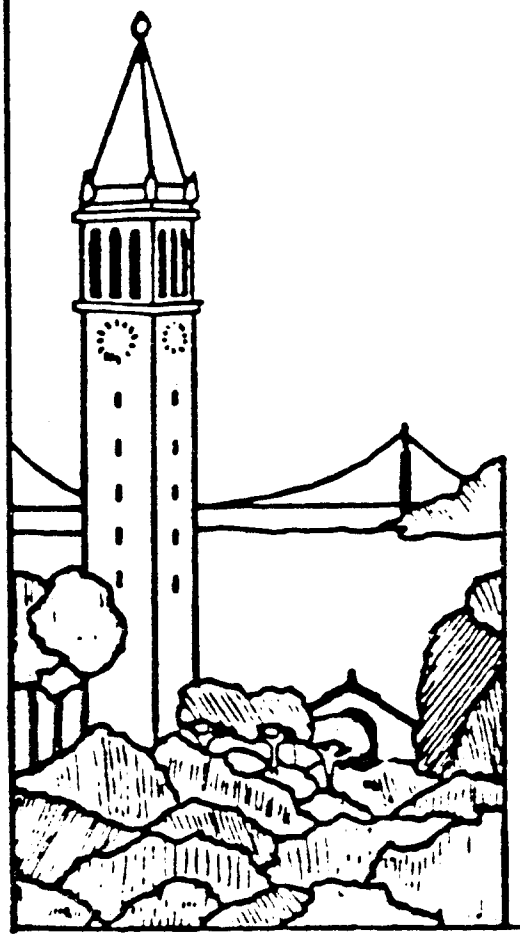
A Characterization Of Sharing In Parallel Programs And Its Applicability to Coherency Protocol Evaluation

Susan J. Eggers and Randy H. Katz



Report No. UCB/CSD 87/387

December 1987

Computer Science Division (EECS)
University of California
Berkeley, California 94720

A CHARACTERIZATION OF SHARING IN PARALLEL PROGRAMS AND ITS APPLICABILITY TO COHERENCY PROTOCOL EVALUATION[1]

Susan J. Eggers and Randy H. Katz

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

Abstract

In this paper we use trace-driven simulation to analyze the memory reference patterns of write shared data in several parallel applications. We first develop a characterization of write sharing (based on the notion of a write run), and then examine the traces, using metrics derived from the characterization. The results indicate that the amount of write sharing in all programs is small; and that it is characterized by short sequences of per processor references, with little contention for either data or locks.

We determine to what extent this analysis can be used to predict the coherency overhead of write-invalidate and write-broadcast protocols. We develop a simple model of write sharing from the write run characterization. By applying the results of the sharing analysis to the model, weighted by machine-specific cycle costs for carrying out coherency-related bus operations, we can approximate relative protocol performance. We compare these results to those from accurate architectural simulations. The model is a good predictor of protocol performance when the unit of the coherency operations matches that in the sharing analysis. This is the case for the write-broadcast protocols, in which one word is broadcast for each write to shared data. However, in Berkeley Ownership, a write-invalidate protocol, the unit of coherency is an entire cache block. When the block size is large, performance for this protocol is quite sensitive to the memory reference patterns within the block.

[1] [2] This work was supported by SPUR/DARPA contract No. N00039-85-C-0269, an IBM Predoctoral Fellowship, XCS/DARPA contract No. N00039-84-C-0089, Digital Equipment Corporation, and California MICRO (in conjunction with Texas Instruments, Xerox, Honeywell, and Philips/Signetics).

1. Introduction

Shared memory multiprocessors [ELXS84, Hill86, Pfis85, Rose85, Sequ84, Thac87] are emerging as an important class of computer systems. Unfortunately, shared memory is a double-edged sword. It provides the simplest parallel programming model -- a single-level of globally accessible memory -- but can be a critical performance bottleneck. Processor caches reduce the bandwidth demands on the shared memory [Good87], but they introduce the problem of keeping a consistent view of memory across them.

Cache coherency protocols describe operations for reading and writing memory that guarantee that this consistent view is maintained, i.e., a system with distributed caches behaves like one without them. Should multiple writes to shared memory locations occur simultaneously, it should be the case that (1) a value received on a memory read is the update of the last write to that location, and (2) the behavior of the coherency protocol is always predictable, i.e., no race conditions exist. A number of coherency protocols have been proposed, but their behavior under real-world workloads has not been determined. In particular, it is not clear which protocols perform best with specific memory system architectures and implementations.

The alternative protocols cover both software and hardware approaches. In the software protocols [Bran85, Edle85, McGr86, Olso85, Saty80, Smit85], the programmer must identify all potentially sharable objects at compile time, so that the operating system or compiler can take appropriate steps to preserve consistency. The overhead of maintaining consistency[1] is incurred whether or not sharing actually takes place during program execution. These protocols use mechanisms such as noncacheable pages, synchronization, and cache flushing to enforce coherency.

Hardware solutions free the programmer from the responsibility of specifying shared structures, and some incur the cost of preserving coherency only when the blocks are actively shared. These benefits occur at some cost in the hardware complexity of the cache and/or memory controllers. The control of the hardware protocol can either be centralized at the memory controller, using a global state directory [Arch84, Cens78, Gust82, Tang76, Widd80, Yen82], or be distributed among the caches and employ a snoop[2] to track shared addresses [Bell85, Bita86, Fiel84, Fran84, Good83, Katz85, McCr84, Papa84, Saty80, Sega84, Thac87].

Many of these alternatives have been evaluated analytically [Dubo82, Pate82] and with parameterized simulation [Arch86, Vern86]. Both techniques require assumptions about the sharing behavior of the workload. Some studies have assumed that shared data exhibits poor locality, and have modeled its access behavior using the independent reference model [Spir77]. Others have varied the amount of contention over shared variables, but have assumed uniform contention for both types of shared data: locks and the structures they protect. If any of these assumptions are incorrect, then the analysis may produce misleading results. At the very least, the assumptions should be verified by analyzing a workload of real parallel programs.

In this paper we analyze the sharing behavior of four parallel applications by trace-driven simulation. We are interested in both (1) the amount of write sharing in the applications, and (2) the pattern of multiprocessor accesses to the write shared data, i.e., whether there is contention for the shared data or whether it is accessed on a per processor basis over long periods of time. Both the quantity and pattern of sharing are important factors in relative coherency protocol performance. The emphasis here is on the pattern of sharing that is inherent in the application programs themselves, rather than that caused by the underlying memory system architecture, or the cache coherency protocol. To this end, the study will be conducted as independent of the architecture, implementation, and coherency protocol as possible.

The primary reason for examining sharing behavior is to evaluate the performance of coherency protocols. We would like to determine how well an analysis of sharing can predict protocol performance for realistic architectures. We apply the results from the sharing analysis to a model that reflects the costs of write sharing under different protocols. We then compare these approximations of protocol performance with the results of trace-driven multiprocessor simulations. The model is a good predictor of protocol performance when the unit of the coherency operations matches that in the sharing analysis. However, when they differ, protocol performance is quite sensitive to the memory reference patterns within the coherency block.

---

[1] Coherency overhead includes additional bus traffic, cache flushing and the delay to normal CPU processing when it is locked out of the cache.

[2] A snoop is bus-watching hardware that monitors the system bus for operations taking place on blocks contained in its cache. When an address match occurs, the snoop performs consistency-preserving operations in the cache directory, based on the type of bus request, the state of the cache block and the particular protocol.

We shall begin the paper by first introducing two distributed, hardware approaches to maintaining coherency, write-invalidate and write-broadcast, and an example protocol from each category. The protocols will be used at the end of Section 2 to illustrate how the values of sharing metrics vary with different protocols, and again in Section 5 in the realistic multiprocessor simulations. The bulk of Section 2 contains a characterization of sharing and appropriate metrics to reflect the characterization. Section 3 covers the methodology of the sharing analysis, the traces used in the study, and the trace-driven simulator. Section 4 presents the sharing results. Section 5 addresses the applicability of the sharing analysis to the two types of distributed, hardware coherency protocols. First, costs reflecting each category are applied to a sharing model that is based on the characterization developed in Section 2. Then the example protocols are simulated in a realistic multiprocessor architecture and implementation, using the same traces as in the sharing analysis. Section 6 summarizes our conclusions and directions for future work.

2. Characterization of Sharing

2.1. Coherency Protocol Examples

When a processor in a shared memory multiprocessor writes to shared data, there are two different procedures that it can follow. It can either invalidate all other cached copies of the data and then update its own without further bus operations. Or, it can broadcast the updates to all other caches, so that all processors always have the most current value of the data. The former method is known as *write-invalidate*, and the latter *write-broadcast*. We are interested in contrasting the relative performance of distributed, hardware implementations of these two coherency approaches in copy back caches. To do this, we shall introduce representative protocols in each category, and then use them in the remainder of the analysis.

Berkeley Ownership [Katz85] is a write-invalidate protocol that has been implemented in the SPUR multiprocessor [Hill86]. It is based on the concept of cache block ownership. A cache obtains exclusive ownership of a block via an invalidating bus transaction. Once ownership has been obtained, the cache can update a block locally without initiating additional bus transfers. Ownership also carries the obligations both to update main memory on block replacement and to provide data to other caches upon request. All cache-to-cache transfers are done in one bus transfer, with no memory update. Because it creates a data writer that can access a shared block without using the bus, we expect Berkeley Ownership to minimize the overhead of maintaining cache coherency in two cases: when there are multiple consecutive writes to a block by a single processor, and when there is little contention for the shared data.

The *Firefly Protocol* is a write-broadcast protocol that has been implemented in the DEC Firefly multiprocessor [Thac87]. Its processors broadcast writes to shared data, but use copy back for private data. The bus-watching snoops assert a special bus line to indicate sharing, whenever they detect an operation for a block that resides in their respective caches. The scheme has potential performance benefits for both private and actively shared blocks. By broadcasting all shared updates, it avoids the ping-ponging of data among caches that would occur with the invalidations of Berkeley Ownership. However, for data that is shared in a sequential fashion, with each processor completing all its accesses to the data before another processor begins, the write through policy for shared data may degrade bus performance.

2.2. A Characterization of Sharing and the Sharing Metrics

Our characterization of sharing serves three purposes. First, it provides an understanding of the memory reference patterns of write shared data. Second, it highlights the essential differences between the protocols, and explains how different patterns of sharing can affect protocol performance. Third, it is used as the basis for a model of sharing that approximates the coherency overhead of particular protocols.

We base our characterization on two aspects of memory accessing: (1) the number of sequences of unique processor references to a shared address, and (2) the length of these sequences. Both can be portrayed by the notion of a *write run*, which is the central concept of our characterization (Cf. Figure 1). A write run is a series of references to a *shared address* by a single processor, uninterrupted by any accesses by other processors. It is initiated by a processor's first write to the address (in the run), and terminated by the first access by another processor, either a read or write. (This latter access is called an *external* read or write, because it is external from the point of view of the processor that is the current owner of a write run.) Write runs are nonoverlapping units. Each shared address has a different sequence of write runs.

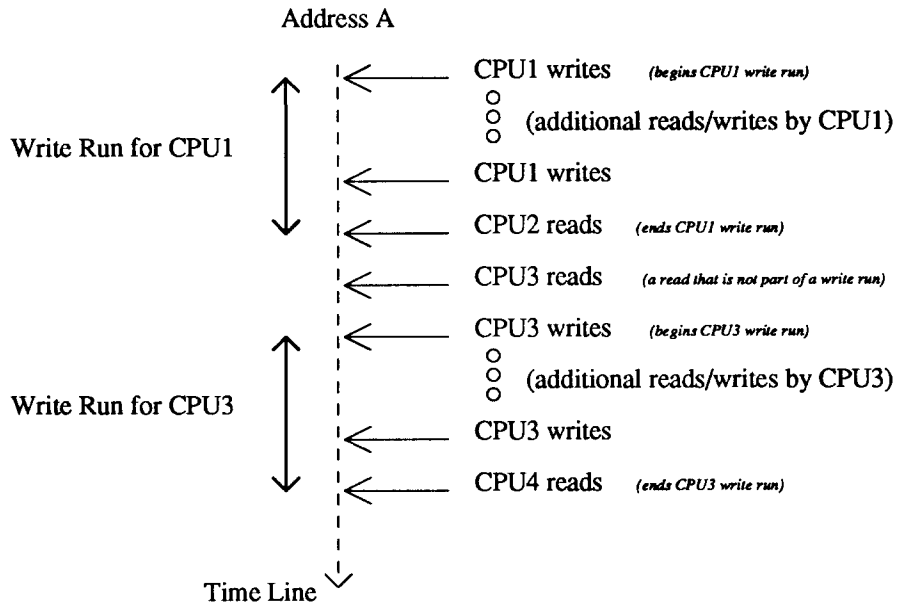


Figure 1: Example Write Run for a Shared Address

A write run is a sequence of references to a shared address by a single processor. It begins with the processor's first write to the address (e.g., the first "Cpu1 writes"), and ends with the first access by another processor (e.g., Cpur2 reads"). The vertical arrows denote the time over which the the write run occurs; the number of writes in this interval is the length of the write run.

A write run can contain solely write references, or both reads and writes. We focus on writes (with one exception), since shared writes cause coherency overhead, and most reads are handled identically in both protocols. In both the write-invalidate and write-broadcast protocols additional bus operations are required to maintain coherency on writes, and in each case the overhead is different. For example, in Berkeley Ownership the initiation of a new write run results in an invalidating bus operation; however, in the Firefly *each* write in a shared write run causes a bus operation.

The initial read to an address is always a miss; and, given an infinite cache assumption (explained in Section 3.1), reads within a write run are all cache hits. *Each type* of read takes the same number of cycles, regardless of the coherency approach adopted. Therefore the reads do not affect the pattern of shared accesses and consequently the variation in performance due to the particular coherency protocol. However, they are important to track in two cases. First, an external read can be the cause of the termination of a write run. The number of initial, per-processor external reads after a write run is an indication of the number of processors actively sharing an address. In write-invalidate protocols, it is a cause of coherency overhead. Second, including reads in the counts of references within a write run provides an accurate basis for measures of locality.

The write run metrics we use to analyze sharing appear in Table 1. The length of a write run is measured in numbers of writes. Beginning with the first write to a shared address by a different CPU, the number of consecutive writes is counted until the first access by another CPU. We define the amount of *sharing* for a given address to be the number of distinct write runs, i.e., the number of times the address changed writers. *Contention* for a shared

Metric	Aspect of Sharing It Measures
Count of writes in a write run	consecutive single processor usage for shared data
Count of write runs	sharing
Count of per processor first external rereads	contention
Ratio of write runs per total write shared addresses	contention
Number of busywaiters for a lock	contention

Table 1: Metrics of Sharing Based on Write Runs

address is quantified in several ways: (1) the number of processors that *re*read[3] the address after the end of a write run; (2) the number of write runs relative to the total number of write shared addresses; and (3) the number of processors that are waiting for a lock when it is unlocked.

2.3. Applying the Metrics

The write run metrics are useful for analyzing the performance tradeoff between the write-invalidate and write-broadcast protocols. A long write run suggests that a write-invalidate coherency protocol should be adopted. After the invalidation signal for the first update, all other writes by that processor can take place locally. On the other hand, write-broadcast would perform better with short write runs, particularly those of length one. In this case both approaches incur a coherency-related bus operation for the first write; but write-broadcast avoids the read misses of the write-invalidate schemes.

A large number of external rereads indicates that the addresses would have been needlessly invalidated had the coherency protocol been write-invalidate, and that a write-broadcast scheme would therefore have been preferable. On the other hand, a low number of rereads indicates that the invalidations would have done little harm.

The performance of the two approaches depends on the combined effects of these measures. Even if the write run is short, but there is no contention, e.g., no external rereads for the address, a write-invalidate scheme still might produce the better performance. The opposite situation calls for write-broadcast; if the number of external rereads is greater than the write run length, that approach would be preferred.

Write run length also indicates whether there is contention for a shared address, or whether it is shared sequentially by each processor over long periods of time. Short write runs, particularly those occurring in a short time interval, suggest that a processor's algorithmic use of the data was interrupted by other CPUs also referencing the address, i.e., a hot spot. Contention is also greater: the greater the number of write runs, the greater the number of external rereads, and the larger the number of waiters for a lock. As was explained in Section 2.1, write-broadcast protocols are well suited for periods of contention.

3. Methodology

The multiprocessor model under study is a shared memory multiprocessor with a common bus interconnect, with the bus supporting a modest number of processors, i.e., five to twelve. Given that model, our objective is to focus on the sharing inherent in the application programs, and abstract away architectural and implementation details, which could affect the pattern of sharing. For example, in write-invalidate protocols the unit of the invalidation is the cache block. If the block size is larger than a word, then invalidations due to processor writes will unnecessarily nullify the other words in the block. If those words are subsequently accessed by different processors, additional bus reads will be incurred to obtain the data. The bus operations are additional, because they are caused by the particular implementation of the memory system and the coherency protocol, rather than being inherent in the

[3] Note the emphasis on the term *re*read. The reads that are counted are those that reference data that have been referenced prior to the termination of the write run. Read misses are not counted, because their cost is not coherency-related.

program's logic.

An analysis of sharing that is independent of the underlying architecture and the coherency protocol has several advantages. First, it provides an understanding of the memory reference pattern of shared data that is inherent in the applications themselves. Second, sharing simulations are simpler to implement (for example, the details of bus arbitration and bus transactions, snoop activity, and cache controller/snoop interaction over the use of the cache can be omitted). Third, for a single trace, only one simulation is needed (as opposed to one per each combination of architecture and protocol parameter values). For these reasons, the sharing study will be conducted as independent of the architecture, implementation, and coherency protocol as possible.

3.1. Architecture/Implementation Independence

Independence from the architecture and its implementation is achieved in several ways. First, the simulations are done with *infinite caches* to eliminate the effect of cache size on block placement. In an infinite cache there is room for all references, and no blocks need to be replaced and consequently reaccessed. Reaccessing increases both bus traffic and miss ratios of shared data directly, and has an indirect effect by altering the order of processor access to the bus, thereby changing the pattern of shared accesses.

Second, addresses, rather than blocks, are tracked to eliminate the effect of changing the cache block size. This is equivalent to setting the cache block size to one word.

Third, all memory references take the same amount of time, regardless of whether they are reads or writes, hits or misses, or the misses are satisfied by main memory or another cache. The differences in the amount of time required to carry out these alternatives (in a real system) is sensitive to the memory organization, particularly memory latency, the bus transfer time, and the cache controller implementation.

Fourth, memory references are satisfied on a per processor, round robin basis, to give all processors equal processing time.

Lastly, the cycle time per instruction is a constant. Varying the instruction time to mirror the underlying implementation affects reference latency, which alters the global sequence of shared accesses by modifying the order in which processors obtain the bus. An argument could be made that instruction cycle times should be included in the simulation, because the particular choice of instructions reflects the semantics of the parallel algorithm. However, in the current programming paradigm (explained below) all parallel processes are executing the same code. Thus the variation in instruction times would be identical across processors.

We intend to compare the sharing results to simulations of the traces under a particular multiprocessor architecture and using Berkeley Ownership and the Firefly protocols (Section 5). To avoid cold start effects in these simulations, all caches obtain steady state before statistics are gathered. Steady state was determined for each trace separately;[4] simulation statistics were then gathered on the next 300K references (per processor). The sharing analysis obeys the same steady state requirements, so that the results of both studies will cover the same portion of the traces.

3.2. Coherency Protocol Independence

Our preliminary studies under realistic systems parameters [Egge88] indicate that the metrics associated with multiprocessor performance, and the sharing aspects in particular, are sensitive to the timing differences introduced by the choice of cache coherency protocol. The differences affect the amount and pattern of sharing and the runtime of the program in three ways: directly, by causing different bus events to occur, and indirectly by (1) altering the multiprocessor (systemwide) order of references to shared data and (2) varying the amount of busywaiting needed to obtain a lock. Therefore it is desirable to have the sharing simulation take place with an ideal coherency protocol, i.e., one with no bus-related overhead involved in carrying out the sharing operations. Under this coherency model, accesses to shared data are still tracked and coherency maintained, but with no cost in time.

Two aspects of processor synchronization (and their corresponding overhead) are still included: barriers and busywaiting for locks. Both of these constructs are reflections of the underlying algorithm. Barriers prevent

[4] Cumulative first reference misses were determined over the first six million references. Steady state was defined to be that point at which the additional first reference miss rate for the remainder of the trace snapshot was .2 percent or less, depending on the trace.

processes from executing beyond a certain point in the algorithm, until all parallel processes have reached that point. They are used to guarantee a correct ordering of phases of the program, e.g., to separate time steps in a circuit simulation. Busywaiting is more difficult to justify. One could argue that busywaiting should be eliminated, because it reflects the timing constraints of the underlying architecture and the policy of the cache coherency protocol, as well as the algorithm. However, under the assumption of architecture and coherency protocol independence, the busywaiting that occurs is a reflection of the contention for the shared locks that is inherent in the application's flow of control.

3.3. Trace-driven Simulation

3.3.1. The Traces

Our analysis is based on trace-driven simulations of six parallel programs. The programs are primarily CAD tools that were developed for shared memory multiprocessors (Cf. Table 2). Some are production code; others are research algorithms. Two of the traces are based on simulated annealing algorithms. PUPPY [Caso86] is a modified simulated annealing algorithm for IC design placement, placing twenty-three cells. TOPOPT [Deva87] does array optimization, and is based on dynamic windowing and partitioning techniques. Its input is a technology independent multi-level logic circuit. PLOVER [Ma87] is a logic verification program using PODEM-based [5] enumeration algorithms. The final trace, PSPICE [McGr86], is a circuit simulator; it is a parallel version of the original direct method, and its input is a chain of 64 inverters.

All applications were constructed using the same programming paradigm for carrying out the parallel activities. The level of granularity of the parallelism is a process. The model of execution is single-program-multiple-data, with each child process executing identical code on a different portion of shared data (Cf. Figure 2). The shared data is divided into units that are placed on a queue. Each child takes a unit of work from the queue, computes on it, and then writes results to shared memory, either creating another entry of work in the queue or overwriting values of a previous iteration. When the programs first begin execution, there is unusual contention for the locks protecting the queue of work, since all child processes try to take a unit of work simultaneously. Only one process will obtain access to the queue at a time. Assuming that each process does a comparable amount of processing, they will thereafter access the queue in the same order and spaced in time by the computation time. This self-scheduling is disrupted by synchronization barriers, which are used to separate phases in the computation. The disruption should cause more busywaiting and therefore an increase in references to shared addresses.

The scope of the traces is limited to memory references of the applications, and the operating system runtime routines used to set up shared memory and carry out synchronization. Because of the well known difficulty in

Parallel Applications			
Trace Name	Architecture, Operating System	Program Description	Number of Processors
PUPPY	Sequent, Unix	simulated annealing algorithm for cell placement	12
TOPOPT	Sequent, Unix	simulated annealing algorithm for array optimization	11
PLOVER	Sequent, Unix	logic verification	12
PSPICE	ELXSI 6400, Embos	direct method circuit simulator	5

Table 2: Traces used in the Simulations

The traces used in the sharing simulations were gathered from parallel programs that were written for shared memory multiprocessors. The programs are all "real", being either production quality or research applications.

[5] "path-oriented decision making" (depth-first search of graphs representing the circuits).

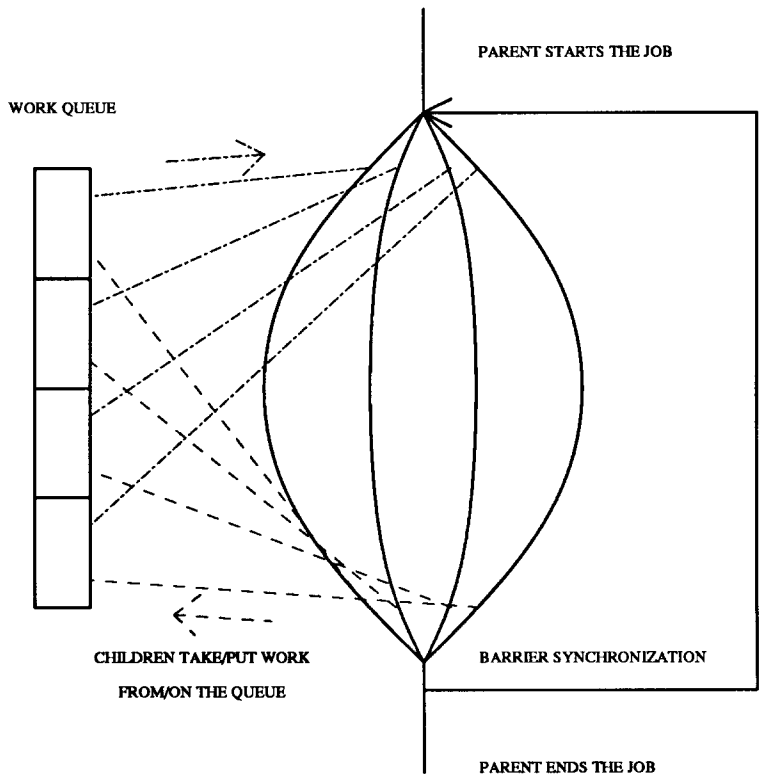


Figure 2: Flow Chart of the Programming Paradigm of the Parallel Traces

This is a simplified representation of the programming paradigm of the parallel traces. The parent process starts and ends the program, and forks child processes that do the parallel portion of the computation. Each child process executes the same code. At the end of the parallel computation, the children resynchronize, and then repeat the computation. Within each iteration, the children process different portions of the work queue, which reside in shared memory. For example, in a parallelized circuit simulator the circuit would be divided into groups of devices (nodes). In each iteration, each child would process a particular node. Sharing for the program data occurs because the inputs and outputs of the nodes interconnect, and a node may be processed by different child processes in different iterations.

tracing operating systems code, the path of the applications through the rest of the operating system is not captured by our traces. In addition, each of the parallel processes was run on a single processor without process migration. Including operating systems code and allowing process migration will increase the amount and therefore the cost of sharing [Agar87]. It is widely believed there is more sharing in operating systems activity than in user applications [MacD84]. In addition, process migration will cause sharing to occur for private data in the applications. Therefore the sharing results of these traces will be underestimated.

The traces are generated on a per processor basis. During trace postprocessing, shared addresses are detected and identified as locks or other shared data. In the ELXSI-generated program, coherency was maintained via software methods; therefore all memory references reflecting that implementation, such as cache flushing instructions, were stripped from the traces. The number of processors in the simulations depends on the number of processors that were used in trace generation. For PSpice this number is 5, and for the Sequent traces either 11 or 12.

3.3.2. The Multiprocessor Simulator

All simulations are run on a multiprocessor simulator, in which each component of a multiprocessor, (e.g., a CPU, a snooping cache controller, the bus) is written as a separate task. The multiprocessor simulator has a deterministic event-driven simulator base [Fuji83, Hell84] that handles all task scheduling, synchronization, and message passing among the various tasks of the multiprocessor. (For example, a message may be a cache controller request to the bus, to read a block of data.)

There are two sets of clocks in the simulator. The global clock indicates the current time in the multiprocessor system as a whole. In addition, each task has its own clock that is incremented to reflect the amount of time taken by a particular function, such as a cache lookup or bus arbitration. All tasks are scheduled by comparing a task's private clock to the global clock and then scheduling the task with the minimum clock value. In the sharing simulations, the clocks were incremented by a constant value for each memory reference (imitating round robin scheduling of instructions). In the protocol simulations the increments accurately mimicked the asynchronous behavior of a multiprocessor system.[6]

Each processor trace is a separate input stream to the simulator. Synchronization among the separate input streams depends on the use of locks and barriers in the the programs and is handled directly by the simulator.

4. Sharing Results

Type of reference statistics for the traces appear in Table 3. The nonsharing-related percentages are within the normal range of program behavior. The important figure for sharing analyses is the low percentage of shared accesses, particularly to write shared data. Unless operating systems activity and process migration add substantially to the number of write shared references, memory references due to coherency overhead will be a small component of the total. However, they may still comprise a substantial proportion of total bus operations, since most references to write shared data result in a bus transaction, and board level caches have fairly low miss ratios.

A further classification of shared references by type of data appears in Table 4. Note the preponderance of references to shared data at the applications level over the locks that protect it. This indicates that there is little contention for shared data. A higher percentage of reads over writes for lock data (e.g., in PSPICE, PUPPY) means that there was busywaiting for the lock. A lock write value exactly twice that of the reads (PLOVER) signifies a total absence of busywaiting. The locking algorithm is the test-and-test-and-set [Sega84] sequence used in the SPUR multiprocessor: the read is the initial access of the lock; the two writes are for setting and clearing. (TOPOPT does not use locks; it protects its shared data with barriers and the semantics of the algorithm, i.e., within a particular phase of the program there are multiple readers for a shared address, but only one writer.)

Histograms for the length of the write runs and the number of external rereads are depicted in Tables 5 and 6. For most of the traces the write runs are short, with an average write run length of 2.83 writes, and roughly two thirds of the write runs containing only one write (.66 for PSPICE; .73 for PUPPY; .61 for TOPOPT). (The lone exception is PLOVER, for which the average write run length is 8.58, and the write runs of length one constitute only one quarter of the total.) In isolation, write runs this short argue for a write-broadcast protocol. However, the situation changes given few external rereads. The average number of rereads for PSPICE and PLOVER is less than one (PSPICE = .64, PLOVER = .65), and over 99 percent of their write runs (PSPICE = 99.5%, PLOVER = 100%) were terminated by one or fewer rereads.[7] The numbers for PUPPY and TOPOPT are slightly higher, but not appreciably. The average number of rereads is 1.16 and 1.20, respectively, and the percentage of the write runs ending with one or fewer rereads is 83 and 80. The low number of external rereads indicates that the invalidations of a write-invalidate scheme would cause little additional coherency overhead in terms of reread bus traffic.

Contention in PSPICE and PLOVER was low because of three factors. (Cf. Table 7). The first is the low number of external rereads mentioned above. Second, there are few write runs per shared writable address. The ratio of write runs/total shared write addresses referenced averages 2.24 during trace runs of 300,000 memory references per processor. The amount of computation in the applications is so large, and pattern of sharing is so sequential, that a 300,000 reference snapshot is not sufficient to capture the sharing of the work queue that occurs over iterations of the algorithm. This is a comment on the extreme sequentiality of sharing in the traces, rather than the

[6] Other multiprocessor simulators use round robin scheduling even for realistic simulations, e.g., [Edle85].

[7] No external rereads occur when the end of a write run is the beginning of the next. In this case the terminating access is a write.

Basic Trace Statistics										
Trace	Refs (1000s)	Code	Data	Reads (Data)	Writes (Data)	Private (Data)	Shared (Data)	Read Shared (Data)	Write Shared (Data)	Shared Data Space (Kbytes)
		(proportion of total references)								
PLOVER	3,604	.650	.350	.287	.063	.213	.137	.127	.010	114.5
PSPICE	1,522	.629	.371	.256	.115	.283	.088	.069	.019	26,431
PUPPY	3,348	.544	.456	.356	.100	.314	.142	.128	.014	326.3
TOPOPT	3,293	.664	.336	.315	.021	.194	.142	.140	.003	22.5

Table 3: Basic Trace Statistics

Basic Trace Statistics: Details of the Shared Data							
Trace	Shared Refs (1000s)	Application Shared Data	Locks	Application Shared Data		Locks	
				Reads	Writes	Reads	Writes
		(proportion of shared references)					
PLOVER	495	.993	.006	.921	.072	.002	.004
PSPICE	134	.906	.094	.703	.203	.084	.010
PUPPY	476	.926	.074	.835	.091	.070	.004
TOPOPT	469	1.000	.000	.980	.020	.000	.000

Table 4: Basic Trace Statistics

The number of references is the total processed in the simulation. The proportions are the arithmetic means across all processors. They were calculated from the sharing simulations, assuming architecture and protocol independence. Shared data space is the number of bytes statically allocated to all shared data. In all traces except PSPICE, it is the amount of shared memory required to execute the program on the particular input used. PSPICE was written in Fortran; therefore the shared space was statically allocated to fit inputs of varying sizes.

Write Run Length Histogram				
	Traces			
Run Length Bins	PLOVER %	PSPICE %	PUPPY %	TOPOPT %
1	26.0	65.9	73.2	60.6
2	27.1	21.9	10.1	11.0
3	5.3	3.6	2.3	4.7
4	11.5	4.0	4.2	7.3
5	3.2	0.8	1.1	3.1
6	5.1	0.1	0.5	1.1
7	1.0	0.5	1.7	1.2
8	3.3	0.4	1.7	1.6
9	1.3	0.0	0.2	0.6
10	1.6	0.1	0.8	0.3
11	0.7	2.6	1.0	0.5
12	2.7	0.5	0.3	0.2
13	0.8	0.1	0.2	0.2
14	0.6	0.0	0.2	0.1
15	0.4	0.0	0.3	0.4
16	0.4	0.1	0.1	0.2
17	0.4	0.0	0.3	0.2
18	0.4	0.0	0.4	0.3
19	0.6	0.0	0.5	0.5
20	0.6	0.0	0.3	0.3
>20	7.9	0.2	1.5	6.9
Total Write Runs	4403	15525	18989	1864

Table 5: Length of the Write Runs

This histogram depicts the number of write runs that have a particular write run length. The traces were heavily biased toward write runs that contained only one write. With the exception of PLOVER, at least two thirds of the write runs for each trace had one write.

External Rereads Histogram				
	Traces			
External Reads Bins	PLOVER %	PSPICE %	PUPPY %	TOPOPT %
0	35.2	37.1	34.9	71.2
1	64.9	62.5	48.4	8.4
2	0.0	0.3	8.5	7.7
3	0.0	0.2	2.1	2.5
4	0.0	0.1	1.0	1.6
5	0.0	0.0	0.7	0.3
6	0.0	0.0	1.0	0.3
7	0.0	0.0	0.8	0.3
8	0.0	0.0	1.4	0.0
9	0.0	0.0	1.0	0.0
10	0.0	0.0	0.7	8.2
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
Total External Rereads	897	9430	15404	886

Table 6: Number of External Rereads Following a Write Run

This histogram depicts the number of write runs that were followed by a particular number of external rereads. The graph indicates that a single or no external rereads terminated most of the write runs in all traces. This means that there was little or no contention for the shared variables, and argues for a write-invalidate coherency protocol.

Ratio of Write Runs per Total Write Shared Addresses		
Trace	Value	Time Period (memory references)
PLOVER	1.256	3,603,936
PSPICE	2.507	1,521,834
PUPPY	4.361	3,348,394
TOPOPT	1.829	3,292,878

Table 7: Write Runs as a Percentage of Total Write Shared Addresses

Contention for the write shared addresses in three of the traces, PSPICE, PLOVER and TOPOPT, was low. This is indicated, in part, by the low number of write runs per shared write address over a long period of time. Time is measured by the total number of memory references processed. Total write shared addresses is a dynamic measure of those locations accessed during the period.

insufficient size of the trace sample. If a larger section of trace were analyzed, the same sequentiality would have been exhibited: both the ratio of write runs/total shared write addresses and the time period (measured in numbers of memory references) would increase.[8] Third, there are few processors busywaiting for locks when the lock is

[8] Longer samples will be simulated to verify the sequentiality of the sharing.

unlocked (Cf. Table 8). Between 83 and 99 percent of unlocks occurred with no other processor wanting the lock. Given these three factors we conclude that the short write runs depicted in Table 5 result from the processors' intention to write to the shared addresses only once, rather than the write sequences being interrupted by accesses from other processors.

In summary, the sharing results of the architecture and coherency protocol independent simulations indicate that, in general, write-invalidate protocols should perform better than the write-broadcast protocols for the traces examined. The performance advantages stem primarily from the lack of contention for shared data (the few number of external rereads), and, for one trace, from the length of the write run. Of all the traces, PUPPY was the best candidate for a write-broadcast protocol, because of the combination of a longer busywaiting sequence, shorter write runs, slightly more external rereads and a greater number of write runs per write shared address.

5. Applicability to Protocol Classifications

5.1. The Write Run Model

We can develop a simple model of write sharing based on write runs. The purpose of the model is to evaluate coherency overhead; therefore it is restricted to coherency-related bus events. The model captures references to shared data that either degrade performance by causing additional bus traffic or are handled differently in the different protocols. It portrays write sharing activity only, because this is the only area where the write-invalidate and write-broadcast protocols differ. We assume that both approaches have similar uniprocessor bus utilization, i.e., for private data and instructions. Second, both are copy back schemes. Therefore we do not explicitly model these activities.

In the model, each shared address is assigned a state, based on its past write activity and current reference. When a shared address is accessed, a state transition occurs, as illustrated by the state diagram in Figure 3. A transition is made to the "different write run" state by a write to a shared address by a CPU other than the current writer. The number of transitions is the count of write runs for that address. The "same write run" state is entered each time a writer continues to write to the address. The number of transitions here is the total of all write run lengths for that address, excluding the first write in each run. The transition to "end of write run" is made by the first external reread to the address by each CPU. The total is the sum of all such rereads.

Busywaiters Histogram			
	Traces		
Bin	PLOVER %	PSPICE %	PUPPY %
0	99.2	83.1	86.8
1	0.9	13.9	10.2
2	0.0	2.1	2.5
3	0.0	0.8	0.6
4	0.0	0.4	0.2
Total Busywaiters	1070	630	930

Table 8: Number of Busywaiters

This histogram depicts the number times a processor was blocked from a critical section because another processor was executing in it. The snapshot was taken when the lock was unlocked, and the count is of the number of processors busywaiting for it. The figures indicate that there was almost no contention for the locks. At the very least over 83 percent of all locks were unlocked with no other processor waiting. (TOPOPT is not depicted, because it does no locking.)

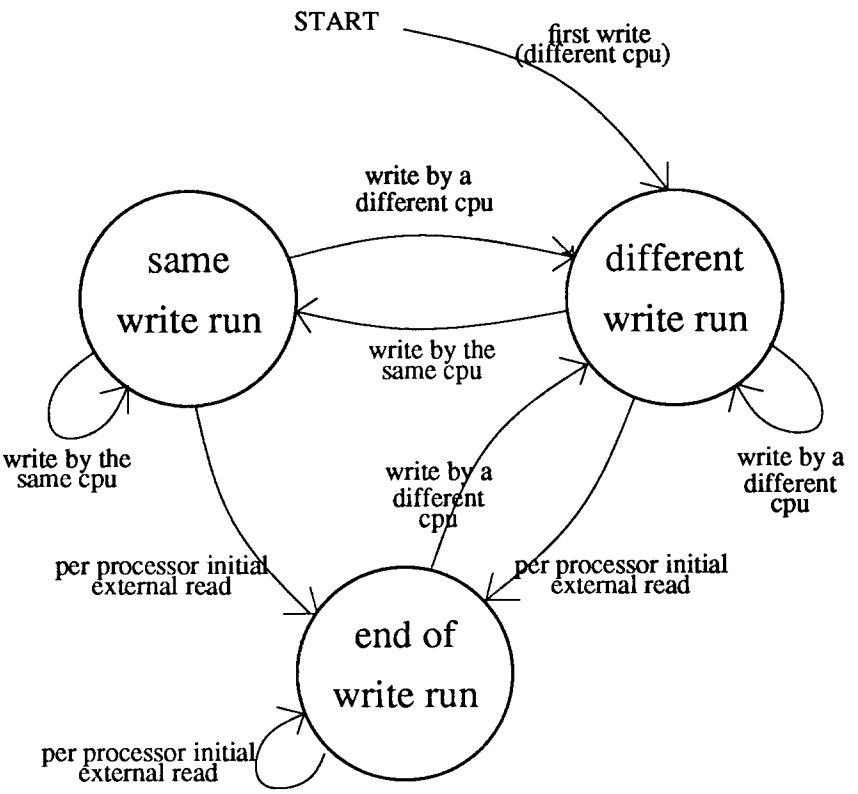


Figure 3: Model of Sharing Based on Write Runs

This finite state diagram reflects the model of sharing developed from a characterization based on write runs. A transition is made to the "different write run" state by a write to a shared address by a CPU other than the current writer. The "same write run" state is entered each time a writer continues to write. The transition to "end of write run" is made by the first external reread by each CPU. By assigning a coherency protocol-dependent cost to each arc in the state machine, an approximate cost of sharing for a particular cache coherency protocol can be determined.

We can use the model to quantify the intuitive conclusions of the last section. Relative coherency protocol performance can be determined by assigning costs (in cycles) to each arc in the finite state diagram and multiplying by the sum of the transitions for each arc across all shared addresses. The costs are a measure of the overhead of sharing traffic for a particular cache coherency protocol and are based on the timing constraints in the implementation of the SPUR multiprocessor [Hill86]. The arc costs for Berkeley Ownership and the Firefly are depicted in Table 9.

The cost to Berkeley Ownership is the invalidating signal for the first write of a write run and the rereads for data that was invalidated. The total cost is based on the sum of the write runs and the external rereads. The coherency overhead with the Firefly protocol is the sum of all broadcast writes. This can be approximated by the total length of all write runs.

The results of the comparison appear in Table 10. The absolute values of the cycle counts should not be taken literally, because of the architecture and protocol independent nature of the studies that produced the counts of state

Costs of Transitions in the Write Run Sharing Model				
Arc	Berkeley Ownership		Firefly	
	Bus Operation	Cost (cycles)	Bus Operation	Cost (cycles)
Write by a Different CPU	invalidation signal	11	word transfer	11
Write by the Same CPU	no cost	0	word transfer	11
First per Processor External Reads	block transfer	18	no cost	0

Table 9: Costs of Transitions for Berkeley Ownership and Firefly

This table classifies coherency overhead by type of bus operation for Berkeley Ownership and the Firefly. For each state in the state diagram (Figure 3), the bus operation required and its costs in cycles is depicted. All bus operations include cycles for address translation, bus arbitration, the bus operation and the appropriate snoop response. The block is assumed to be eight words. The small time difference between a full block transfer, and the invalidation signal and a one-word transfer is caused by (1) the overhead of snoop/cache controller interaction over updating the cache controller's copy of the state and (2) generating and detecting an explicit acknowledgment from each snoop for the latter two operations. The exact choice of cycle value is based on the implementation of the SPUR multiprocessor.

Cost of Berkeley Ownership & Firefly in the Write Run Sharing Model						
Trace	Coherency Protocol	Different Write Run (number)	Same Write Run (number)	End of Write Run (number)	Coherency Overhead (cycles)	Normalized to Berkeley Ownership
PLOVER	**Berkeley Ownership**	4403	33389	582	58909	1.00
	Firefly				415712	7.06
PSPICE	**Berkeley Ownership**	15525	13062	6006	278883	1.00
	Firefly				314457	1.13
PUPPY	Berkeley Ownership	18989	26417	17847	530125	1.00
	Firefly				499466	0.94
TOPOPT	**Berkeley Ownership**	1864	7700	1088	40088	1.00
	Firefly				105204	2.62

Table 10: Write Run Model Comparison of Berkeley Ownership & Firefly

This table depicts the number of occurrences of each arc in the write run model. The total number of cycles is obtained by multiplying the cost of each arc transition times the arc costs in Table 9. The bold entries indicate which of the protocols had better performance according to the write run analysis of sharing inherent in the programs.

transitions. What is important is the relative performance of the protocols for a particular trace. The figures of Table 10 support the conclusions of the last section, given SPUR cost assignments; namely, that for most of the traces, write-invalidate protocols (as represented by Berkeley Ownership) should get better performance than the write-broadcast protocols (as represented by the Firefly). A protocol's performance advantage hinges on the ratio of the length of the write runs (excepting the first write in each run) to per processor initial rereads. In the case in which this relation is closest to 1, the Firefly has the performance edge.

It should be pointed out that the coherency cycles are sensitive to both the overhead in the bus operations and the transfer size on a block read. In the SPUR implementation, both are on the high side. For example, the cache controller was implemented assuming that the priority for using the cache belonged to the processor rather than the snoop. Therefore all arc costs include cycles for the snoop's negotiating to obtain use of the cache, and acknowledging that it has finished. In addition, the block transfer cost is based on an eight word block size. If the arc costs had reflected a more optimized implementation, e.g., that used in the Firefly multiprocessor, the relative performance of the two protocols would have been different for one of the traces (PSPICE).[9]

5.2. Architecture Dependent Simulations of Snooping Protocols

We would like to determine the accuracy of the simple model of sharing in evaluating the performance of coherency protocols in a real system. We therefore compared the predictions of the write run model with simulation results using realistic architecture and protocol parameters. The memory system architecture was identical to SPUR (a 128K byte direct mapped, unified, board-level cache, with a 32 byte line size; one cycle cache reads, two cycle cache writes; a test-and-test-and-set sequence for securing locks; and the timing constraints of the SPUR cache controller [Wood88]). Bus arbitration was implemented using the NuBus [Texa83] arbitration protocol, and bus contention was accurately modeled. The number of cycles for instruction execution reflected those of the machines on which the traces were generated. The number of cycles for coherency-related bus operations were the arc costs in the previous section. Last, the coherency protocols were Berkeley Ownership and the Firefly.

The results of the simulations appear in Table 11. The data is the number of bus operations used to maintain coherency, and the cycles required to carry them out. The coherency cost (in bus operations) to Berkeley Ownership are the the invalidation signals and the reaccesses of invalidated data (corresponding to the sum of Different Write Run and End of Write Run in Table 10); for the Firefly, it is the total number of write-broadcasts to shared data (the sum of Different Write Run and Same Write Run).

The difference between the number of bus operations in the simulations and the state transitions in the sharing analysis demonstrates the dramatic effects of particular architectural parameters on actual performance. The discrepancy occurs whenever the unit of coherency operations in the sharing analysis, which is one word, does not match that in the real machine. In Berkeley Ownership, the unit of invalidation and reread is an entire cache block; and the block size in SPUR is 32 bytes. Therefore the effects of SPUR's large block size overshadow the coherency overhead that is due to the intrinsic sharing pattern in the applications.

These effects produce either a savings or an additional coherency cost, depending on the memory access pattern to words within the blocks. In PSPICE and PUPPY both the number of invalidations and the rereads were decreased, because of the spatial locality of the shared data. For example, after a writing processor invalidates, it possesses the only cached copy of the block. It pays the coherency overhead for the first write to the block, but can update the remaining words without using the bus. In contrast, the model of Section 5.1 records a separate write run for each word within the block. Therefore the invalidating signal is counted for the initial write to *each* word in the block, rather than just once. An analogous situation exists for the rereads.

The large block size improved Berkeley Ownership's performance in PSPICE and PUPPY, because of the sequential pattern of sharing and the contiguous allocation of shared memory. If there had been more contention for the shared data within the block, or if there had been less spatial locality, the large block size would have hurt performance. More contention means more invalidations interrupting all processors' use of the data in the block and a corresponding increase in the number of rereads to get it back. Less spatial locality results in separate invalidations for each word of shared data, an increase in the number of rereads, or both.

---

[9] The comparable figures for the Firefly multiprocessor are 4 cycles for a word transfer, and, presumably, 4 for an invalidation and 11 for a block transfer, assuming the SPUR block size.

Cost of Berkeley Ownership & Firefly in Realistic Simulations						
Trace	Coherency Protocol	Invalidation Signal (number)	Misses due to Invalidations (number)	Write Broadcasts (number)	Coherency Overhead (cycles)	Comparison to Sharing Analysis (percent)
PLOVER	Berkeley Ownership	14297	18368		487891	-728
	Firefly			37734	415074	**.15**
PSPICE	**Berkeley Ownership**	3602	3048		94486	66
	Firefly			23208	255288	19
PUPPY	**Berkeley Ownership**	7157	10442		266683	-50
	Firefly			45522	500742	**-.25**
TOPOPT	Berkeley Ownership	1252	12419		237314	-492
	Firefly			9564	105204	**0**

Table 11: Comparison of Berkeley Ownership & Firefly in a Realistic Simulation

The table contains the number of bus operations needed to maintain cache coherency, assuming a SPUR-like multiprocessor, and using either Berkeley Ownership or the Firefly protocols. For the Firefly, the total number of cycles required to carry out the operations matches those approximated by the write run model. However, for Berkeley Ownership, in which coherency operations take place on an entire cache block, rather than a word, the effects of the cache block size outweigh those of the sharing pattern in the application.

The latter factors were prevalent in the remaining two traces. In PLOVER, alternating writes by different processors to the words within a block caused separate invalidations for each write. The increase in invalidations was responsible for the subsequent rise in read misses due to invalidations. Recall that the average write run length of PLOVER was 8.58, the highest of all the traces by a wide margin. In the sharing analysis, only the first of the writes in these runs caused an invalidation. However, in the realistic simulations most of the writes caused invalidations, because of the interleaved (by processor) accesses within the large blocks.

In TOPOPT, only the number of read misses due to invalidations increased. An invalidation to one word in a block causes all other words to be nullified; additional read misses are needed to get the words back. In the model of Section 5.1 there are separate write runs for each word in the block, and the rereads are internal to the write runs. Accesses that are read misses in the realistic simulations are therefore considered hits in the sharing analysis, and consequently are not counted as coherency overhead. Sizing the trace's shared data structures to the cache block size would have eliminated the problem in the realistic simulations.

For all traces the number of rereads in the Berkeley ownership simulations was a large proportion of total misses: TOPOPT = 93%; PLOVER = 47%; PUPPY = 41%; PSPICE = 25%. The high figures are due, in part, to the low miss ratio for all references in the 128K byte cache (TOPOPT = .4%; PLOVER = 1.5%; PUPPY = .8%; PSPICE = .8%). Between 82 and 100 percent of these invalidation-caused misses were for applications shared data. TOPOPT, of course, has no locks; for the other traces the high proportion is exaggerated by the low amount of busywaiting.

The Firefly results more closely correspond to those of the prediction, because the unit of the coherency operations is identical in both the sharing analysis and the simulations. The 128K byte cache size in the simulations also contributes to the match. The caches are large enough that shared blocks are replaced infrequently, approximating the infinite cache assumed by the model. (Shared blocks were replaced on .02% of all memory references in PSPICE; figures for the other traces are: PUPPY = .009%, PLOVER = .003%, none in TOPOPT). Therefore once data was shared, it remained shared, and the write broadcasts continued.

6. Conclusion

This paper has reported on a new methodology for analyzing the sharing of a significant collection of parallel programs. To our knowledge, this is one of the first efforts to analyze traces of such programs. The results indicate that:

(1) The amount of application-level write sharing is small, on the order of 2 percent of total references or less. It is characterized by short sequences of per processor references (an average of 2.83 writes per write run for three of the traces), and little contention for either program data or locks. The lack of contention is exhibited by few rereads by processors other than the current data writer (.87 on average), a low ratio of total write runs to total write shared addresses (the average is 2.24), and a marked absence of busywaiting for locks (between 83 and 99 percent of unlocks occurred with no other processor wanting the lock.)

(2) The architecture and protocol independent sharing model of Section 5.1 is a good predictor of protocol performance when the unit of the coherency operations matches that in the sharing analysis. This is the case for the write-broadcast protocols, such as the Firefly, in which one word is broadcast. However, a model based solely on the sharing intrinsic to the programs is not sufficient, when the unit of the coherency operations is larger, such as a multi-word cache block. In this case, accesses to individual words within the cache block alter the order of coherency-related bus operations. The result is either a dramatic savings or additional cost in coherency overhead, depending on the exact memory reference pattern of the shared accesses within the block. Write-invalidate protocols, such as Berkeley Ownership, exhibit this type of behavior.

(3) The correct choice of protocol for the SPUR architecture is still open to question. The traces split evenly as to whether Berkeley Ownership or the Firefly gave the better performance.

Our future work will continue in three directions. First, one of the simplifying assumptions of the write run model will be dropped to more accurately model the pattern of sharing for real machines. In particular, the sharing analysis will be done using a protocol-specific unit for coherency operations, rather than a shared address. For write-invalidate protocols, such as Berkeley Ownership, this will mean tracking the sharing activity of a cache block. Second, more detailed simulations will be run to determine the sensitivity of coherency protocol performance to various architectural parameters, particularly block size. Third, the studies will be extended to traces generated by a parallel version of ATUM [Agar86] to ascertain the effects of operating system references and process migration.

We wish to acknowledge the efforts of several others who contributed to the work in this paper. Alan Smith made many useful suggestions during discussions of the sharing model. Mark Hill, David Wood, Corinna Lee and Dave Patterson gave valuable comments on earlier drafts. Dominico Ferrari and Yale Patt provided resources for running the simulations. Dianne DeSousa and her coworkers at ELXSI helped with the generation of the ELXSI trace. And Frank Lacy single-handedly shouldered the Herculean task of generating and postprocessing the Sequent traces.

References

[Agar86] A. Agarwal, R. L. Sites and M. Horowitz, "ATUM: A New Techniqwue for Capturing Address Traces Using Microcode", *Proceedings Thirteenth International Symposium on Computer Architecture*, 14, 2 (June 1986), 119-127.

[Agar87] A. Agarwal and A. Gupta, "Memory-Reference Characteristics of Multiprocessor Applications under MACH", submitted to Sigmetrics (1987).

[Arch84] J. Archibald and J. Baer, "An Economical Solution to the Cache Coherency Problem", *Proceedings 11th Annual International Symposium on Computer Architecture*, 12, 3 (June 1984), 355-362.

[Arch86] J. Archibald and J. Baer, "An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors", *ACM Transactions on Computer Systems*, 4, 4 (November 1986), 273-298.

[Bell85] C. G. Bell, "Multis: A New Class of Multiprocessor Computers", *Science*, 228 (April 1985), 462-467.

[Bita86] P. Bitar and A. M. Despain, "Multiprocessor Caches: Cache Synchronization and Busy-Wait Locking, Waiting, and Unlocking", *Proceedings 13th Annual International Symposium on Computer Architecture*, 14, 2 (June 1986), 424-442 .

[Bran85] W. C. Brantley, K. P. McAuliffe and J. Weiss, "RP3 Processor_Memory Element", *Proceedings 1985 International Conference on Parallel Processing* (1985), 782-789.

[Caso86] A. Casotto, F. Romeo and A. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells", *IEEE International Conference on Computer-Aided Design*, Santa Clara, CA (November 1986), 30-33.

[Cens78] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems", *IEEE Transactions on Computers*, C-27, 12 (December 1978), 1112-1118.

[Deva87] S. Devadas and A. R. Newton, "Topological Optimization of Multiple Level Array Logic", *IEEE Transactions on Computer-Aided Design* (November 1987).

[Dubo82] M. Dubois and F. A. Briggs, "Effects of Cache Coherency in Multiprocessors", *IEEE Transactions on Computers*, C-31, 11 (November 1982), 1083-1099.

[ELXS84] ELXSI, System 6400 Introduction (April 1984).

[Edle85] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller and J. Wilson, "Issues Related to MIMD Shared-memory Computers: the NYU Ultracomputer Approach", *Proceedings 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985).

[Egge88] S. J. Eggers, "Simulation Analysis of Data Sharing Support in Shared Memory Multiprocessors", Ph.D. thesis, in progress, University of California, Berkeley (completion, 1988).

[Fiel84] G. Fielland and D. Rogers, "32--bit Computer System Shares Load Equally Among up to 12 Processors", *Electronic Design* (September 1984), 153-168.

[Fran84] S. J. Frank, "Synapse Tightly Coupled Multiprocessors", Unpublished manuscript (1984).

[Fuji83] R. M. Fujimoto, "SIMON: a Simulator of Multicomputer Networks", Technical Report No. UCB/Computer Science Dpt. 83/140, University of California, Berkeley (September 1983).

[Good83] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", *Proceedings 10th Annual International Symposium on Computer Architecture*, 11, 3 (June 1983), 124-131.

[Good87] J. R. Goodman, "Cache Memory Optimization to Reduce Processor/Memory Traffic", *Journal of VLSI and Computer Systems*, 2, 1 & 2 (1987), 61-86.

[Gust82] R. N. Gustafson and F. J. Sparacio, "IBM 3081 Processor Unit: Design Considerations and Design Process", *IBM Journal of Research and Development*, 26, 1 (January 1982), 12-21.

[Hell84] D. E. Heller, "Multiprocessor Simulation Program SIMON", Shell Development Company (November 1984).

[Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N.

Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "SPUR: A VLSI Multiprocessor Workstation", *IEEE Computer*, 19, 11 (November 1986), 8-22.

[Katz85] R. Katz, S. Eggers, D. Wood, C. L. Perkins and R. Sheldon, "Implementing a Cache Consistency Protocol", *Proceedings 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985), 276-283.

[Ma87] H. T. Ma, S. Devadas, R. Wei and A. Sangiovanni-Vincentelli, "Logic Verification Algorithms and their Parallel Implementation", *Proceedings of the 24th Design Automation Conference* (July 1987), 283-290.

[MacD84] M. H. MacDougall, "Instruction-Level Program and Processor Modeling", *Computer*, 17, 7 (July 1984), 14-26.

[McCr84] E. McCreight, "The DRAGON Computer System: An Early Overview", *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, Urbino, Italy (July 1984).

[McGr86] S. McGrogan, R. Olson and N. Toda, "Parallelizing Large Existing Programs - Methodology and Experiences", *Proceedings of Spring COMPCON* (March 1986), 458-466.

[Olso85] R. Olson, "Parallel Processing in a Message-Based Operating System", *IEEE Software* (July 1985), 39-49.

[Papa84] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", *Proceedings 11th Annual International Symposium on Computer Architecture*, 12, 3 (January 1984), 348-354.

[Pate82] J. H. Patel, "Analysis of Multiprocessors with Private Cache Memories", *IEEE Transactions on Computers*, C-31, 4 (April 1982), 296-304.

[Pfis85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proceedings 1985 International Conference on Parallel Processing* (1985).

[Rose85] C. D. Rose, "Encore Eyes Multiprocessor Market", *Electronics* (July 8, 1985), 118-119.

[Saty80] M. Satyanarayanan, "Commercial Multiprocessing Systems", *IEEE Computer*, 13, 5 (May 1980).

[Sega84] Z. Segall and L. Rudolph, "Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor", *Proceedings of the 11th International Symposium on Computer Architecture*, 12, 3 (June 1984), 340-347.

[Sequ84] Sequent Computer Systems, Inc., Balance 8000 Technical Summary (November 1984).

[Smit85] A. J. Smith, "CPU Cache Consistency with Software Support Using 'One Time Identifiers'", *Proceedings of the Pacific Computer Communications Symposium*, Seoul, Republic of Korea (October, 1985).

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

[Tang76] C. K. Tang, "Cache System Design in the Tightly Coupled Mutliprocessor System", *Proceedings of National Computer Conference* (1976), 749-753.

[Texa83] Texas Instruments, "NuBUS Specification", TI-2242825-0001 (1983).

[Thac87] C. P. Thacker and L. C. Stewart, "Firefly: A Multiprocessor Workstation", *Preceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA (October 1987), 164-172 .

[Vern86] M. K. Vernon and M. A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets", *Preceedings of Performance '86 and ACM Sigmetrics 1986*, Raleigh, NC (May 1986), 9-17 .

[Widd80] L. C. Widdoes, Jr., "The S-1 Project: Developing High-Performance Digital Computers", *Proceedings of Compcon 80*, San Francisco, CA (February 1980), 282-291.

[Wood88] D. A. Wood, S. J. Eggers, G. A. Gibson, D. Jeong, R. H. Katz and D. A. Patterson, "The SPUR Cache Controller Chip", *Proceedings of SouthCon '88*, Orlando FL (March 1988).

[Yen82] W. C. Yen and K. S. Fu, "Analysis of Multiprocessor Cache Organizations with Alternative Main Memory Update Policies", *Proceedings 8th Annual International Symposium on Computer Architecture*, 9, 3 (May, 1982), 89-100.