514	0.000	0.486	0.691	0.243	0.934	0.064	0.002	0.066
fft64	7.4	0.420	0.002	0.578	0.560	0.204	0.764	0.118	0.118	0.236
simple64	26.3	0.437	0.054	0.509	0.462	0.238	0.700	0.269	0.031	0.300
weather64	31.4	0.430	0.079	0.491	0.805	0.156	0.961	0.039	0.000	0.039
speech64	11.8	0.000	0.000	1.000	0.342	0.201	0.543	0.441	0.016	0.457
Arithmetic Averages										
Ardent	20.0	0.654	0.002	0.344	0.195	0.134	0.329	0.485	0.186	0.671
VAX T-bit	7.3	0.538	0.000	0.462	0.657	0.183	0.840	0.132	0.028	0.160
MIT	19.2	0.432	0.060	0.508	0.635	0.195	0.830	0.142	0.028	0.170

Table 2: Reference characteristics

This table shows the total number of references in each trace, along with the 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. Locks are not present in the VAX T-bit traces. MIT averages do not include *speech64*, since limitations in the Mul-T tracing environment preclude the tracing of instructions.

Table 2 separates the total number of memory references in each trace into the following categories: instruction and lock references, reads and writes to *private data*, and reads and writes to *shared data*. Here *shared data* is defined as global data referenced by more than one processor during the course of the trace. Global data used by only one processor and data explicitly defined as local to each process is considered *private data*. In the IBM traces *simple64* and *weather64*, the large fraction of lock references is due to a combination of (a) naive synchronization techniques (all processors often spin on one lock), and (b) the large number of processors sharing locks in these traces.

The Ardent traces assume a 64-bit memory interface; thus some of the Ardent data references in Table 2 are to eight-byte, double-precision quantities. These references will be split into two four-byte halves when block sizes smaller than eight bytes are analyzed, and left as a single reference for larger block sizes. [Note: due to the 64-bit memory interface, 32-bit Ardent instructions are normally fetched two at a time, while Ardent data references are fetched one at a time. This implementation artifact does not affect our research results, as we only examine sharing patterns in data references. Instructions are shared with no cache consistency overhead].

Table 3 separates the amount of referenced address space into instruction and data space, and further separates data space into (a) private read, (b) private write, (c) shared read, and (d) shared write categories. Private space is read and written by only one processor during the course of the trace. Shared read space is data unmodified during the trace and used by two or more processors. Shared write space is data modified during the trace and used by two or more processors (although only one of the processors may have performed all modifications to that data). These address space statistics are based on a four-byte block size. Double-precision Ardent data references were split into two four-byte halves when estimating address space size.

From Table 2 we see that the Ardent vectorized workload contains a much larger fraction of shared references compared to the VAX T-bit or MIT scalar workloads. References to shared data make up some 70% of all data references, and nearly 30% of all references. In contrast, shared data references in the VAX T-bit and MIT traces are only 16% to 24% of all data references and 7% to 14% of all references. The T-bit and MIT numbers are similar to observations from other studies described earlier. Read sharing is also more prevalent in the T-bit and MIT workloads, as the ratio of reads to writes of shared data is larger relative to the same ratio in the Ardent workload. In addition, Table 3 shows that a larger fraction of shared data space in the T-bit and MIT workloads is shared in a read-only manner.

We believe that the heavy presence of sharing in the Ardent workload is due to inherent differences between vector and scalar workloads. Vectorized applications usually operate on large data structures which are stored in and referenced from main memory. Since these data structures are referenced repeatedly during the duration of a program, and the scheduling of work on vector processors is usually independent of which processor last used the data (no cache effects were considered in the scheduling algorithm), much of the data region eventually becomes shared.

Address Space Breakdown							
Trace	Total Kbytes	Inst Kbytes	Data Kbytes	*Percent (%) of Data Bytes*			
				Private Read	Private Write	Shared Read	Shared Write
arc3d	1712.3	64.4	1647.9	0.1	36.1	2.4	61.4
bmk1	110.2	3.8	106.4	0.6	77.5	0.0	21.9
bmk11a	364.4	11.4	353.0	0.0	11.9	0.0	88.1
flo82	240.4	82.7	157.7	2.4	30.0	3.3	64.3
lapack	4415.3	0.8	4414.5	0.0	25.9	0.0	74.1
simple	228.7	51.7	177.0	0.3	4.9	5.4	89.4
wake	183.3	18.3	165.0	0.1	3.3	5.1	91.5
mp3d	449.1	3.1	446.0	43.7	46.8	1.4	8.1
p-thor	435.5	3.7	431.8	48.9	27.5	12.0	11.6
locus route	174.8	7.1	167.7	20.8	53.7	17.3	8.2
fft64	132.5	2.7	129.8	0.5	2.4	0.0	97.1
simple64	1194.8	6.6	1188.2	0.4	3.3	5.3	91.0
speech64	479.9	0.0	479.9	2.3	49.4	36.8	11.5
weather64	2518.1	2.2	2515.9	23.9	64.3	11.8	0.0
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	661.4	0.0	656.9	1.7	12.6	0.0	0.0
Arithmetic Averages							
Ardent	1036.1	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	1081.3	2.9	1078.5	14.3	44.0	12.4	29.3

Table 3: Address space breakdown in kilobytes

This table lists total, instruction, and data address space in kilobytes. Data space is also broken down (in percent) into Private Read, Private Write, Shared Read, and Shared Write categories. Shared data is data referenced by more than one processor during the course of the trace. Write data is data written during the trace. Shared Write data is shared data written by at least one processor during the trace. The address space was measured using a block size of 4 bytes.

Another possibility is that the longer length of the Ardent traces, relative to the number of processors in the trace, was a factor in the increased sharing. Each Ardent trace contains 5 million references per processor, compared to half a million references per processor for the T-bit and MIT workloads. The longer Ardent trace length (on a per-processor basis) may increase the probability that a block becomes shared. We investigated this theory by measuring the fraction of data references to shared space as a function of the number of data references examined. As Table 4 shows, the fraction of data that is shared increases with trace length, but even for the first trace segment, sharing is at a much higher level than for the other traces.

Sharing in the Ardent Traces as a Function of Trace Length								
Data Refs (millions)	Fraction of Data References to Shared Data							
	arc3d	bmk1	bmk11a	flo82	lapack	simple	wake	AVG
0.5	0.396	0.190	0.900	0.302	0.558	0.749	0.541	0.519
1.0	0.366	0.182	0.920	0.601	0.557	0.755	0.579	0.566
1.5	0.364	0.178	0.920	0.682	0.555	0.766	0.589	0.579
2.0	0.389	0.176	0.920	0.716	0.555	0.774	0.599	0.590
2.5	0.459	0.174	0.920	0.734	0.556	0.775	0.606	0.603
3.0	0.507	0.173	0.920	0.751	0.556	0.775	0.602	0.612
3.5	0.554	0.172	0.920	0.761	0.556	0.772	0.603	0.620
4.0	0.590	0.172	0.919	0.767	0.556	0.774	0.604	0.626
4.5	0.619	0.172	0.919	0.773	0.555	0.772	0.607	0.631
5.0	0.643	0.172	0.919	0.775	0.556	0.773	0.609	0.635

Table 4: Fraction of data references to shared data vs. trace length

4. Characterization Metrics and Applications

We characterize the reference behavior in multiprocessor applications to evaluate how these applications may perform under a given cache-consistency protocol, and to collect information that may lead to improvements on existing protocols. Through characterization, we can also compare reference behavior across different workloads and determine the best choice of protocol for a given workload. Analyzing traces in this manner may yield more insight into protocol performance than a straightforward and time-consuming simulation of the entire protocol space.

We characterize memory reference behavior by examining data reference streams to shared blocks, examples of which are shown in Figure 1. Each row represents the sequence of processors referencing a specific shared data item, with write references specified in boldface. Note that the number of processors actively sharing data varies from as little as two processors to as many processors as are represented in the trace.

Our characterization process consists of the following steps:

[1] We examine *processor locality* by measuring the number of consecutive data references that a processor makes to a block of shared data. The processor locality present in a workload is a key factor in choosing between invalidate or update-based protocols for that workload.

[2] We examine *temporal locality* by measuring the times between two successive references to a shared data block. Temporal locality indicates whether a block is likely to be reused before being replaced in a cache. In this study, we also use temporal measurements to evaluate the likelihood of consistency errors, i.e. writes followed shortly by a read from a different processor. This should shed some insight as to whether strict

arc3d **0 3 2 1 0 3 2 1 0 0 0 1 2 3 0 1 3 2 2 1 0 3 2 0 3 1 2 3 1 2**

bmk11a **3 0 1 2 2 1 0 3 0 2 3 1 2 0 3 1 3 0 2 0 3 1 2 0 3 1 2 1 0 3**

lapack **1 0 3 2 3 1 0 3 2 1 0 2 3 0 1 3 2 1 0 3 0 1 2 1 3 2 3 0 1 2**

wthr64 **0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29**

arc3d 0 0 2 2 3 3 1 1 0 0 2 2 1 1 3 3 0 0 0 0 3 3 1 1 0 0 2 2 3 3

bmk11a 2 3 1 0 1 0 2 3 3 1 0 2 3 1 0 2 3 1 0 2 3 1 0 3 2 1 0 2 3 1

flo82 0 2 1 3 3 0 2 1 0 1 3 2 1 3 2 0 0 2 1 3 1 3 0 2 0 2 1 3 3 0

p-thor 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 0 0

mp3d 13 13 14 8 8 12 12 6 6 7 7 9 9 10 10 11 11 4 4 10 10 0 0 13 13 3 3 6 6 11

locusr 5 5 8 8 8 8 8 14 14 14 14 14 14 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

spch64 3 4 2 3 1 2 0 1 0 62 61 62 60 61 59 60 58 59 57 58 56 57 55 56 54 55 53 54 52 53

arc3d 1 2 0 3 **0** 3 2 1 0 **0** 2 0 3 1 **0** 3 2 0 1 **0** 1 0 3 2 **0** **0** 0 3 0 1

flo82 **3** **3** 0 0 0 0 0 0 0 0 0 0 3 3 0 0 0 0 2 2 **2** **2** 2 2 0 0 2 2 2 3

lapack **1** 0 **0** 0 **0** 3 **3** 3 **3** 0 **0** 0 **0** 3 **3** 2 **2** 3 **3** 0 **0** 1 **1** 2 **2** 0 **0** 1 **1** 1

p-thor 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 **0** 1 2 3 4 5 6 7 8 9 10 11 12 13

mp3d 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 **0** 1 **1** 2 **2** 3 **3** 4 **4** 5 **5** 6 **6**

locusr 5 **5** 5 5 5 **5** 5 5 5 **5** 5 5 5 **5** 5 5 5 **5** 5 5 5 **5** 5 5 5 **5** 5 5 5 **5**

spch64 0 **0** 0 **0** 0 **0** 0 **0** 0 **0** 0 **0** 62 0 **0** 0 **0** 61 0 **0** 0 **0** 60 0 **0** 0 **0** 59 0 **0**

Figure 1: Sample reference patterns to shared data
(numbers correspond to processor IDs; write references are in boldface)

prevention of such errors is necessary, or whether they are infrequent enough that detection and ‘fix-up’ would make sense.

[3] We measure the number of processors *contending* for shared data by (a) counting the number of data copies invalidated on each write to shared data, and (b) by measuring the number of processors which subsequently reread this data. This estimates to some degree the negative performance impact of invalidate-based protocols, and also estimates the number of directory entries needed in a directory-based protocol.

[4] We also attempt to describe the reference process using Markov chains. Several Markov models are developed, and transition probabilities between Markov states were measured directly from the traces.

Unlike most prior studies, we explicitly carried out our analysis over a range of block sizes. Measurements were taken from each trace over block sizes ranging from 4 to 64 bytes. Ardent double-precision data references are split into two 4-byte references when block size is 4 bytes, and are left as one reference for larger block sizes.

Due to the large number of traces that have been analyzed, we organized our results into different workloads to provide average results for a given program mix. Three different workloads were constructed from (a) the Ardent traces, (b) the VAX T-bit traces, and (c) the IBM 370 and Encore Multimax traces from MIT. Results presented for a workload represent averages over the data from each trace, with the data from each trace weighted equally.

5. Results

5.1. Processor Locality

We define *processor locality* as the tendency for shared data to be used by only one processor over periods of time. For applications with large amounts of processor locality, an invalidate-based protocol should minimize consistency overhead by purging data from other caches, which are unlikely to need their copies in the near future. Update-based protocols are tuned for weak processor locality, as valid copies of modified data are maintained in multiple processors in the assumption that any processor is equally likely to initiate the next reference to that data.

We measure processor locality using *reference runs*, strings of consecutive references to a shared data block by one processor without any interleaved references to that block by another processor. Reference runs can be divided into two classes: (1) *read runs*, which consist solely of read references, and (2) *read/write runs*, which contain at least one write. Read runs actually have little significance to protocol performance because all protocols, with the exception of directory schemes with limited pointer entries, allow any number of processors to share read-only data with no overhead. In contrast, writes can cause considerable overhead, since a write may require invalidates or updates of remote copies of data; invalidated data may later need to be reread. Thus read/write run durations will have a strong impact on protocol performance.

In addition to measuring read and read/write run length, we also measure *write run length* [Egge88], which is basically the number of writes within each read/write run. If write runs are short, data tends to bounce between processors when invalidates are used to maintain consistency. This *ping-ponging* effect results in poor performance, since virtually each write will generate a cache miss. Update-based protocols handle short write runs more efficiently, but are inefficient on long write runs, since many updates are made that are never used by other processors.

Table 5 lists the number of read, read/write, and total runs in each workload as a function of block size. Note that there is a write run for every read/write run in a workload, and that the number of runs varies with block size. Figure 2 displays average run lengths for all workloads as a function of block size. The four plots in Figure 2 show average lengths for (a) read runs, (b) read/write runs, (c) write runs, and (d) total (read + read/write) runs.

Number of Runs in a Workload				
Workload	Block Size	Read	Read/Write	Total
Ardent (7 traces)	4	14,777,416	10,6693,356	25,470,772
"	8	7,353,199	5,623,328	12,976,527
"	16	4,966,168	3,238,862	8,205,010
"	32	3,738,987	2,001,898	5,740,885
"	64	3,124,931	1,375,252	4,500,183
T-bit (3 traces)	4	517,393	237,629	755,022
"	8	691,753	267,653	959,406
"	16	637,456	279,721	917,177
"	32	624,233	287,754	911,987
"	64	622,703	291,938	914,641
MIT (4 traces)	4	9,727,844	399,789	10,127,633
"	8	9,887,770	802,535	10,690,305
"	16	12,726,799	1,411,305	14,138,104
"	32	13,061,860	1,589,247	14,651,107
"	64	13,218,865	1,689,494	14,908,359

Table 5: Number of runs in each workload vs. block size

For a block size of 4 bytes, average run lengths for all workloads agree fairly well. For larger block sizes, average run lengths for the T-bit and MIT workloads remain fairly constant, while average run lengths for the Ardent vector workload increase considerably. The rate of increase is largest for write and read/write runs, as processor locality is clearly stronger in write-shared blocks. In [Gee92b] we found that vector workloads traced on Ardent machines contain large amounts of spatial locality. As we increase block size, this spatial locality leads to longer run lengths to shared data. These results suggest that for the Ardent workload, the best results would be obtained from an invalidate-based protocol and a large block size. Conversely, the T-bit and MIT workloads should be paired with an update-based protocol and a small block size.

For comparison, both Eggers and Katz [Egge88] and Agarwal and Gupta [Agar88a] measured average write run lengths for a block size of 4 bytes. At that block size, average write run lengths for all three of our workloads are fairly short (1-2 writes) and in good agreement with [Agar88a]. Eggers and Katz found average write runs to also be short in two of their traces but much longer (5-6 writes) in two others. Agarwal and Gupta also measured average lengths for read/write runs and for all run types. Average read/write run lengths range from 4 to 9 references, and the average length for all runs is roughly 2 references. Both figures are in line with our observed results, which is unsurprising given that two of the three applications used in [Agar88a] are in our T-bit applications workload and are traced on the same VAX architecture, although tracing methods and the number of processors differ.

Distributions for the various types of reference run lengths are shown in Figures 3 through 5. Each figure contains results for a different workload. Curves in the figures are parameterized by block size. In Figure 3, we observe the shift in Ardent toward longer run lengths with increasing block size. Across all block sizes, a significant fraction (greater than 20%) of Ardent write runs contain only one write. These runs typically represent events where processors reach a synchronization barrier and increment or decrement a global variable to signal their arrival.

Distributions for the scalar VAX T-bit and MIT workloads, in Figures 4 and 5, are much less affected by increasing block size. Write run distributions in the T-bit workload are uniformly short. Write run distributions in the MIT workload are somewhat longer compared to the T-bit workload, but decrease in length for block sizes larger than eight bytes due to false sharing.

5.2. Temporal Locality

Temporal locality refers to the property that data items currently being referenced have a high probability of being referenced again in the near future. Temporal locality arises from sources such as program loops, stack variables, and often-used global data. In multiprocessor systems, we wish to know whether shared data references contain temporal locality, and especially whether shared references by *different processors* occur over short time intervals. If several processors are accessing and modifying shared data within short periods of time, the resulting bus traffic to maintain consistent caches may lead to poor performance.

In this section we use *reference intervals* to characterize the temporal locality within shared data references. Reference intervals are the times between two successive data references to a shared block, where each data reference in the trace is counted as a unit of time. We measure various types of reference intervals: (1) *local reference intervals*, the times between two successive references to shared data made by the same processor, (2) *remote reference intervals*, the times between two successive references to shared data where the referencing processors differ, and (3) *remote write* intervals, the times between the *last write* of a read/write run and the first use of this result by a different processor.

In a weakly-consistent system, consistency violations can occur if a processor is allowed to buffer its own writes (making them visible locally), and these writes do not propagate to other processors within the nominal remote write interval. If remote write intervals are large enough to make such violations very infrequent, then multiprocessor designs may wish to focus on detecting and correcting consistency errors, rather than strictly preventing such errors. Prevention requires processor stalls to insure that shared-memory operations have propagated to all processors, and may degrade performance unnecessarily if remote write intervals are large, or if synchronization barriers are normally present to enforce correctness. To factor in the effect of barriers, we measured remote write intervals for the Ardent and MIT traces a second time, assigning an *infinite* time to the interval if the write and read by the two processors are separated by synchronization requests. We could not do the same for the T-bit applications due to the absence of locks in these traces. Table 6 provides some statistics on the use of locks in the Ardent and MIT traces.

Remote Requests to Shared Modified Data		
Program	Preceded by Locks	
	Num	Pct.
arc3d	867,096	95.2
bmk1	8,967	80.5
bmk11a	3,472,748	84.9
flo82	974,503	92.5
lapack	1,458,930	88.8
simple	1,418,006	88.4
wake	1,092,723	99.0
fft64	32,257	99.6
simp64	335,874	100.0
speech64	0	0.0

Table 6: For the Ardent and MIT traces, the number and percentage of remote references to dirty data preceded by a lock reference.

Figures 6 through 8 show the cumulative fraction of reference intervals as a function of interval length for our three workloads. A point (X,Y) on a graph indicates that a fraction Y of all reference intervals are less than X references. Measurements were taken only for shared data, and represent the number of data references within a trace between successive references to a shared data block. Table 7 lists mean and median reference intervals for all three workloads across all block sizes. We factor *infinite* intervals into the medians in Table 7, but do not include them in the means.

The Ardent results in Figure 6 show a strong presence of temporal locality in local references, especially for larger block sizes. Ardent remote reference intervals are much larger than local reference intervals, and decrease far more slowly as block size increases. The T-bit results in Figure 7 do not vary much at all with changing block size, but like the Ardent results, remote reference intervals tend to be larger than local reference intervals. This last result differs from results in [Agar88a] for two of the same programs, although different tracing techniques and the larger number of processors in our traces may be responsible for the differences. In the MIT traces, remote reference intervals are as small or smaller than local reference intervals, which does agree with the data in [Agar88a]. In addition, *remote write intervals* in the MIT workload decrease sharply as block size increases, a likely result of false sharing.

As noted earlier, significant performance gains are possible if a weakly consistent programming model is implemented rather than a strongly consistent one. We therefore measured the frequency with which data written by one processor was then read very soon after by another. If this occurrence was infrequent enough, then an implementation which detected consistency violations and ‘‘repaired’’ them might on the average improve performance. ‘‘Repair’’ would require saving enough state to back the processors up to the state prior to the consistency error; a strongly consistent execution mode would then be used to execute past the error point. Although remote write intervals tend to be larger than other intervals within a workload, it is always the case that a significant fraction (10 to 20 percent) of all remote write intervals are very short, on the order of 10 data references or less. After adjusting remote write intervals to account for synchronization references, the fraction of short remote write intervals in the Ardent traces drops dramatically. However, the distribution of remote write intervals in the MIT traces remains relatively constant. The observed frequency of short remote write intervals seems to us to be far too high to support the use of a ‘‘repair’’ strategy.

Reference Interval Summary by Workload

Workload	Block Size	Local		Remote		Remote Write	
		Mean	Median	Mean	Median	Mean	Median
Ardent	4	23465	420	48953	7729	371	>500000
Ardent	8	9640	51	39114	8260	370	>500000
Ardent	16	3871	1	31648	4028	223	>500000
Ardent	32	1796	1	23349	574	181	>500000
Ardent	64	895	1	15522	98	147	>500000
T-bit	4	13413	38	43537	466	84228	44282
T-bit	8	9891	23	28783	175	59972	18276
T-bit	16	4965	31	19770	154	40402	8367
T-bit	32	2857	56	12094	126	23213	4069
T-bit	64	1781	71	8813	77	17398	2397
MIT	4	11404	569	8939	323	43586	>500000
MIT	8	1641	148	10060	224	33943	136886
MIT	16	732	73	3981	67	7021	60
MIT	32	483	72	2140	40	3042	40
MIT	64	361	72	1412	38	1871	39

Table 7: Reference Interval Summary by Workload

5.3. Contention for Shared Data

Several processors *contend* for shared, writable data when they all request contemporaneous access to this data. Contention usually occurs for global data structures and work queues shared by all processors, or for the locks which serialize accesses to such data.

The presence of large amounts of contention can have serious negative effects on consistency protocol performance. Invalidate-based protocols may perform particularly poorly, as these protocols purge data from other caches when writes occur. These other caches may immediately request fresh copies from the writing processor, saturating the interconnection network (e.g. a bus) with block requests, and causing delays in processing. Similar problems can occur with update protocols, for which the interconnection network may be saturated by updates.

To quantify contention for shared, writable data, we simulated the actions of the Berkeley invalidate-based protocol [Katz85], using an infinite cache size. During the simulation, we measured (1) the average number of cache copies invalidated per shared write [Webe89], and (2) the average number of processors that reread previously invalidated data back into their caches between external write runs to that data (otherwise known as *external rereads* [Egge88]). Large numbers of external rereads corresponds to the type of behavior where one processor writes data (invalidating it in all other processors), and many processors immediately read back the data.

Figure 9 shows both the average number of invalidations per shared write and the average number of external rereads. Both are plotted for each workload as a function of block size. In general, there are very few invalidates per shared write. Increasing block size results in only minor increases from false sharing in the T-bit and MIT workloads. In the Ardent workload, the average number of invalidates per shared write is a rapidly decreasing function of block size. This behavior is due to the high spatial and processor locality in this workload. As block size increases, each invalidate allows more writes to that block by the same processor to complete without need for further invalidations.

The results for external rereads show a similar pattern. As block size increases, there is some increase in contention in the T-bit and MIT workloads, but a slight decrease in contention in the Ardent workload. Overall, the average number of rereads is quite low for all workloads and block sizes.

Our results confirm work previously carried out by [Webe89] and [Egge88] for a four-byte block size. Using the same T-bit traces as used in [Webe89], we observed the same low average number of invalidates per shared write. At the same time, the average number of external rereads for our three workloads are as low as measurements from [Egge88]. While these results may suggest low levels of contention, results presented later in this paper, and in [Gee93a] show that the contention levels are still unacceptably high.

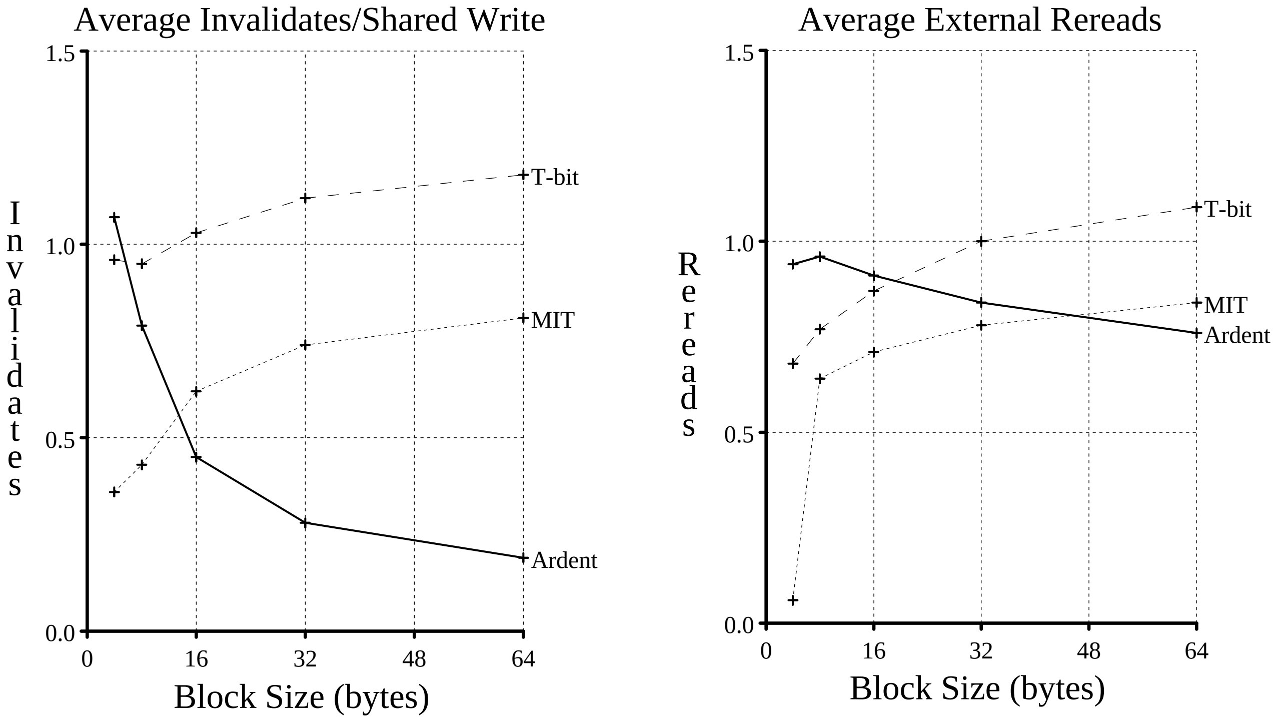
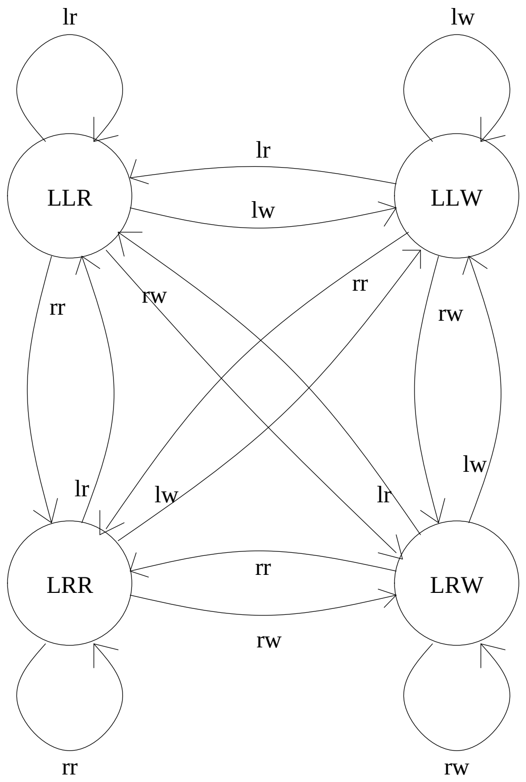


Figure 9: Average invalidates per write and external rereads

5.4. Markov Models

So far, we have examined processor locality, temporal locality, and contention within our multiprocessor address traces. In this final phase of our analysis, we attempt to describe the reference process using Markov chains. This methodology has been used extensively in past studies on page and file reference patterns [Lewi73, Spir77, Kure88]. After defining the models, we analyze the traces using a separate Markov chain to model the behavior of each shared data block in a trace.

Our first Markov model is shown in Figure 10, and contains four states specifying whether the last reference to a shared data block was a *local read*, *local write*, *remote read*, or *remote write*. This model assumes that there is only one copy of the data, with only one processor accessing it locally, and that the cache size is infinite. The states in the model are labeled LLR (Last reference a Local Read), LLW (Last reference a Local Write), LRR (Last reference a Remote Read), and LRW (Last reference a Remote Write). Remote reads and writes cause transitions to states LRR and LRW, while local reads and writes cause transitions to states LLR and LLW.



LLR: last reference a local read　　lr: local read
LLW: last reference a local write　　lw: local write
LRR: last reference a remote read　　rr: remote read
LRW: last reference a remote write　　rw: remote write

Figure 10: First Markov model

Transition probabilities measured for block sizes of 4 to 64 bytes are listed in Tables 8 through 10. Increasing block size affects all workloads, although the Ardent results are more noticeably affected due to the strong spatial locality in the Ardent applications.

The Ardent workload is characterized mainly by large amounts of processor locality. Transition probabilities from local states back to local states increase greatly with block size, and approach 90 percent at a block size of 64 bytes. Transition probabilities from remote states to local states are also very large, although not quite as large as those from local states back to local states. Note that even 90% is a low number; it means that at least one reference out of ten (>10%) is remote, and that a bus transaction may be required. This is like having a 10% miss ratio, which is extremely high for a large cache.

The T-bit workload also contains processor locality, provided the local processor is currently reading the data. After a local processor has written shared data (i.e. moves to the local write state), the next reference to that block will often be a remote read. This suggests a producer-consumer relationship between different processors. There are also fairly large diagonal probabilities for state LRR, indicating the presence of fine-grain read sharing where a remote read is typically followed by another remote read. Diagonal probabilities for state LRW are almost as large, which suggests the presence of surprising amounts of fine-grain write sharing.

Fine-grained read sharing is even stronger in the MIT workload, as diagonal probabilities for state LRR are well over 80 percent. Fine-grain write sharing is also present, but is mainly a product of false sharing, as diagonal probabilities for state LRW are not quite as large for smaller block sizes as observed in larger block sizes. We note that local writes are very often followed by local, rather than remote reads. This reference pattern does not follow the producer-consumer paradigm observed in the Ardent and T-bit workloads.

Stationary probabilities, representing the probability that a shared data block is in a specific state, are listed in Table 11. In the Ardent workload, stationary probabilities for the local states increase with block size at the expense of the remote states. Remote state probabilities are larger only when the block size is four bytes. T-bit and MIT stationary probabilities are less affected by block size. Blocks in the T-bit workload are most likely to be in the remote read state, although the local read state is just as likely when block size reaches 64 bytes. Blocks in the MIT workload spend the vast majority of their time in the remote read state. For all workloads, stationary probabilities for state LRW are consistently low, as reference runs rarely begin with a write.

For the MIT workload (Table 10), we notice a strange trend in the stationary probabilities for state LLR. They peak at a block size of 8 bytes, drop when block size is increased to 16 bytes, and then increase again for larger block sizes. We believe that there are two conflicting processes at work here: (a) false sharing, which reduces the chance of a local read at block sizes beyond 8 bytes, and (b) spatial locality within the local processor, which improves the possibility of a local read at block sizes larger than 16 bytes.

Mean state durations are listed in Table 12. These durations represent the average time, in references to that block, that a block spends in a particular state. Ardent state durations for local states greatly increase with block size, while remote state durations are fairly constant. T-bit and MIT state durations are affected far less by varying block size. As the T-bit and MIT workloads share data on a much finer granularity, state durations for remote states LRR and LRW are higher relative to the Ardent workload.

In addition to this Markov model, we also looked at a slightly different Markov model which distinguishes between remote reads of (a) clean and (b) dirty blocks. Blocks become dirty after a processor writes the block, and dirty blocks become clean only after they are read by a different processor. By making a distinction between remote clean and remote dirty reads, we can provide separate analyses for read-shared and write-shared blocks. The new Markov model replaces state LRR of the first model with two new states: LRCR (Last reference a Remote Clean Read) and LRDR (Last reference a Remote Dirty Read). Remote reads which follow a read run are remote clean reads, while remote reads following a read/write run are remote dirty reads.

State transition probabilities are listed in Tables 13 through 15. Note that not all state transitions are possible. A remote read following any write is always a dirty remote read. Similarly, any remote read immediately following a remote read is always a clean remote read, since the previous remote read cleaned the block. This leaves only one state, LLR, from which a block can make a transition to either state LRCR or state LRDR. For all workloads, transition probabilities from state LLR to state LRCR are much higher than transition probabilities from LLR to state LRDR.

In the Ardent and T-bit workloads, write-shared blocks contain far more processor locality relative to read-shared blocks, as transition probabilities from state LRDR to other local states are much higher than corresponding probabilities from state LRCR. The same is not true for the MIT workload. From either of states LRDR and LRCR, the probability of a remote reference is far greater than the probability of a local reference. For all workloads, blocks in state LRCR are usually read-shared blocks referenced in a highly-interleaved manner, which is evident from the large diagonal probabilities for that state.

Table 16 lists stationary probabilities for the second Markov model. Stationary probabilities for state LRDR are low relative to state LRCR, because (a) state LRDR has a maximum duration of one reference, and (b) this state can only be entered after a read/write run. Stationary probabilities for state LRCR are extremely large in the MIT workload, moderately large in the T-bit workload, and quite small in the Ardent workload. Mean state durations are shown in Table 17. As mentioned above, the duration of state LRDR is always one reference, since a remote read to a dirty block cleans the block. For state LRCR, mean durations range from roughly two references in the Ardent workload, to up to five references in the T-bit workload, and up to ten references in the MIT workload.

6. Conclusions

6.1. Summary of Results

In this paper, we have used trace-driven simulation to study the reference behavior of three multiprocessor workloads: a vector-scientific workload running on a four-processor Ardent Titan, and two scalar workloads running on VAX, IBM, and Encore machines. This study provides a major contribution to this field simply by analyzing all of these traces, which represent a wide range of application programs *including an actual, production-quality vector workload.*

The other major contribution of this research is the detailed evaluation of sharing behavior in multiprocessor systems, carried out through a number of measurements across a range of block sizes. Our results show that both the amount and type of sharing present in the traces are extremely workload dependent. The Ardent vectorized workload makes many more references to shared data (70% of all data references) relative to the two scalar workloads (less than 30% of all data references). The Ardent workload also contains a much larger fraction of writes to shared data, while references to shared data in the T-bit and MIT workloads are predominantly reads.

Processor locality is very strong in the Ardent workload, due mainly to inherent spatial locality in vector applications coded with short strides, and partly to the fact that the applications ran on a small multiprocessor system (4 processors). Our two scalar workloads, running on larger 16 and 64-processor systems, contain far less spatial locality and share data on a much finer grain. Increasing block size in the scalar workloads only increases the chance that different processors will reference data in the same block due to false sharing. One commonality in all workloads is that processor locality is stronger for write-shared data than for read-shared data.

In the MIT traces, 10 to 20 percent of all shared data writes followed by reads from different processors are closely spaced in time (within 10 data references), without locks to serialize access to the data. These are examples of applications which may not be able to tolerate a weakening in the memory consistency model, despite its performance benefits, without large amounts of recoding. The Ardent applications, however, do frequently serialize access to shared data, and would benefit greatly from relaxing consistency. In fact, relaxing consistency on the Ardent applications should improve performance beyond earlier estimates [Zuck92, Ghar91, Tore90] because vectorized applications generate such large numbers of shared references.

Finally, two Markov models were developed to provide further insight into reference and sharing behavior. Transition probabilities were measured directly from the traces, and used to generate solutions to the Markov chain. The results provide further evidence that the Ardent traces contain large amounts of processor locality, while sharing behavior in the T-bit and MIT workloads consists mostly of fine-grain read-sharing.

6.2. Implications for Consistency Protocols

Based on this analysis, we can determine to some extent the types of consistency protocols and protocol features that should be matched to these different applications and workloads. The vectorized workload that we examined contained large amounts of processor locality and little contention for write-shared data, characteristics that are well-suited to invalidate-based protocols and large block sizes. The temporal characteristics of this workload also favor protocols with relaxed consistency. Our two scalar workloads contained less processor locality and more false sharing, characteristics that are better-suited to update-based protocols and smaller block sizes. Unlike the Ardent workload, the scalar workloads may require protocols which support sequential consistency. We carefully note that these conclusions are very much a function of how these particular programs were structured and coded, and should not be considered representative of all vector or scalar programs.

We note that the state transition probabilities for our Markov models show relatively high rates of transition from local reference to remote reference states. This suggests a high level of bus (or interconnection network) traffic for the maintenance of consistency. As will be seen in [Gee93a], without significant recoding of application and systems software, we do not believe that multiprocessors can function very effectively.

Since sharing behavior does seem to vary widely across different workloads, it appears that cache-consistency protocols which *adapt* to different sharing patterns [Karl86, Arch88, Egge89c] may be a useful alternative for machines running a wide variety of applications. In [Gee93a], we propose a protocol which updates data only once, and then invalidates the data on the next write if the previous update has not been used by other processors. This protocol is partially motivated by observed write run lengths in our programs, which vary in length from as little as 1 to 2 writes in the T-bit and MIT workloads, to as many as 6 writes in the Ardent workload with a 64 byte block size. If the first update is never used by other processors, then it is highly likely that many more updates will also be performed and never used.

In a companion paper [Gee93a], some of the ideas presented here are evaluated and validated through simulation of a large number of cache consistency protocols. Results from that paper confirm that invalidate-based protocols combined with large block sizes perform best on the Ardent vectorized workload, while update-based protocols and small block sizes perform better for the scalar T-bit and MIT workloads. Two adaptive protocols which switch from update to invalidate based on reference history yield satisfactory performance across all workloads, and would be a good alternative for general purpose machines.

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 Int’l Symp. Comp. Arch.*, May, 1988, Honolulu, HI, pp. 280-289.

[Arch84] J. Archibald and J.L. Baer, ‘‘An Economical Solution to the Cache Coherence Problem,’’ *Proc. 11th Int’l Symp. Comp. Arch.*, also *Sigarch News*, 12, 3, June, 1984, pp. 355-362.

[Arch86a] J. Archibald, ‘‘High Performance Cache Coherence Protocols for Shared-Bus Multiprocessors,’’ Technical Report 86-06-12, June, 1986, Computer Science Dept., University of Washington, Seattle, Washington

[Arch86b] 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. Conf. on Supercomputing*, July, 1988, St. Malo, France, pp. 337-345.

[Bayl89] S.J. Baylor and B.D. Rathi, ‘‘A Study of the Memory Reference Behavior of Engineering/Scientific Applications in Parallel Processors,’’ *Proc. 1989 Int’l Conf. on Parallel Processing*, August, 1989, St. Charles, IL, pp. I: 78-82.

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

[Chai90] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, ‘‘Directory-Based Cache Coherence in Large-Scale Multiprocessors,’’ *Computer*, vol. 23, no. 6, 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.

[Dare87] F. Darema-Rogers, G.F. Phister, and K. So, ‘‘Memory Access Patterns of Parallel Scientific Programs,’’ *Proc. 1987 ACM Sigmetrics*, May, 1987, Banff, Canada, pp. 46-58.

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

[Dubo86] M. Dubois, C. Scheurich, and F.A. Briggs, ‘‘Memory Access Buffering in Multiprocessors,’’ *Proc. 13th Int’l Symp. Computer Architecture*, June, 1986, Tokyo, Japan, pp. 434-442.

[Dubo88] M. Dubois, C. Scheurich, and F.A. Briggs, ‘‘Synchronization, Coherence, and Event Ordering in Multiprocessors,’’ *Computer*, February, 1988, pp. 9-21.

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

[Egge89a] S. Eggers, ‘‘Simulation Analysis of Data Sharing in Shared Memory Multiprocessors,’’ Ph.D. Thesis, University of California Berkeley, April, 1989, Tech. Rpt. No. UCB/CSD 89/501.

[Egge89b] 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.

[Egge89c] S. Eggers and R. Katz, ‘‘Evaluating the Performance of Four Snooping Cache Coherency Protocols,’’ *Proc. 16th Int’l Symp. Comp. Arch.*, June, 1989, Jerusalem, Israel, pp. 2-15.

[Egge90] S. Eggers and T. Jeremiassen, ‘‘Eliminating False Sharing,’’ University of Washington Technical Report 90-12-01, 1990.

[Gee92a] J. Gee and A.J. Smith, ‘‘The Performance Impact of Vector Processor Caches,’’ *Proc. of the 25th Hawaii Int’l Conf. on System Sciences*, Kauai, HI, Jan. 1992, pp. I:437-448.

[Gee92b] J. Gee and A.J. Smith, ‘‘Vector Processor Caches,’’ U.C. Berkeley Computer Science Division Technical Report UCB/CSD-92-707, 1992.

[Gee93a] J. Gee and A.J. Smith, ‘‘Absolute and Comparative Performance of Cache Consistency Algorithms,’’ paper in preparation, 1993.

[Gee93b] J. Gee, ‘‘Analysis of Cache Performance in Vector Processors and Multiprocessors’’, Ph.D. dissertation, Computer Science Division, UC 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.*

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

[Gupt92] A. Gupta, W. Weber, ''Cache Invalidation Patterns in Shared-Memory Multiprocessors,'' *IEEE Trans. Comp.*, vol. 41, no. 7, July, 1992, pp. 794-810.

[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, et. al., ''Implementing a Cache Consistency Protocol,'' *Proc. 12th Int'l Symp. Comp. Arch.*, June, 1985, Boston, MA, pp. 276-283.

[Kold91] E. Koldinger, S. Eggers, and H. Levy, ''On the Validity of Trace-Driven Simulation for Multiprocessors,'' *Proc. 18th Int'l Symp. Comp. Arch.*, May, 1991, Toronto, Canada, pp. 244-253.

[Kuma89] M. Kumar and K. So, ''Trace Driven Simulation for Studying MIMD Parallel Computers,'' *Proc. 1989 Int'l Conf. on Parallel Processing*, August, 1989, St. Charles, IL, pp. I: 68-72.

[Kure88] O. Kure, ''Optimization of File Migration in Distributed Systems,'' U.C. Berkeley Technical Report No. UCB/CSD 88/413, April, 1988.

[Lamp79] L. Lamport, ''How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,'' *IEEE Trans. Computers*, Sept. 1979, pp. 690-691.

[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.

[Lewi73] P. Lewis and G. Shedler, ''Empirically Derived Micromodels for Sequences of Page Exceptions,'' *IBM J. Res. Develop., March, 1973, pp. 86-100.*

[Okra89] B. O'Krafka, ''An Empirical Study of Three Hardware Cache Consistency Schemes for Large Shared Memory Multiprocessors,'' Electronics Research Laboratory Memorandum UCB/ERL M89/62, University of California, Berkeley, May, 1989.

[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.

[Sche87] C. Scheurich and M. Dubois, ''Correct Memory Operation of Cache-Based Multiprocessors,'' *Proc. 14th Int'l Symp. Computer Architecture*, June, 1987, Pittsburgh, PA, pp. 234-243.

[Site88] R. Sites and A. Agarwal, ''Multiprocessor Cache Analysis Using ATUM,'' *Proc. 15th Int'l Symp. Comp. Arch.*, May, 1988, Honolulu, Hawaii, pp. 186-195.

[Smit91] A.J. Smith, ''Second Bibliography on Cache Memories,'' *Computer Architecture News*, June, 1991, pp. 154-182.

[Spir77] J. Spirn, *Program Behavior: Models and Measurements*, Elsevier North-Holland, Inc., New York, NY, 1977.

[Stun91] C. Stunkel, B. Janssens, and W.K. Fuchs, ''Address Tracing for Parallel Machines,'' *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.

[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 Technical Report CMUCDS-93-4, March, 1993.

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

[Zuck92] R. Zucker and J. Baer, ''A Performance Study of Memory Consistency Models,'' Technical Report 92-01-02, January, 1992, Computer Science Dept., University of Washington, Seattle, Washington.