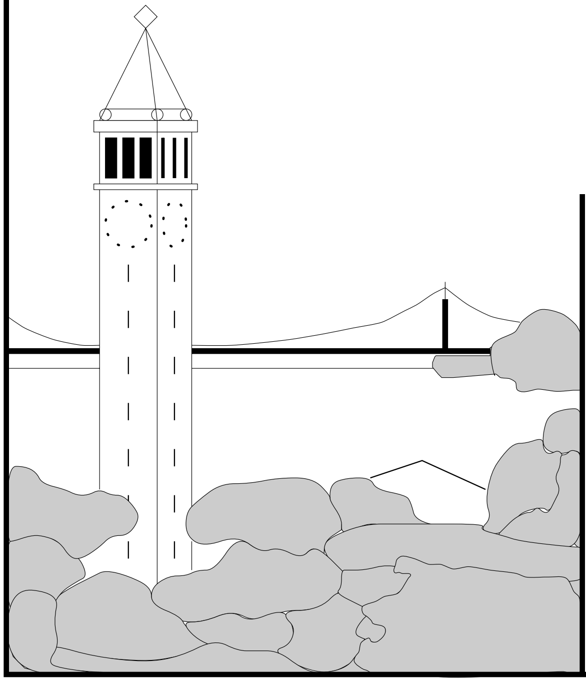
Minerva: An Adaptive Subblock Coherence Protocol for Improved SMP Performance

Jeffrey B. Rothman and Alan Jay Smith



Report No. UCB/CSD-99-1087

December 1999

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

Minerva: An Adaptive Subblock Coherence Protocol for Improved SMP Performance[†]

Jeffrey B. Rothman and Alan Jay Smith
Computer Science Division
University of California
Berkeley, CA 94720

December 10, 1999

Abstract

The major limitation on the performance of shared memory multiprocessors running parallel programs is the memory traffic due to sharing, i.e., the coherence or consistency induced memory traffic. Much of this traffic occurs due to false sharing (when two or more processors use disjoint portions of the same cache block) and dead sharing (the transfer of unreferenced words in a block when the block moves between caches). Dead and false sharing can be minimized or eliminated by the use of a small block size, but at the cost of substantially increased miss ratios due to true sharing and regular cache misses. Because shared memory multiprocessors are likely to be used most often for the multiprogramming of single thread programs and not for parallel programs, optimizing solely for the latter case is a poor idea.

In this paper, we present a new cache protocol, **Minerva**, which allows the effective cache block size to vary dynamically. **Minerva** works using sector caches (also known as block/subblock caches). Cache consistency attributes (from the MESI set of states) are associated with each 4-byte word in the cache, and consistency is maintained on a word basis. Each block (sector) in each cache can itself have one of the attributes of: invalid, exclusive or shared. Each block also has a current subblock (subsector) size, of 2^k words and a confidence value for that size. The subblock size is reevaluated every time there is an external access (read or invalidate) to the block, and is changed when the confidence level reaches zero. When a fetch miss occurs within a block, a subblock equal to the current subblock size is fetched. Note that the fetch may involve a gather operation, with various words coming from different sources; some of the words may already be present.

Despite the apparent complexity of the protocol, it can be implemented with fairly simple combinational logic. The nature of **Minerva** makes it easy and convenient to also implement optimizations such as bus read-sharing, read-broadcast (snarfing), and write-validate. For non-parallel workloads, **Minerva** converges to the Illinois protocol on 64-byte blocks, so performance for conventional workloads is also high.

Depending on the assumed cache sizes, block sizes, and bus timings, we find that **Minerva** reduces execution times by 19-40%, averaged over 12 test parallel programs. Our evaluation considers the utility of various other optimizations, compares the use of **Minerva** with restructuring the code, and considers the extra state bits required.

1 Introduction

Symmetric multiprocessor (SMP) systems with a small number of processors are becoming increasingly popular. The aggregate performance of such systems is typically limited by the bandwidth of the interconnecting bus. The bus traffic is increased in the case of parallel workloads by two problems, false sharing and dead sharing. False sharing occurs when consistency (coherency) operations are required, even though they would not be needed if the block size were one word. Dead sharing is a result of block granularity cache coherence enforcement, in that most of the data transferred between caches is not referenced before being invalidated by another cache. Dead sharing is, of course, a simple extension of what occurs in the uniprocessor case, when there are unreferenced words within blocks.

Our examination of multiprocessor workloads in

[†] Funding for this research has been provided by the State of California under the MICRO program, and by Cirrus Corporation, Cisco Corporation, Fujitsu Microelectronics, IBM, Intel Corporation, Microsoft Corporation, Quantum Corporation, Sun Microsystems, and Toshiba Corporation.

[RS99a] showed that a large fraction of the false sharing occurs due to references to a few badly behaved blocks. We found that (in the 64-byte block case): (1) only a single word was referenced in a block between the time the block was fetched and the time it was invalidated for 40 percent of the blocks; and (2) around 60 percent of the invalidated blocks that are dirty have only a single dirty (modified) word. Of the blocks that had more than 2 words written, the next largest number of words written was 16 (the maximum number per block).

To address this problem, we present our new coherence protocol **Minerva**, which employs dynamically sized subblocks to reduce the effects of false and dead sharing. **Minerva** is evaluated by comparing it to schemes with fixed block or subblock invalidation and fetch sizes using traces of 12 parallel programs. In addition, we also examine read-broadcasting (snarfing) and bus read-sharing (sharing the results of pending read transactions between multiple caches).

Minerva works using sector caches (also known as block/subblock caches). Cache consistency attributes (from the MESI set of states) are associated with each 4-byte word in the cache, and consistency is maintained on a word basis. Each block (sector) in each cache can itself have one of the attributes of: invalid, exclusive or shared. Each block also has a current subblock (subsector) size of 2^k words and a confidence value for that size. The subblock size is reevaluated every time there is an external access (read or invalidate) to the block, and is changed when the confidence level reaches zero. When a fetch miss occurs within a block, a subblock equal to the current subblock size is fetched. Note that the fetch may involve a gather operation, with various words coming from different sources; some of the words may already be present.

Despite the apparent complexity of the protocol, it can be implemented with fairly simple combinational logic. The nature of **Minerva** makes it easy and convenient to also implement optimizations such as bus read-sharing, read-broadcast (snarfing), and write-validate. For non-parallel workloads, **Minerva** just converges to the Illinois protocol on 64-byte blocks, so performance for conventional workloads is also high.

The remainder of this paper is organized as follows: Section 2 provides some background on cache memories and the issues involved in cache coherency for multiprocessor systems, and describes previously published subblock coherence schemes. Section 3 describes our workloads and our simulation methodology. We review some simple protocol independent performance enhancements in Section 4. Section 5 introduces our new protocol **Minerva**. The results and comparisons of the simulations are provided in Section 6. We present our conclusions in Section 7.

2 Background

2.1 Previous Coherence Protocols

Much research on cache coherence protocols for SMP systems has focussed on determining the best protocol in systems with fixed block sizes [EK88, Arc88, Lil93, NS94, GS96]. It has been found that invalidation-based protocols, such as Illinois [PP84], typically outperform update-based protocols, but protocols that can dynamically switch between update and invalidate can do slightly better [GS96]. None of the fixed-block size protocols, however, have properly addressed the false-sharing issue.

Some research attacking the false sharing problem has focussed on associating several coherence/transfer units (subblocks) with each address tag. This type of design, also known as a *sector cache*, has been studied extensively for uniprocessor systems [Lip68, HS84, Prz90, Sez94, RS99c, RS99d]. Several designs have been proposed for multiprocessor systems using a fixed size subblock [Goo87, AB94, CD93], showing good performance improvement over systems using block-size coherence granularity. Variable size block coherence was investigated in [DL92], showing better performance than fixed size blocks for most workloads, but no implementation details were provided. One dynamically adjustable subblock protocol allowed a block to be divided into two (possibly unequal) pieces for coherence purposes, yielding a slight improvement in performance [KB95].

2.2 Sharing Issues

False sharing occurs when different processors reference different words within a block; i.e., the block is shared, but the words are not. This was identified as a problem in [EK89a]. It has been investigated in a number of papers such as [BS93]. [TLH94, DSS95] found that false sharing is generally not the major source of cache misses for the workloads they studied. Attempts to restructure data to avoid false sharing were found to have some success in [EJ91, TLH90, JE95, RS99a].

Our analysis of sharing patterns in [RS99a] using an invalidation-based protocol with an infinite cache showed that false sharing is the largest source of misses for some of the parallel programs studied; in those cases sharing became a larger source of misses than true sharing and cold start misses once the block size became sufficiently large (16 bytes and greater). We found also that approximately 80 percent of bus traffic consists of data that is unused before being invalidated, but is required to be transferred because of block granularity coherence. This effect, which we call *dead sharing*, is mostly closely associated with false sharing, although it can be caused by true sharing (e.g., one shared word and 15 inactive words in a block).

3 Methodology

Our work is based on TDS (trace-driven simulation). Initially our research used execution-driven simulation (EDS), which is slightly more accurate [GH93]. However we found that results could be generated much more quickly using modern PCs and workstations (using compressed traces generated by our EDS tool **Cerberus** [RS99b]) than on the obsolete DEC5000 workstations on which our EDS system depends. To keep our simulations as accurate as possible, synchronization objects (barriers and locks) are simulated at run-time. Examination of some of the key data points simulated by both EDS and TDS showed extremely similar results.

3.1 Workload Characteristics

We examined a variety of parallel programs (12) to provide the results for this paper (Table 1). Ten of the programs come from the SPLASH 1 and 2 suites from Stanford University, which have been available to the research community as a *de facto* benchmark for comparing parallel program execution. These programs have all been used in a number of papers analyzing parallel code performance, and are described and characterized in more detail in [SWG92, WOT$^+$95, RS99a]. The other two programs (**topopt** and **pverify**) were created by the CAD group at U.C. Berkeley, and used for measurements at Berkeley and the University of Washington [EK89a], [EK89b], [EJ91], [AB95].

It is important to note that we are examining a specific type of workload with our simulations: those which are able to use the processors in a system to cooperate in solving a single problem. In particular, these programs use shared data structures, and individual processors are often doing the same operation to nearby data, or different operations to the same data. Some of our results (e.g., related to bus read-sharing and snarfing) are clearly specific to this type of workload. In practice, SMP systems are likely to spend most of their time running an assortment of single threaded programs concurrently. We do not test this particular type of application; however, our new protocol **Minerva** would behave exactly like a standard MESI cache protocol in such a case and fetch full blocks. Therefore, it would cause no performance degradation relative to a standard protocol.

Table 1 shows the reference characteristics of our workload using 16 processors and 4-byte blocks (to determine the number of truly shared words), as measured with a perfect single cycle memory system (when simulating with realistic cycle times, the number of references can also change slightly, due to changes in synchronization behavior, which is dynamically simulated at runtime and can be affected by changes in the number of misses and other stalls). The fraction of shared accesses (a shared access being a reference to a block that is accessed (at some time) by multiple processors) has quite a large variation; it ranges from 0.15 in **fmm** to 0.90 in **topopt** with an average value of 0.43. Due to memory space limitations in our trace generation machines, 16 processors was the maximum number we could simulate. The runs represent full execution of a problem and capture the entire behavior of the program. We picked 16 processors for our experiments to maximize the effects of parallel execution and sharing.

3.2 Bus Design and Timings

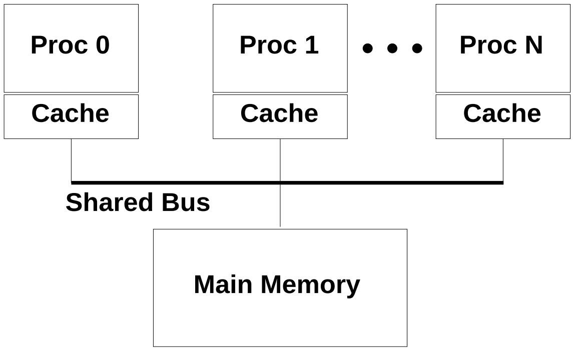


Figure 1: Bus-based shared memory design.

Our simulation testbed consists of 16 RISC processors communicating with an interleaved shared memory over a split-transaction 32-bit wide common bus (Figure 1). Each processor has its own cache, which snoops the bus for coherency information. The caches are dual-ported, to support snoopy cache (bus watch) operations without conflicting with the processors' accesses to their own caches. We simulate bus transactions cycle-by-cycle to provide the highest level of accuracy and to properly model contention.

Program Characteristics									
Programs	References (Millions)		Data Space		Fraction of Data References				
			Shared	Private	Shared			Private	
	Inst	Data	(KBytes)		Reads	Writes	Locks	Reads	Writes
barnes	114.23	42.31	33.02	34.64	0.16	0.00	0.00	0.46	0.38
cholesky	90.67	34.32	970.02	783.38	0.50	0.07	0.01	0.29	0.13
fmm	288.30	166.82	380.41	460.14	0.14	0.01	0.00	0.35	0.51
locus	805.62	164.45	1405.70	1151.79	0.56	0.02	0.00	0.26	0.16
mp3d	174.88	60.82	701.91	181.53	0.32	0.22	0.00	0.30	0.16
ocean	234.09	92.37	140.16	984.45	0.26	0.03	0.01	0.56	0.13
pthor	275.86	97.76	1233.09	1026.75	0.38	0.05	0.04	0.35	0.18
pverify	181.32	55.24	23.08	149.67	0.47	0.02	0.01	0.32	0.19
raytrace	471.08	196.94	667.30	2144.09	0.32	0.00	0.00	0.43	0.25
topopt	655.75	141.60	19.22	38.76	0.81	0.09	0.00	0.08	0.02
volrend	351.62	79.92	395.61	2340.98	0.48	0.01	0.00	0.29	0.23
water	366.23	127.67	44.51	102.37	0.18	0.02	0.00	0.58	0.23
Total					Average				
Overall	4009.6	1260.2	6014	9399	0.38	0.04	0.01	0.35	0.21

Table 1: Reference characteristics of programs for 16-processor simulation.

Our bus design incorporates the major features of modern high-performance busses. Table 2 shows the timing of the various bus operations. Times shown are in bus cycles, which are 4 processor cycles in duration. The timings we use are typical of a contemporary system with 400MHz processors, a 100 MHz shared bus, and 60ns SDRAM (6 bus cycle latency for the first word(s) with data available every bus cycle after that). The following paragraphs contain some of the operational details of our simulated bus.

Transaction	**Number of Bus Cycles**
Bus Arbitration	0/2/4 Cycles
Initiate Read	1 Cycle
Cache-to-Cache Read	1 Cycle per Bus-Width
Memory Latency	6 Cycles
Transfer Time	1 Cycle per Bus-Width
Write-back	1 Cycle per Dirty Bus-Width
Invalidation	1 Cycle
Lock operation	Read + 1 Cycle

Table 2: Duration of bus transactions. All transactions except memory response are preceded by bus arbitration.

To perform a read transaction on the bus, the processor must first arbitrate for the bus, except when it was the last agent to use the bus (bus parking). If the bus is busy when a bus request is made, the arbitration is pipelined and the bus requires only two cycles to grant bus ownership once the current bus transaction has ended. Four bus cycles are required to gain ownership when the bus is empty and the processor does not currently have ownership.

If one or more of the other caches has the requested data (which can be as large as a block or as small as a word), the data is immediately transferred between the caches, gathering the data from multiple caches if necessary (possibly with other caches passively receiving the data if *snarfing* or bus read-sharing is enabled). When cache-to-cache transactions occur, the request to main memory is automatically aborted. If the particular word that caused the cache miss is not available in any cache, the main memory responds to the request after a delay of six bus cycles. The final phase of data transfer requires 1 bus cycle for each bus-width of data transferred. In the case of a read-modify-write instruction (LOCK), the bus is held for an additional bus cycle to indicate whether the lock was successful and the transferred subblock should be invalidated in other caches. Some additional bus transactions specific to **Minerva** are described in Section 5.

Table 3 shows some of the characteristics of contemporary SMP systems using a common bus to interconnect multiple processors. As an example, the time to initiate a read operation is the bus arbitration time plus the Read time. The latency to get data from memory is presumably 60 ns (none of these sources specify the exact memory latency) plus the number of bus cycles required to transfer the data to the processor. As far as we were able to establish, these systems all use timings similar to those of our simulated bus. Not as much information as we would

Timings for Commercial Busses (Number of Bus Cycles)

Name	Cluster	Runway	AlphaServer	PowerPC 60x	Pentium Pro Bus
Vendor	SGI	HP	DEC	Motorola	Intel
Max Procs.	4	4	4	> 2	4
Block Size	64/128 bytes	32 bytes	32 bytes	32/64 bytes	32 bytes
Bus Arbitration	0/2/4 Cycles	2 Cycles	0–? Cycles	1–2 Cycles	0/2/4 Cycles
Invalidation	1 Cycle	1 Cycle	?	2 Cycles	7 Cycles
Read Request	1 Cycle	1 Cycle	?	2 Cycles	2 Cycles
Write Request	1 Cycle	1 Cycle	2 Cycles	2 Cycles	2 Cycles
Memory Latency[†]	12 Cycles	8 Cycles	25 Cycles	6 Cycles	6 Cycles
Transfer Time[‡]	16 Cycles	4 Cycles	2 Cycles	8 Cycles	4 Cycles
Width	64 bits	64 bits	128 bits	32/64 bits	64-bits
Transfer Size(s)	1–8 Bytes, Block	16–32 bytes	Block Only	1–8 bytes, Block	1–8, 16, 32 bytes
Bus Freq.	50–200MHz	120MHz	416MHz	66–100 MHz	100MHz
Ref.	[MIP96]	[BCF96]	[Hay94]	[Mot97]	[Int96, Alp99]

Table 3: Characteristics of commercial multiprocessor busses. Total time for a read transaction to occur is the sum of bus arbitration, read request, memory latency, and transfer time.

like appears in Table 3, as we were unable to determine some of the details in the references we found. These data points show that our model of bus behavior is within the values in existing shared bus systems, except that ours pushes the number of processors on the bus to 16 and uses a narrower bus (4-bytes) than most of the systems. In Section 6.4 we will evaluate the effect of changing the bus timings and bus width on the results.

3.3 Cache Structures

The simulated caches are incorporated on-chip with the processor, with single (processor) cycle access times. Two-level caches were not simulated in the system because the workload would not significantly exercise the second level of cache; a single level is sufficient for these programs. Instruction caches were ignored, since for these programs instruction references always hit in the cache, and because simulating the instruction cache would have significantly increased the time to evaluate each data point. The instruction misses would have increased the length of program execution only a few thousand cycles out of the 100s of millions to billions of cycles simulated, leading to imperceptible changes in the statistics we measured. Instead of simulating the instruction cache, we assume each instruction takes 1 processor cycle. The simulations used fully-associative caches to eliminate conflict misses caused by data-mapping artifacts in the data space, so the misses reported only contain cold-start, capacity and (mostly) coherence misses.

For the standard or base case, our experiments used the Illinois [PP84] write-invalidate-based protocol with fixed block size in a normal (non-sectored) cache. The Illinois protocol also forms the basis of the state changes of **Minerva** at the word level. We chose the Illinois protocol for two reasons: (1) it is an invalidation-based protocol, which generates less bus traffic than update-based protocols [Lil93] and generally outperforms update-based protocols [GS96]; and (2) write-invalidate protocols like the Illinois protocol are the most popular class of protocols that are actually implemented in real systems [Ste90, HP96], which makes them a more attractive target for performance improvement.

In addition to our dynamic subblock size invalidation protocol **Minerva**, fixed size subblock (sector cache) and full block protocols were simulated. The fixed size subblocks that were simulated ranged from 4 bytes up to the (64-byte) block size. The results we present show 64-byte blocks using the Illinois protocol (full block invalidation), 8- and 16-byte subblocks and the **Minerva** adaptive protocol. The cache sizes we tested were 4 to 128 Kbytes; for detailed examples we present the results for the 64 Kbyte caches.

4 Simple Performance Improvements

Before examining hardware and protocol means for the improvement of shared bus performance, we review two known methods of improving performance. These concentrate on taking advantage of the universal visibility of all transactions across the bus to better exploit the data stream. Figure 2 (with definitions in Table 4) shows the improvement these

[†] Assumes 60ns DRAM response time.

[‡] Estimated using maximum performance settings (widest, fastest) of bus parameters.

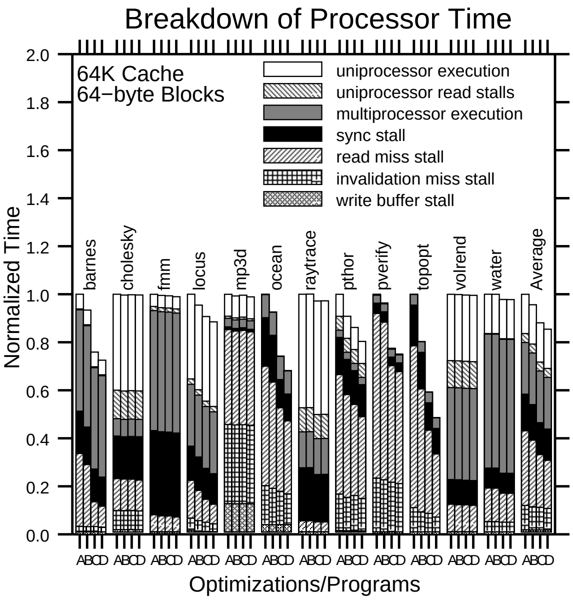


Figure 2: Breakdown of bus time (64K byte caches) for simple optimizations: (A) no optimizations, (B) bus read-sharing, (C) snarfing, and (D) both optimizations.

changes can make to the execution time of the various pieces of the workload, broken down into the various ways the processors spend their time. All the times are normalized with respect to execution time with no optimization (label A), showing the improvements for bus read-sharing (B), snarfing (C), and both optimizations together (D).

The first method, which we call *Bus Read-Sharing*, reduces the number of read transactions on the bus by allowing multiple stalled processors to share the results of a read operation. Each processor is capable of "seeing" the addresses of the transactions taking place across the bus. Occasionally a processor is stalled waiting its turn for the bus and sees another processor requesting the same data in which it is interested. Instead of initiating a read for the same location when it gets a chance at the bus, it passively waits for the data it wants to come over the bus. In such a manner, some transactions can be avoided. Some programs show little benefit from this optimization (**cholesky**, **fmm**, **mp3d**, **raytrace**, **volrend**, and **water**). The other programs show a significant amount of improvement (on the order of 12 to 51 percent), particularly those with large amounts of false sharing (cases B and D in Figure 2 for **barnes**, **locus**, **ocean**, **pthor**, **pverify**, and **topopt**).

The second simple improvement is read broadcast, also known as *snarfing* (case C, and together with bus read-sharing in D). Snarfing uses read transactions passing over the bus to fill empty (invalid) parts of blocks in the cache. Such an invalid block or subblock can occur in two ways. First, an invalidation may result in a slot in the cache containing an address tag, but with the "invalid" bit set to on. Second, a subblock (subsector) may be missing (invalid) in a sector cache for a variety of reasons. Since a snoopy cache is required to check its contents for most transactions that cross the bus, it is not much of an extension to identify addresses on the bus that match address tags for invalid blocks or subblocks, and grab the information broadcast on the bus. This is an almost free type of prefetching which requires no additional bus transactions, just a little more work from the cache. This form of performance enhancement has been included in Futurebus+ [Can90] and studied in [Dah95, AB95]. Snarfing serves as a balancing counterpart to invalidations, by allowing a read initiated by one processor to fill all invalid copies of a block across multiple processors, as invalidations invalidate all copies (but one) of a particular block. Both optimizations have beneficial effects; on average bus read-sharing reduces execution time by 4.4 percent, snarfing by 11.9 percent; together they reduce execution time by 14.5 for the 64K cache with 64-byte blocks. Naturally, this assumes that these operations do not have any negative impact upon the processor. Certainly bus read-sharing has no negative impact (the processor was waiting for the data anyway), but snarfing may interfere with processor operations if the cache does not have enough ports to serve both the bus and processor side interfaces (we assume it does). The source of the reductions comes, as would be expected, from reducing the amount of time the processor stalls for the various read operations (read, read-exclusive and read-modify-write). The Illinois protocol (to be explained in Section 5) with these two enhancements form the starting basis (normal case) for comparisons with the more hardware intensive protocols presented in Section 6.

One concern might be that snarfing might not work very well when invalid blocks are overwritten quickly, as might be the case when the cache size is small with respect to the working set. However, examination of snarfing for a smaller cache shows it to be still a useful optimization, as it reduces execution time by 6.5 percent for 16K caches (Figure 13 in Appendix B), and the two optimization together reduce execution time by 9.2 percent.

Definitions of Terms	
Uniprocessor Mode	Initialization time at the beginning of the program.
Multiprocessor Mode	Duration of program from the first time multiple processor threads are created.
Uniprocessor Execution	Time spent performing calculations in uniprocessor mode.
Uniprocessor Read Stalls	Read miss processing time in uniprocessor mode
Multiprocessor Execution	Time spent performing calculations in multiprocessor mode.
Sync Stall	Time spent waiting at barriers and locks.
Read Miss Stall	Read miss processing time in multiprocessor mode.
Invalidation Miss Stall	Stall time to invalidate blocks in other caches due to a write reference to a shared block (assumes sequential consistency).
Write Buffer Stall	Stall time waiting for the evicted cache block buffer to empty.
Uniprocessor Read	Fraction of time bus processes read misses in uniprocessor mode.
Uniprocessor Idle	Fraction of time bus is unused in uniprocessor mode.
Multiprocessor Idle	Fraction of time bus is unused in multiprocessor mode.
Multiprocessor Read	Fraction of time bus processes read misses in multiprocessor Mode.
Multiprocessor Inv.	Fraction of time bus processes invalidation operations in multiprocessor Mode.
Multiprocessor Write	Fraction of time bus processes dirty block evictions to main memory in multiprocessor mode.
Cache to Cache	Bus read request fulfilled by another cache.
Read Shares	Multiple bus read requests (to the same address) fulfilled by a single read transaction.
Uniprocessor Misses	Bus read request processed by main memory during uniprocessor Mode
Multiprocessor Misses	Bus read request processed by main memory during multiprocessor mode

Table 4: Definitions of terms in Figures 2, 7, 8, 10, 12, and in the text.

5 Minerva Protocol Description

Minerva is an invalidation-based protocol that maintains coherence state for each word in a block, which allows it to perform variable size fetch and invalidation transactions and correctly track the state of each word. The size of the transactions are based on each cache's estimate of the most appropriate subblock size for that block. Each word can independently be present or absent in the cache, as in sector caches for uniprocessors [RS99c].

A specification of the Minerva protocol requires several state diagrams; we describe **Minerva** here:

1. Consistency is maintained on a word basis in **Minerva**. Figure 3 shows the word state transitions, which are the same as the block transitions for the Illinois protocol [PP84]. Each word (or subblock for fixed size subblock protocols we evaluated) within a block has one of the MESI states associated with it (**modified** (**m**), **exclusive** (**e**), **shared** (**s**), or **invalid** (**i**)), depending on whether the data is different than the main memory value, only in one cache but consistent with memory, potentially shared among several caches, or not a valid word, respectively.

2. A block state is associated with each block as a whole, to track if copies of the block or portions of the block exist in other caches (Figure 4). If the block is in the **exclusive** state, the block (i.e., words from the block) is only in one cache. This allows write-validate operation, by which words can be written without first requiring the block be fetched from main memory [Jou93]. If the block is in the **shared** state, then write-validate is disabled.

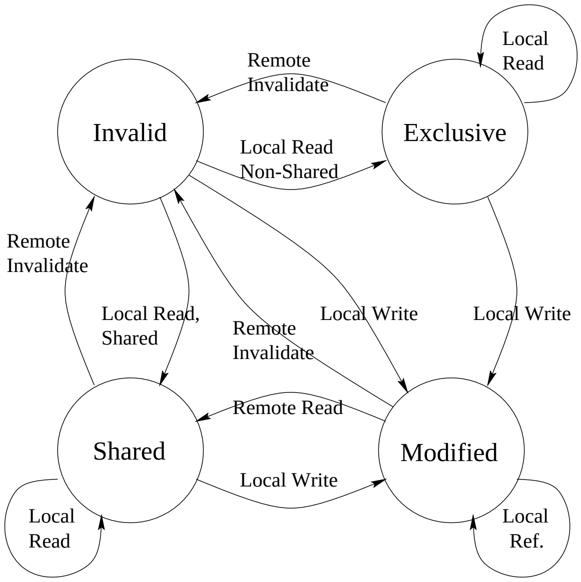


Figure 3: State transitions for coherence units using the Illinois protocol. These states are used in all the protocols evaluated.

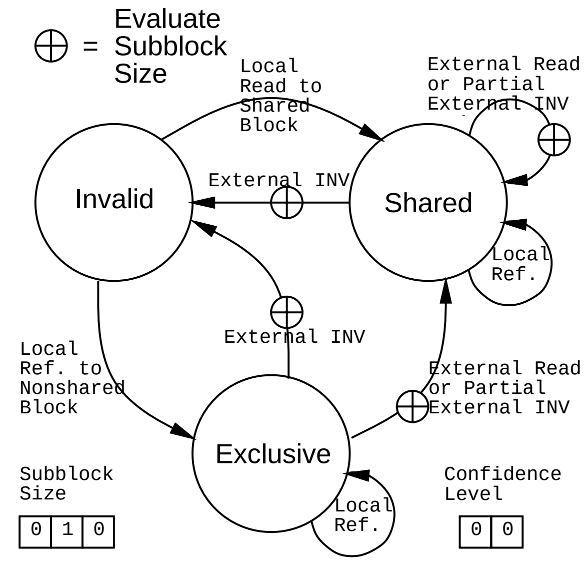

⊕ = Evaluate Subblock Size
Local Read to Shared Block
External Read or Partial External INV
Invalid
External INV
Shared
Local Ref.
External INV
Local Ref. to Nonshared Block
External Read or Partial External INV
Exclusive
Local Ref.
Subblock Size
0 1 0
Confidence Level
0 0

Figure 4: State transitions for blocks in the adaptive **Minerva** protocol. Any observed read or invalidate bus transactions to the block triggered remotely cause the protocol to reevaluate the subblock size based upon the number of words written locally.

3. Associated with each block in each cache is the current subblock size and a confidence level. The current subblock size consists of 2^k words, aligned on a 2^k word boundary. When a cache observes an external event (other than write-backs) associated with a block it contains, it evaluates the minimum subblock that encompasses all the most recently written words (i.e., words for which the "W-bit" (word write bit) is on — see below) for that block (transitions in Figure 4 with the $\bigoplus$ symbol). The confidence level is incremented (up to a saturation level, 3 for our experiments) if the newly calculated subblock is the same size as the old subblock size. Otherwise, the confidence level is decremented. Once the confidence level is 0, the new subblock size replaces the old subblock size. The confidence level aids in settling the subblock size on the most commonly measured value. Associated with each word in the block is a "W" (write) bit. The W-bit is set when the word is written. It is cleared whenever the subblock size is evaluated.

4. When a miss occurs on a reference to a given cache, the size of the subblock fetched is equal to the current subblock size for that block in that cache, except in one case. When a miss occurs to a block that is not present in any cache, the entire block is fetched; this is the only case in which the subblock size is ignored, because it speeds up fetching of non-shared data. The initial subblock size is 4 words (16 bytes), which was determined to be the best size by simulation.

5. When an invalidate signal is sent, it specifies precisely the words affected.

Note that different caches may have different subblock sizes for the same block at the same time. This means that the data for fetch misses may come from multiple bus agents, which is not an issue for fixed-size subblock protocols. When a fetch miss occurs, the processor issuing the fetch provides the address of the missing word and a bit vector of the particular subblock pieces it wants over the bus. If a requested (target) word is available from another cache, that word and as much of the subblock as is available is immediately supplied by the other cache. Since word state is maintained by the caches, unavailable words are just marked invalid. If the target missing word is not in any of the other caches, the main memory responds somewhat later with the missing data. Since the caches and the memory "know" which words are available from the caches, the memory supplies only the words unavailable in any of the caches. Caches supply the particular words they have in a cooperative manner, so the entire request is fulfilled with the latest values. It is possible that multiple agents containing the same word will drive the bus at the same time; however, only shared data can be available in multiple caches, so the values driven must be identical.

The protocol is aided by letting blocks that have no remaining valid subblocks stay in the cache until removed by the natural LRU block eviction process, to keep behavioral information (subblock size, confidence level, etc.) intact. For frequently accessed shared blocks in our workloads, there is a reasonably good possibility that the block will be fetched back into the cache again and the learned behavioral information will still be available. Experiments with smaller (16K) caches (Appendix B) show that some invalid blocks remain in the cache long enough to be reused by blocks with the same address tag (thus having access to behavioral data); however, under more strenuous workloads the invalid blocks may not survive long enough to be useful. Note that this type of behavior is a peculiarity of the particular workload we study — parallel shared memory programs. Such programs frequently have different processors operating on the same data over short time intervals. Other workloads would likely show very different behavior.

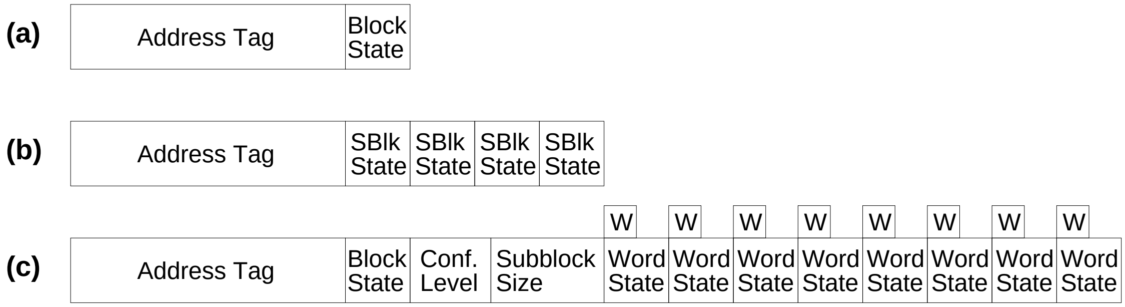


Figure 5: Information required for each block for (a) normal Illinois, (b) Illinois subblock coherence, and (c) the adaptive protocol. For case (c), the W-bit tracks recent writes to the words.

5.1 Implementation Details

Figure 5 provides a schematic of the information required to maintain coherence using (a) block granularity; (b) subblock granularity; and (c) the extra requirements for the **Minerva** protocol. The Illinois protocol (Figure 5a) requires the least number of bits, needing only the address tag (required by all protocols) and 2 bits to track the block state. For the pure subblock protocols (Figure 5b), 2 state bits are required for each subblock. For the adaptive protocol, a fair amount of additional information is required to provide the ability to dynamically change the subblock size (Figure 5c). In addition to the 2 bits to track the state for each word, the state of the block as a whole is tracked (whether it is exclusive, invalid, or shared), $\lceil log_2(1 + log_2 \frac{block\ size}{4})\rceil$ bits are used to track the subblock size (e.g., 3 bits for 64-byte blocks with five possible subblock sizes: 4, 8, 16, 32, and 64 bytes), and 2 bits as the confidence level (maximum value 3).

Associated with each word in each cache block (in our adaptive protocol) is a W (write) bit (Figure 5c) that tracks the words that have been written since the last external event to the block was observed over the bus (as defined in Figure 4). The intent is to provide the most accurate idea of what the adaptive subblock size should be. The W-bit is set whenever a write occurs to that word, and cleared after the subblock size has been evaluated.

A significant advantage the 64-byte block Illinois protocol has over the others presented here is the cost of implementation, in terms of tag bits and consistency and replacement logic. For a 64K 4-way set-associative data cache with 64-byte blocks and 32-bit addresses, full-block Illinois requires 21.5 bits for each block (18 address tag, 2 state bits, and 6 LRU (least recently used) bits per set). 64-byte block **Minerva** requires 74.5 bits (18 address tag bits, 34 state bits, 16 W-bits, 5 adaptive bits and 6 LRU bits for each set). This increases the number of bits associated with the data cache by 9.9 percent or about 6.78K bytes over the Illinois implementation. Since modern caches are likely to use longer addresses as well as set-associative cache organizations, the extra overhead per cache block would actually be less than the 9.9 percent estimated here. Based on the performance improvement the adaptive cache provides by reducing execution time and data traffic, it is well worth the investment in the extra bits.

An important feature of our dynamic subblock evaluation algorithm (and any useful cache coherence protocol) is that it is easily implementable in hardware. Figures 19 and 20 in Appendix C.1 show some of the combinational logic required for evaluating subblock size and maintaining the proper confidence levels. Determining the subblock size requires two gate levels of logic; a priority encoder then finds the minimum subblock size that spans all the written bits. The circuitry to compare the new subblock size with the old subblock size and determine whether to increment (or decrement) the confidence level requires another 4 gate levels of logic. Additionally some logic is required to increment or decrement the confidence level and to select which subblock size. So the entire logic to evaluate this algorithm is relatively small and can easily be performed in parallel with other actions occurring to the block (such as state changes, invalidation of data, etc.).

5.2 Example of Subblock Sizing

A demonstration of subblock sizing in **Minerva** is shown in Figure 6. In this example, each processor references a unique word within this block (a pattern observed in **topopt** and **pverify**), which causes a tremendous amount of false sharing activity for full block coherence. Before time 1, processor B has fetched the entire block (unshared blocks are fetched in their entirety) and written its word. The block

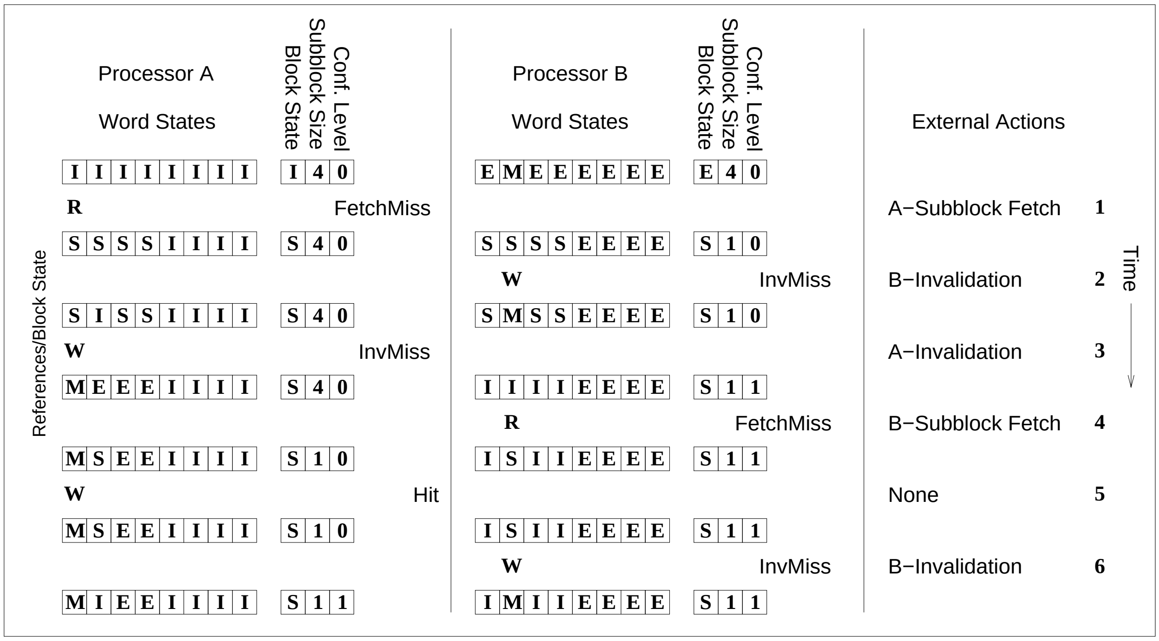


Figure 6: Adaptive handling of false sharing by **Minerva**.

state and most of the subblocks are in the exclusive state, except for the one written word in the modified state. A read by processor A at time 1 reads a subblock's worth of data (4 words) and the block (as well as some of the words) becomes shared between the two processors. Processor A evaluates its subblock size at times 4 and 6, and processor B evaluates its subblock size at times 1 and 3, which in all cases are the first external events observed by the cache after a local write has occurred. The first evaluation for both processors causes the subblock size to decrease from 4 words to 1 word; the second evaluation reinforces the single word subblock size by increasing the confidence level. Further read and write operations in this pattern cause no bus activity, potentially eliminating a significant number of bus transactions.

6 Results

This section presents the results of our multiprocessor simulations. We evaluate 4 different protocols using 64-byte blocks with 64K byte data caches per processor for each of the twelve programs. The four protocols consist of: (1) full-block 64-byte Illinois protocol (labeled **F** (for "fixed") in the figures); (2) a subblock protocol using eight-byte subblocks (**E**) (for "eight"); (3) a sixteen-byte subblock protocol (**S**) (for "sixteen"); and (4) our adaptive protocol **Minerva**, using an initial subblock size of sixteen bytes (**A**) (for "adaptive"). These labels are consistently used for the figures in the following sections. The subblock protocols fetch and invalidate data using fixed subblock size operations. The subblock states and state transitions are identical to the Illinois protocol (Figure 3). Each of the protocols evaluated in this section incorporate the simple performance enhancements described in Section 4 — snarfing and bus read-sharing.

Our simulations model all the events which occur between the processors and main memory (excluding instruction fetches), and use the bus timings previously presented. We simulated 16 processors for each workload, which is the maximum number that our trace generator could provide for all workloads.

6.1 Execution Time

Figure 7 shows the breakdown of processor execution time for 64K byte (per processor) data caches. The time is broken down into uniprocessor time: uniprocessor execution (actually performing calculations) and uniprocessor stall (waiting for the main memory to respond); and multiprocessor time: execution, read stall, write stall, and invalidation stall (all defined in Table 4). Synchronization stall is also measured, which is the time the average processor spends waiting to acquire locks and waiting at barriers. From this figure, we can see that neither the 8- nor the 16-byte subblock are consistently the best fixed subblock size, nor do they always outperform the normal (no subblock) case. **Minerva** outperforms the full-block

Illinois protocol in 11 out of 12 cases (slower by 1.4 percent for **cholesky**) and beats the fixed subblock sizes for all of the programs. On average, **Minerva** reduces execution time by 25.2 percent relative to the 64-byte full-block protocol for 64K byte caches (using arithmetic averages). It does this by reducing the stall time for both invalidations and read fetches by about half. Adding the other optimizations discussed in Section 4, the average improvement over the original unoptimized 64-byte block Illinois coherence protocol exceeds 33 percent. This is a tremendous improvement that can take place without having to recode any of the programs. Other changes such as weaker consistency models (which may require recoding the programs) can also be done in addition to these hardware changes, potentially providing a greater performance boost.

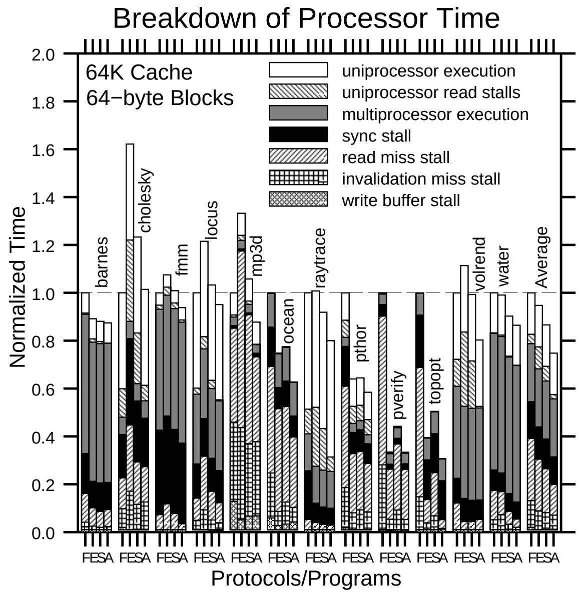


Figure 7: Execution time, normalized to full-block coherence, 64-byte blocks, 64K cache.

Synchronization stall time is fairly prominent in some of the programs. Much of this time is due to stalling at barriers when the workload is not well distributed. For example, **cholesky** uses a set of barriers to allow the master processor (processor 0) to execute in single processor mode in the middle phase of the calculation. This has the effect of wasting in excess of 15 percent of total execution time, which for that workload, exceeds the time that the processors spend actually performing calculations in multiprocessor mode.

Note that we use arithmetic averages in Figure 7 and other similar figures. As we will explain later in Section 6.4, we believe the geometric average is better for calculating the effect of various improvements in the execution time, but the arithmetic average is more suitable for these figures. However, this tends to underestimate the real performance improvement using the **Minerva** protocol, which reduces the average execution time by more than 40 percent as computed with a geometric average.

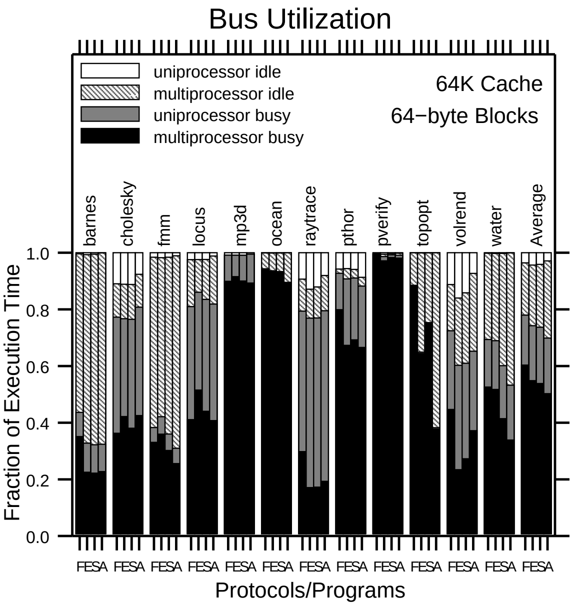


Figure 8: Bus utilization 64K cache with 64-byte blocks.

6.2 Bus Utilization

Figure 8 shows the absolute bus utilization for each of the programs in conjunction with each protocol. Using the adaptive **Minerva** protocol, the bus utilization actually decreases as the execution time decreases, from 78.0 percent busy for the full-block coherence to 69.9 percent busy. From the bus' perspective, much of the decline occurs by cutting the fetched data traffic (uniprocessor and multiprocessor read) by roughly half (for **Minerva** with 64K caches). All the times for the operations presented here include the times for setting up the transaction, including arbitration, address transmission and response time of caches (if appropriate). Invalidation bus utilization is very small due to the lack of data transmitted (invalidation operations only require 1 bus cycle plus arbitration). From Figure 7 we see that processors have to spend 10 percent of total time stalled waiting for an invalidation to occur (normal protocol), or 5 percent for **Minerva**, which is due almost exclusively to bus contention.

Breakdown of Bus Transactions												
Protocol	**Fetches (bytes)**						**Invalidations (bytes)**					
	4	8	16	32	64	total	4	8	16	32	64	total
64-byte Illinois	0	0	0	0	1.0	1.0	0	0	0	0	1.0	1.0
8-byte Subblocks	0	2.85	0	0	0	2.85	0	2.09	0	0	0	2.09
16-byte Subblocks	0	0	1.82	0	0	1.82	0	0	1.46	0	0	1.46
Minerva	0.21	0.03	0.81	0.06	0.16	1.27	0.39	0.08	0.60	0.08	0.11	1.28

Table 5: Total number of fetch and invalidation transactions relative to 64-byte Illinois with snarfing and bus read sharing enabled, broken down by operation size.

6.3 Other Metrics

Table 5 shows the number of fetch and invalidation transactions for each of the protocols, relative to the number of each transaction type for the 64-byte Illinois protocol, which are broken down by size. Note that we make a distinction between (fetch) misses and fetches. A fetch miss (as opposed to an invalidation miss) occurs when the requested data in not found in the cache. A fetch miss can be serviced by our bus read-sharing optimization, so not all fetch misses cause actual fetches to occur.

To illustrate the breakdown of the various transaction types, the 8-byte subblock protocol has 2.85 times as many fetches initiated compared to 64-byte Illinois, all of which are 8 bytes in size. Only the **Minerva** protocol has multiple transaction sizes, due to its adaptive subblock sizing. Of the subblock protocols, **Minerva** shows a far smaller increase in transactions relative to the fixed block (no subblock) case: 27 and 28 percent for fetches and invalidations, respectively. The fixed-size subblock protocols show a much larger increase in those operations, 185 and 109 percent for 8-byte subblock and 82 and 46 percent for 16-byte subblocks. The total number of bytes fetched and invalidated decreases with subblock size, as does data traffic for uniprocessor sector cache systems [RS99c], but since each transaction has overhead, it is necessary to balance the number and size of the transactions to reduce bus usage, as **Minerva** does. Additional statistics about the various protocols, such as the fetch miss ratios, can be found in Appendix A.

6.4 Sensitivity Analysis

To project this design to future architectures, it is necessary to determine how sensitive these results are to different timing parameters. For example, future designs will use more advanced DRAMs that have higher bandwidth, wider busses, and faster processors. Table 6 shows the effects of changing various memory system parameters. Unlike the figures and tables in in the previous sections (which use arithmetic averages because of the desire to show such features as the breakdown of execution time and bus utilization), Table 6 shows the geometric mean of the speedups achieved using various system parameters.

The results here show that the simple optimizations lead to a reduction of execution time (compared to unoptimized 64-byte block Illinois) of 6.5 to 16.0 percent, and the simple optimizations combined with the **Minerva** protocol lead to a reduction of execution time of 19.4 to 41.2 percent, depending on the configuration parameters. Generally the wider the bus, the less the **Minerva** protocol will outperform the Illinois protocol, but it is still a very significant performance improvement.

6.5 Effects of Software Restructuring

In [RS99a], we investigated the impact of certain data structure improvements on execution time. We found that changes in the data layout based on information obtained by profiling the worst behaving 10 (64-byte) blocks in four of our programs could lead to an average reduction of 51 percent of the false sharing misses and 20 percent of the total cache fetch misses (for an infinite cache), resulting in a 15 percent reduction in the execution time for both 16K and 64K byte caches (using 64-byte blocks). As a further evaluation of the **Minerva** protocol, we examine the effect of the protocol on code that has been restructured to eliminate many of the false sharing problems.

Table 7 shows the reduction of execution time for the four workloads, each restructured to reduce false sharing as in [RS99a], when the simple optimizations from Section 4 are applied and with the **Minerva** protocol. *Original* refers to the unoptimized source code; *restructured* refers to the modified code. For these four workloads, **Minerva** reduces execution time by 55.9 percent for the original code and 44.8 percent for the optimized code.

When both the original code and the optimized code are run under **Minerva**, the resulting execu-

Sensitivity of Performance to System Parameters

Cache Size	Bus Width (bits)	Bus Frequency (MHz)	Transfers per Cycle	Processor Freq. (MHz)	DRAM delay (ns)	Simple Opt. Reduction	**Minerva** Reduction	Comments
64K	32	100	1	400	60	16.0%	41.2%	**Standard Configuration**
16K	32	100	1	400	60	11.5%	38.3%	Small Cache
8K	32	100	1	400	60	6.5%	19.4%	Tiny Cache
64K	64	100	1	400	60	13.8%	29.6%	Wider Bus
16K	64	100	1	400	60	10.7%	27.0%	Smaller Cache and Wider Bus
64K	64	200	1	1000	50	14.5%	31.5%	1GHz Processor With Fast Memory
64K	64	300	2	1200	50	13.0%	28.6%	1.2GHz Processor With RAMBUS

Table 6: Execution time reduction vs. unoptimized Illinois with variation of simulation parameters.

Execution Time Reduction with Restructured Code

Program		Simple Opts	**Minerva**
Original	Barnes	27.4%	36.6%
	Pthor	19.7%	53.0%
	Topopt	51.4%	85.1%
	Water	2.2%	15.4%
	Average	27.4%	55.9%
Restructured	Barnes	25.4%	31.8%
	Pthor	19.0%	49.6%
	Topopt	50.3%	71.0%
	Water	1.9%	7.16%
	Average	26.4%	44.8%

Table 7: Execution time reduction of **Minerva** on restructured code. Geometric averages are used for the averages.

tion times are very similar. The optimized code is 2.9 percent faster than the unoptimized code, compared to the 15 percent reduction in execution time for optimized code when using the Illinois protocol [RS99a]. From these results it is possible to conclude that **Minerva** provides most of the performance improvement of code restructuring, but without the effort of actually restructuring the code.

6.6 Performance and Cost

Figure 9 shows the performance of various cache configurations, showing the number of bits (data, tag and other overhead) for 4 to 128 Kbyte caches, 64-byte blocks, and 4- to 64-byte subblocks. 4-byte subblocks generally have the worst performance for a given block size. 16- and 32-byte subblocks perform the best of the fixed subblock sizes. 8-byte and 64-byte subblocks have similar performance. The adaptive caches (indicated with the ⋆ symbols), i.e., those using **Minerva**, show the best performance overall. For the larger cache sizes (32K-128KB), the improvement in running time relative to the best of the other designs is 10.8% to 12.2%. The performance improvement using the **Minerva** protocol over the normal Illinois protocol is sufficient to outperform caches two to four times as large (for 16K caches and beyond). Even though the **Minerva** protocol requires more bits to manage a given amount of data space, the performance improvement more than justifies the cost.

6.7 Observations

Despite the significant improvement that the **Minerva** protocol provides, it is obvious that much more needs to be done in software to improve some of these programs. An examination of Figure 7 shows that only a tiny fraction of execution time in the parallel phase of some programs is used performing calculations (labelled **multiprocessor execution**). Such programs are **cholesky**, **mp3d**, **pthor**, **pverify**, and **topopt**. There is little point in trying to speed up the execution time by adding more processors to solve these problems, as the read and invalidation stall times heavily dominate the calculation time. There are other programs in which the initialization time in uniprocessor mode dominates the multiprocessor calculation time, also leading to poor speed-ups with more processors. Programs that fall into this category are **cholesky**, **locus**, **raytrace**,

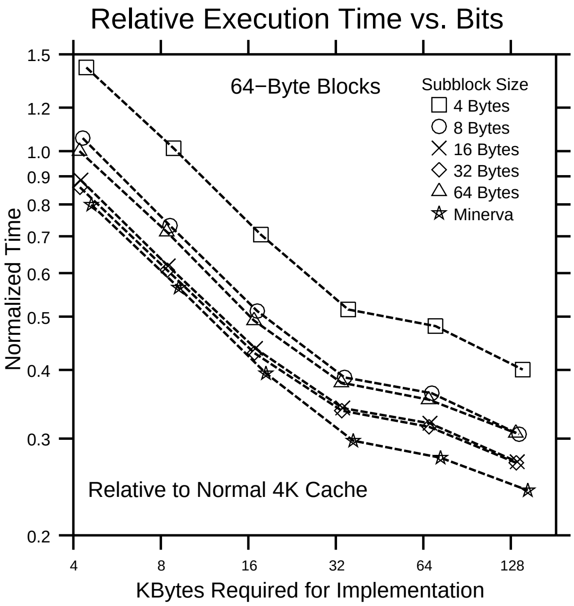


Figure 9: Relative execution time for a given number of bits, 64-byte blocks (1.0=4 Kbyte cache, 64-byte block, 64-byte subblock).

and **volrend**. This leads us to two general principles that should be considered when writing parallel programs: (1) uniprocessor initialization time should be minimized; methods must be explored to initialize the data space in parallel; and (2) data objects should be constrained as much as possible to single processors to reduce communications overhead. In addition, processors should be given tasks that are as independent as possible; barriers should rarely be used (unlike **cholesky** and **fmm**). Using these principles, programs should be designed to behave like **water**, where multiprocessor execution (calculation) time still heavily dominates the total run time, even with 16 processors on the job.

7 Conclusion

In this paper we have presented and evaluated our new cache coherency protocol **Minerva**. **Minerva** is a MESI-based protocol that allows the transfer block size to vary dynamically for each block in each cache, according to the reference pattern. Our new protocol in combination with other simpler and more general optimizations results in up to a 40 percent average reduction of execution time on a realistic shared memory machine. There is also a reduction of bus utilization, and only a small increase in the total number of fetch and invalidation transactions, much less than for fixed size subblock protocols.

Minerva achieves its success by reducing the invalidation size when it is wasteful to invalidate whole cache blocks. Smaller invalidation and fetch sizes aid in reducing the dead sharing traffic caused predominantly by false sharing behavior. By reducing bus traffic, processors spend much less time stalled waiting for bus transactions to complete and proportionally more time performing calculations.

An important feature of this protocol is that it can be simply and efficiently implemented in hardware, using a few levels of combinational logic to calculate the smallest subblock size every time certain external events affecting that cache block are observed on the bus. It also takes advantage of its ability to track word level coherence to implement write-validate, allowing writes to invalid words in order to reduce data fetches.

Our experiments with restructured code show that **Minerva** is able to largely overcome false sharing caused unintentionally by programmers. This frees the programmer from worrying about data layout and aids in fully realizing the simplicity of programming using the shared memory model.

References

[AB94] Craig Anderson and Jean-Loup Baer. Design and Evaluation of a Subblock Cache Coherence Protocol for Bus-Based Multiprocessors. Technical Report TR-94-05-02, University of Washington, May 1994.

[AB95] Craig Anderson and Jean-Loup Baer. Two Techniques for Improving Performance on Bus-based Multiprocessors. In *Proc. First IEEE Symposium on High-Performance Computer Architecture*, pages 264–275, Raleigh, NC, January 22–25 1995.

[Alp99] Donald Alpert, Intel Corp. Private communication, 1999.

[Arc88] J. K. Archibald. A cache coherence approach for large multiprocessor systems. *Proc. 2nd International Conference on Supercomputing*, pages 337–345, July 1988.

[BCF96] William R. Bryg, Kenneth K. Chan, and Nicholas S. Fiduccia. A High-Performance, Low-Cost Multiprocessor Bus for Workstations and Midrange Servers. *Hewlett–Packard Journal*, pages 1–7, February 1996.

[BS93] William J. Bolosky and Michael L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proc. USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–71, San Diego, CA, September 22–23 1993.

[Can90] Jay Cantrell. Futurebus+ Cache Coherence. In *WESCON/90 Conference Record*, pages 90–94, Anaheim, CA, November 13–15 1990.

[CD93] Yung-Syau Chen and Michel Dubois. Cache Protocols with Partial Block Invalidations. In *Proc. Seventh International Parallel Processing Symposium*, pages 16–23, Newport, CA, April 13–16 1993.

[Dah95] Fredrik Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 60–69, Santa Margherita Ligure, Italy, June 22–24 1995.

[DL92] Czarek Dubnicki and Thomas J. LeBlanc. Adjustable Block Size Coherent Caches. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 170–180, Gold Coast, Queensland, Australia, May 19–21 1992.

[DSS95] Michel Dubois, Jonas Skeppstedt, and Per Stenström. Essential Misses and Data Traffic in Coherence Protocols. *Journal of Parallel and Distributed Computing*, pages 108–125, September 1995.

[EJ91] Susan J. Eggers and Tor E. Jeremiassen. Eliminating False Sharing. In *Proc. 1991 International Conference on Parallel Processing*, pages I–377–I–381, St. Charles, IL, August 12–17 1991.

[EK88] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs And Its Applicibility to Coherency Protocol Evaluation. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 373–382, Honolulu, HI, May 30–June 2 1988.

[EK89a] Susan J. Eggers and Randy H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proc. 16th Annual International Symposium on Computer Architecture*, pages 2–15, Jerusalem, Israel, May 28–June 1 1989.

[EK89b] Susan J. Eggers and Randy H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, MA, April 3–6 1989. ACM.

[GH93] Stephen R. Goldschmidt and John L. Hennessy. The Accuracy of Trace-Driven Simulation of Multiprocessors. In *Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–157, Santa Clara, CA, May 10–14 1993.

[Goo87] James R. Goodman. Coherency For Multiprocessor Virtual Address Caches. In *Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 72–81, Palo Alto, CA, October 5–8 1987.

[GS96] Jeffrey D. Gee and Alay Jay Smith. Evaluation of Cache Consistency Algorithm Performance. In *Proc. Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 236–248, San Jose, CA, February 1–3 1996.

[Hay94] Fidelma M. Hayes. Design of the AlphaServer Multiprocessor Server Systems. *Digital Technical Journal*, 6(3):8–13, Summer 1994.

[HP96] John Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan-Kaufmann, 2nd-edition edition, 1996.

[HS84] Mark D. Hill and Alan Jay Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proc. 11th Annual International Symposium on Computer Architecture*, pages 158–166, Ann Arbor, MI, June 5–7 1984.

[Int96] Intel Corp. *Pentium Pro Family Developer's Manual*, 1996. Volume 1: Specifications.

[JE95] Tor E. Jeremiassen and Susan J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, July 19–21 1995.

[Jou93] Norman P. Jouppi. Cache Write Policies and Performance. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 191–201, San Diego, California, May 16–19 1993.

[KB95] Murali Kadiyala and Laxmi N. Bhuyan. A Dynamic Cache Sub-block Design to Reduce False Sharing. In *Proc. International Conference on Computer Design: VLSI in Computers and Processors*, pages 313–18, Austin, TX, October 2–4 1995.

[Lil93] David J. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[Lip68] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[MIP96] MIPS. *MIPS R10000 Microprocessor User's Manual*, 2.0 edition, October 1996.

[Mot97] Motorola. *PowerPC Microprocessor Family: The Bus Interface for 32-Bit Microprocessors*, March 1997.

[NS94] Håkan Nilsson and Per Stenström. An Adaptive Update-Based Cache Coherence Protocol for Reduction of Miss Rate and Traffic. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *Proc. PARLE '94 (Parallel Architectures and Languages Europe)*, pages 363–374, Athens, Greece, July 4–8 1994.

[PP84] Mark S. Papamarcos and Janak H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. 11th Annual International Symposium on Computer Architecture*, pages 348–354, Ann Arbor, MI, June 5–7 1984.

[Prz90] Steven Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. In *Proc. 17th Annual International Symposium on Computer Architecture*, pages 160–169, Seattle, WA, May 28–31 1990.

[RS99a] Jeffrey B. Rothman and Alan Jay Smith. Analysis of Shared Memory Misses and Reference Patterns. Technical Report UCB/CSD-99-1064, Computer Science Division, University of California, Berkeley, September 1999.

[RS99b] Jeffrey B. Rothman and Alan Jay Smith. Multiprocessor Memory Reference Generation Using **Cerberus**. In *Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '99)*, pages 278–287, College Park, MD, October 24-28 1999.

[RS99c] Jeffrey B. Rothman and Alan Jay Smith. Sector Cache Design and Performance. Technical Report UCB/CSD-99-1034, Computer Science Division, University of California, Berkeley, January 1999.

[RS99d] Jeffrey B. Rothman and Alan Jay Smith. The Pool of Subsectors Cache Design. In *Proc. International Conference on Supercomputing*, pages 31–42, Rhodes, Greece, June 20–25 1999.

[Sez94] André Seznec. Decoupled Sectored Caches: conciliating low tag implementation cost and low miss ratio. In *Proc. 21st Annual International Symposium on Computer Architecture*, pages 384–393, Chicago, IL, April 18–21 1994.

[Ste90] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–25, June 1990.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical report, Stanford University, June 1992. Report No. CSL-TR-92-526.

[TLH90] Josep Torrellas, Monica S. Lam, and John L. Hennessy. Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors. Technical report, Stanford University, February 1990. Report No. CSL-TR-90-412.

[TLH94] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[WOT^{+}95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24 1995.

A Analysis of Performance Related Metrics

This appendix contains statistics for 64 Kbyte caches with 64-byte blocks. These statistics demonstrate the manner in which **Minerva** works to improve performance as measured over a variety of metrics.

A.1 Cache Misses/Fetch Operations

Figure 10 show the number of fetches (normalized to the full-block coherence fetches) for the various protocols. The number of fetches is a proxy for the number of fetch misses; however, they are not exactly the same, since the bus read-sharing optimization (from Section 4) reduces the total number of fetches by allowing several caches to be satisfied with a single transaction. A miss occurs when the requested data is not found in the cache, but it may be satisfied without a fetch being issued if the desired data is observed on the bus.

All of the subblock protocols have more fetches than the full-block protocol, but **Minerva** causes fewer fetches than the other subblock protocols in general (32 percent for 64K caches). It is also the only protocol that has a variety of fetch sizes; the others all use a fixed fetch size. Having more misses (thus more fetches) is not necessarily bad. Because the data transferred on a miss is reduced for the subblock protocols, there is smaller delay for the information to be transferred. Thus an increase in the number of fetches may not yield an increase in the execution time. In addition, less data on the bus means that the delay waiting to start the fetch transaction will be reduced due to lower overall contention for the bus, leading to a significant reduction in delay for the whole read operation to complete. Because the processor-spatial locality of the problem shared blocks is very low [RS99a], the prefetching effect of large blocks is a waste of data and time. It is evident from Figure 7 that the read stall time is reduced by using the smaller fetch sizes. As demonstrated in Section 6.2, the amount of data and associated address transmission information is also greatly reduced by using subblocks.

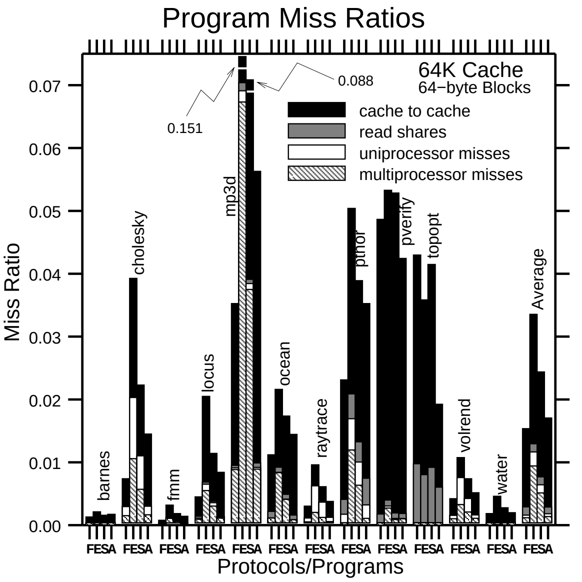


Figure 11: Workload of miss ratios for 64-byte block, 64K cache.

Figure 11 shows the actual miss ratios for the various programs. The misses are broken down to show how they were satisfied: uniprocessor misses (from main memory), multiprocessor misses (from main memory), by read-sharing an existing memory transaction (which could be coming from another cache or main memory), or cache-to-cache, which means the miss was satisfied by information found in another processor's cache. Of the various types of miss sources, only 3 (uniprocessor, main memory, or cache-to-cache) cause an actual fetch operation to occur. The vast majority of misses were satisfied by other caches (for the full-block protocol, 81 percent for 64K caches on average), indicating that we can visualize the aggregate system caches together as a distributed second level cache, but a less efficient "parallel" cache (as opposed to the serial "in-line" hierarchy in most two-level organizations). The aggregate caches in our simulations effectively form a 1M byte (64K cache) second level cache. For the 64K cache (Figure 11), the misses that actually fetch data from memory can be considered cold-start misses, as the aggregate capacity is sufficient to hold all the referenced locations. For the adaptive and the normal Illinois protocol (64K cache), the misses that must be fetched from memory are approximately 10 percent of total misses in the average case. This demonstrates the general importance of reducing coherence induced misses as opposed to the less prominent cold-start misses in

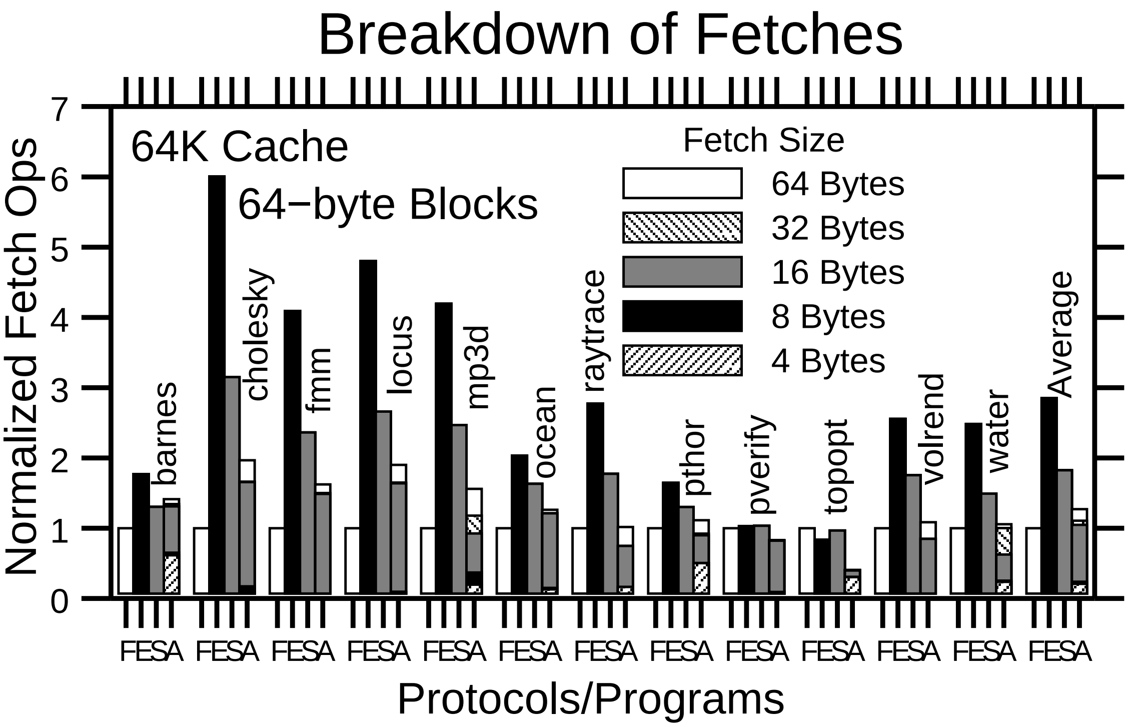


Figure 10: Fetch operations, normalized to full-block coherence, 64-byte block, 64K cache.

parallel workloads.

A.2 Invalidations

Unlike fetches, invalidations require the same number of bus cycles to perform, regardless of the size of the (sub)block invalidated. Reducing the size of the invalidation may cause a large increase in the number of these operations without providing any particular benefit if the workload has data structures well suited for large granularity invalidations. For some of the programs, the number of invalidations is greatly increased by subblock coherence. **Cholesky**, **locus** and **mp3d** are stand-outs in having greater than a three times increase in the number of invalidations when using 8-byte subblocks. This is the result of good processor-spatial behavior, which may be the result of data objects migrating between processors. From Figure 12, it can be seen that the fixed-block protocol has the fewest invalidations on average (although performs the worst for **raytrace** and **pverify**). The **Minerva** protocol generally has fewer invalidations than the fixed subblock protocols, and approximately 25 percent more invalidations than the fixed-block case. However, many of the invalidations are very small (33 to 39 percent are 4-byte invalidations), meaning that much of the data is left undisturbed when an invalidation occurs. Since processor-spatial locality is generally very small, this is a useful improvement, particularly with workloads that suffer from great amounts of false and dead sharing.

B Results for 16 Kbyte Caches

This Appendix shows results for 16 Kbyte caches with 64-byte blocks. This information is similar enough to the results presented in the main section so that is does not warrant inclusion there, but it is interesting enough to include as an appendix.

Figure 13 shows the effect of the bus read-sharing and snarfing improvements. Both optimizations have beneficial effects. On average bus read-sharing reduces execution time by 3.7 percent and snarfing by 6.5 percent. Together they reduce execution time by 9.2 percent for the 16K byte cache, with corresponding values of 4.6 percent, 11.9 percent, and 14.4 percent for the 64K cache.

In Figure 14, processors have to spend 10 percent of total time stalled waiting for an invalidation to occur (normal protocol), or 5 percent for **Minerva**, which is due almost exclusively to bus contention. Using the arithmetic average, **Minerva** reduces average execution time by about 23 percent over the Illinois protocol with the simple bus optimizations.

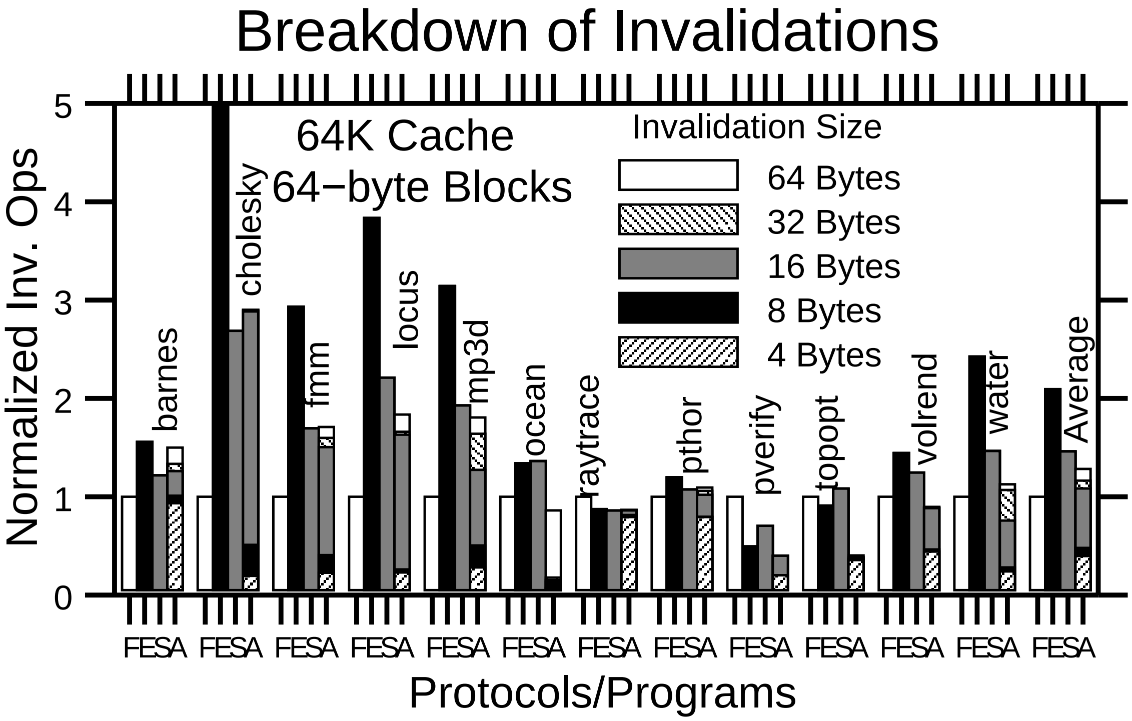


Figure 12: Invalidation signals issued by size, normalized to full-block coherence, 64-byte blocks, 64K cache.

Figure 15 shows the absolute bus utilization. The absolute bus idle time for the adaptive coherence does not change much compared between the various protocols on average; **Minerva** has only slightly more bus idle time.

Figure 16 displays the number and size of fetches for full-block, subblock and the **Minerva** protocols for 16K caches. All of the subblock protocols have more fetches than the "normal" protocol, but the **Minerva** protocol has fewer fetches than the other subblock protocols in general (40 percent increase in fetches for 16K caches, 32 percent for 64K caches). Note that the full-block and 16-byte fetches dominate the other size subblock fetches for the adaptive cache. There are other fetch sizes, ranging from 4 bytes to 32 bytes (as can be seen for such programs as **pthor**), but on average these other fetch sizes are almost invisible for 16K caches (7.2 percent are 4-byte fetches, 1.7 percent 8-byte fetches, and 1.0 percent 32-byte fetches). It is evident from Figure 14 that the read stall time is reduced by using the smaller fetch sizes.

The vast majority of misses were satisfied by other caches (for the "normal" protocol, 72 percent for 16K caches (Figure 17), 81 percent for 64K caches (Figure 11) on average with 64-byte blocks), indicating that we can visualize the aggregate system caches together as a distributed second level cache, but a less efficient "parallel" cache (as opposed to the serial "inline" hierarchy in most two-level organizations).

Figure 18 shows the number and size of the invalidation signals for each of the workloads for 16K caches. The number of invalidations for the subblock protocols relative to the Illinois protocol is roughly the same as for the 64K caches (Figure 12).

C Surplus Tables and Figures

This section contains tables representing the tabulated versions of the figures in the main sections plus additional figures that present related information.

C.1 Combinational Logic

To demonstrate that the computations to calculate the subblock size can be performed relatively easily, we implemented some of the circuitry using standard TTL style circuits. The circuits here use 32-byte blocks as an example; circuitry for larger blocks could be created in much the same fashion. Figure 19 calculates the subblock size based on the 8 input W-bit lines (one for each recently modified word in the block). If at least one word has been written in the upper and the lower 4 words of the block, then the subblock size is determined to be 8-words,

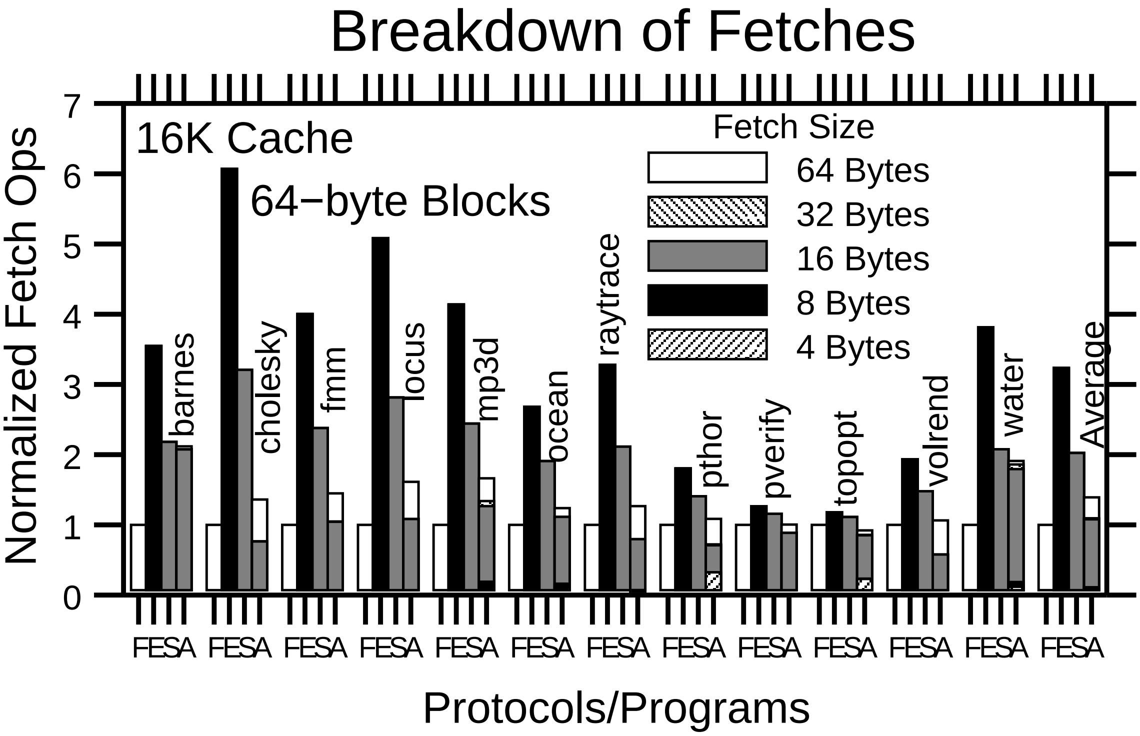


Figure 16: Fetch operations, normalized to full-block coherence, 64-byte block, 16K cache.

or full block size. Each of the 4-word portions of the block is divided and examined to see if the subblock spans the 4-word portion. Circuitry is included to only allow the appropriate output signal ([1-8]-word) for biggest subblock that spans all of the written to be set to 1. The **Any word** signal is 1 if any of the words have been written, indicating that a subblock size evaluation must be performed.

Figure 20 shows how the signals from Figure 19 are used to generate a 2-bit subblock size code, which is compared to the existing subblock size. If the new subblock size is the same as the old, the confidence level increment signal is activated, unless the counter has reached its maximum value. If the subblock sizes are different, the decrement signal is activated. If the confidence level is 0, then the replace signal is activated, which indicates that the newly computed subblock size replaces the old subblock size.

C.2 Tabulated Data

This section shows the tabulated versions (Tables 8–14) of all the figures in the main section and the 64K cache figures in Appendix A. Figures 21 and 22 show the performance achieved for a given number of bits for systems with 16- and 32-byte blocks, respectively. These are companion figures to Figure 9 in the main section and are also found in tabulated form in Table 8.

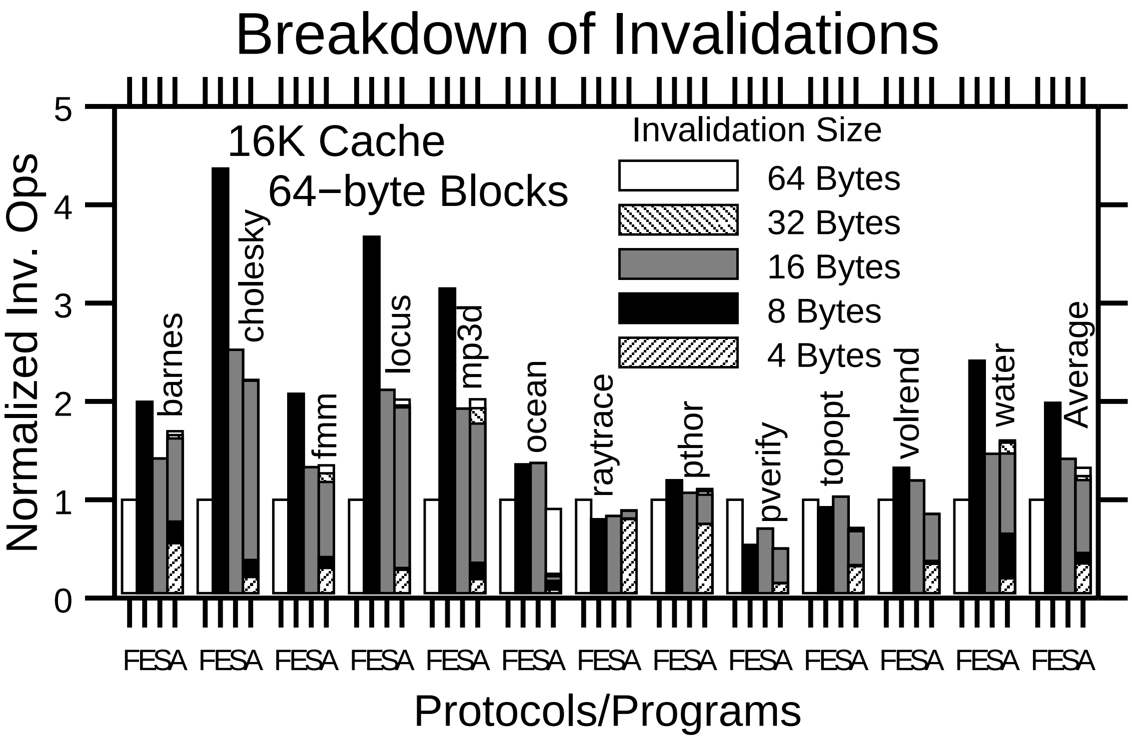


Figure 18: Invalidation signals issued by size, normalized to full-block coherence, 64-byte blocks, 16K cache.

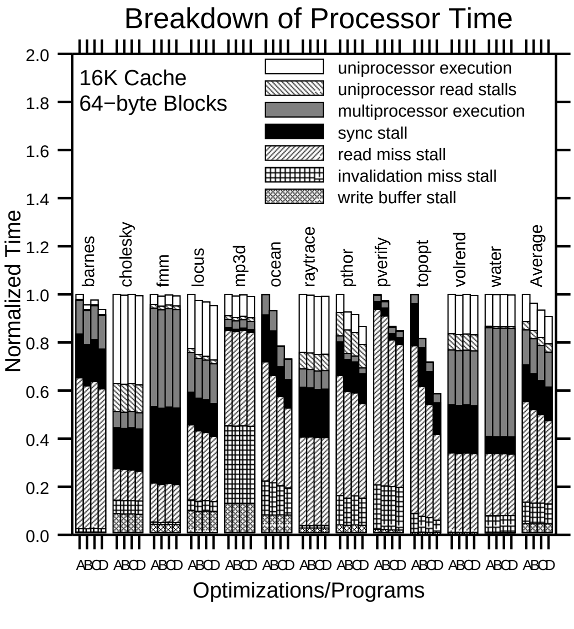


Figure 13: Breakdown of execution time (16K byte cache) for simple optimizations: (A) no optimizations, (B) bus read-sharing, (C) snarfing, and (D) both optimizations.

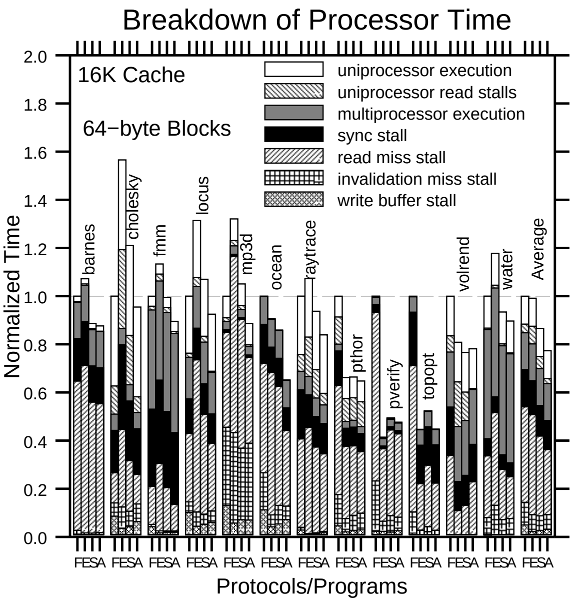


Figure 14: Execution time, normalized to full-block coherence, 64-byte blocks, 16K cache.

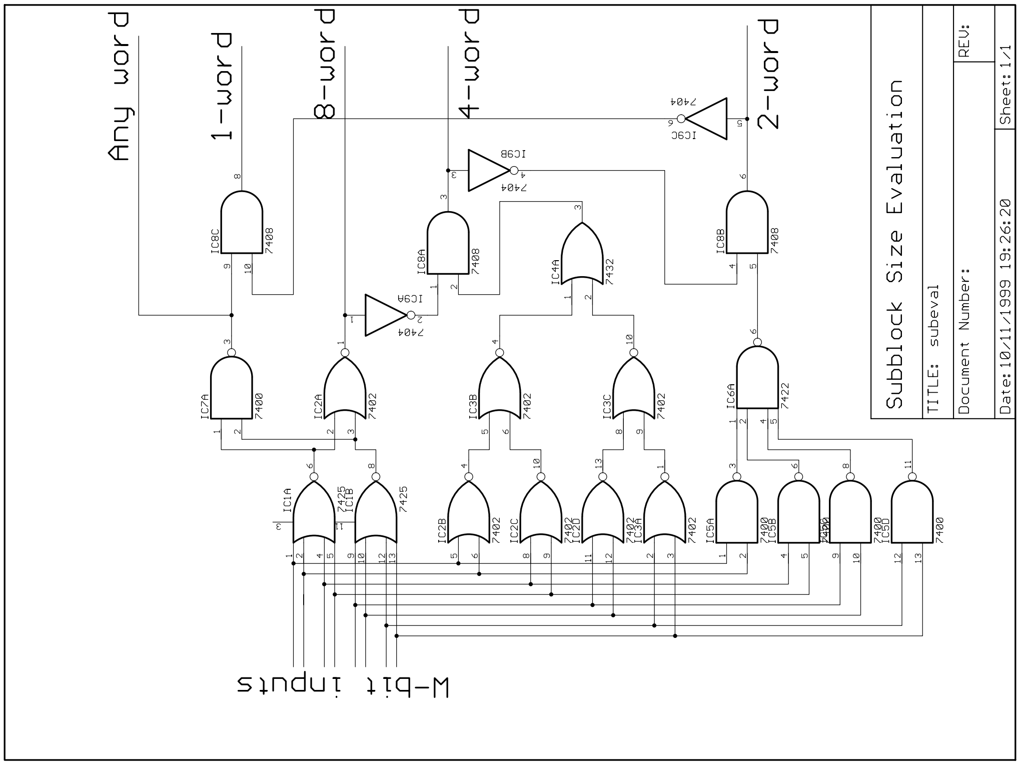


Figure 19: Combinational logic to evaluate subblock size from input W-bits, 32-byte block.

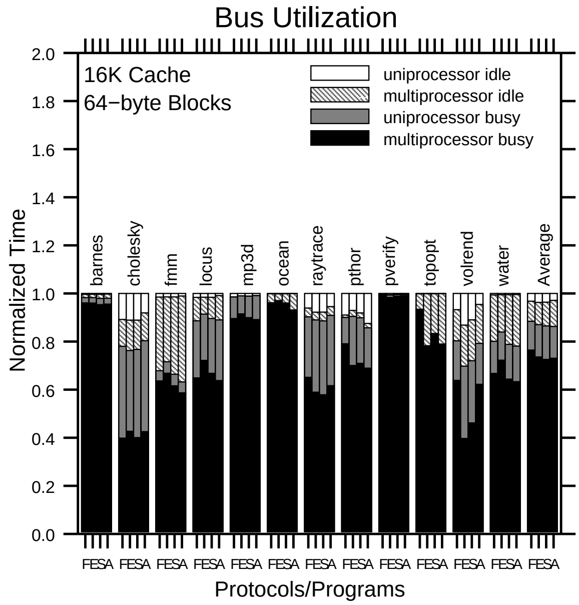


Figure 15: Bus utilization normalized to full-block coherence, 16K cache with 64-byte blocks.

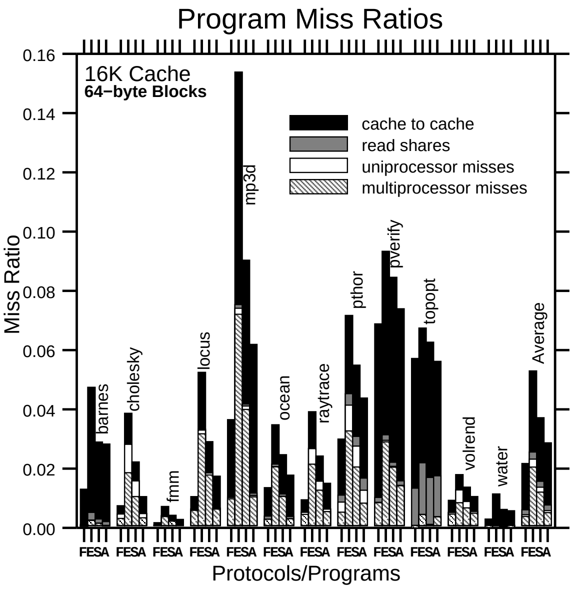


Figure 17: Workload miss ratios for 64-byte block, 16K cache.

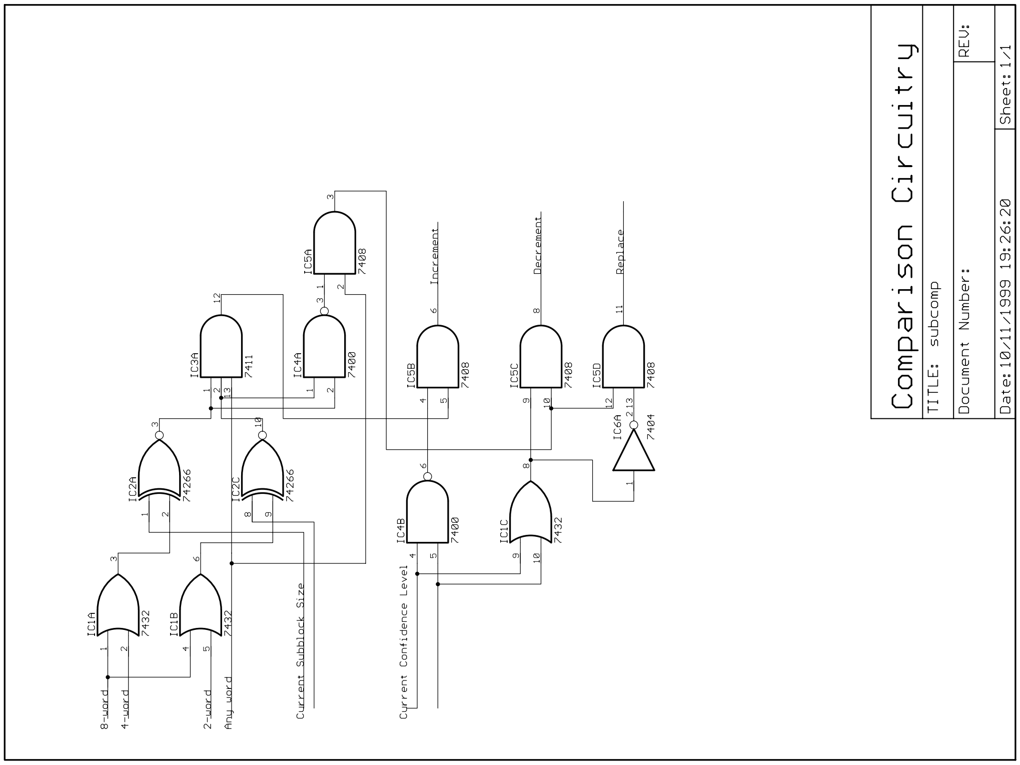


Figure 20: Combinational logic to compare subblock sizes, using outputs from Figure 19.

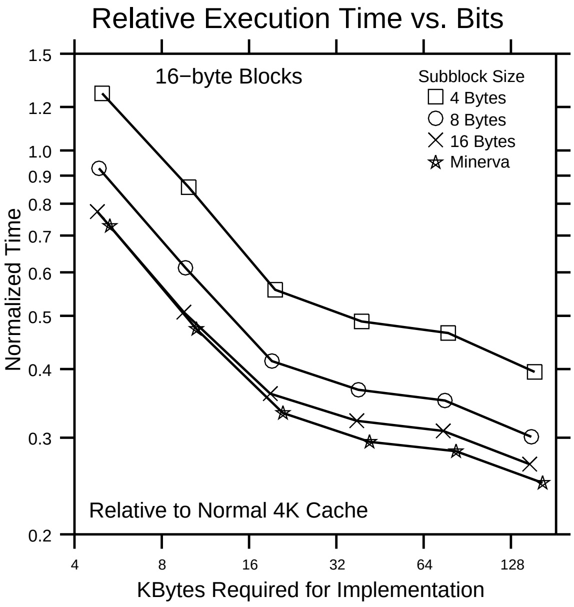


Figure 21: Relative execution time for a given number of bits, 16-byte blocks (1.0=4 Kbyte cache, 64-byte block, 64-byte subblock).

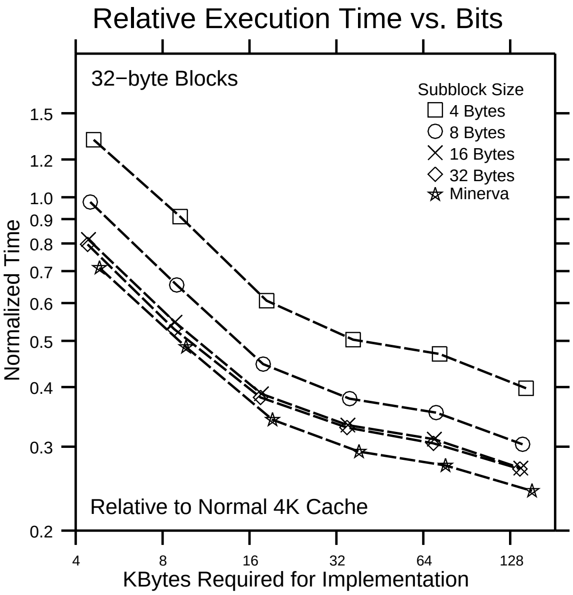


Figure 22: Relative execution time for a given number of bits, 32-byte blocks (1.0=4 Kbyte cache, 64-byte block, 64-byte subblock).

Relative Execution Time and Implementation Bits (in Kbytes)							
Cache Size	Block Size	Subblock Size (Bytes)					
		4	8	16	32	64	**Minerva**
4K	16	1.228 4.984	0.908 4.859	0.765 4.797			0.715 5.297
	32	1.285 4.617	0.957 4.492	0.804 4.430	0.791 4.398		0.695 4.836
	64	1.397 4.434	1.040 4.309	0.877 4.246	0.856 4.215	1.000 4.199	0.784 4.613
8K	16	0.809 9.906	0.587 9.656	0.495 9.531			0.458 10.531
	32	0.862 9.203	0.628 8.953	0.533 8.828	0.521 8.766		0.467 9.641
	64	0.962 8.852	0.704 8.602	0.602 8.477	0.597 8.414	0.707 8.383	0.543 9.211
16K	16	0.530 19.688	0.402 19.188	0.354 18.938			0.325 20.938
	32	0.574 18.344	0.430 17.844	0.378 17.594	0.374 17.469		0.329 19.219
	64	0.658 17.672	0.487 17.172	0.423 16.922	0.420 16.797	0.485 16.734	0.375 18.391
32K	16	0.463 39.125	0.355 38.125	0.315 37.625			0.287 41.625
	32	0.476 36.562	0.365 35.562	0.325 35.062	0.323 34.812		0.283 38.312
	64	0.487 35.281	0.373 34.281	0.332 33.781	0.331 33.531	0.373 33.406	0.286 36.719
64K	16	0.438 77.750	0.337 75.750	0.301 74.750			0.275 82.750
	32	0.441 72.875	0.340 70.875	0.303 69.875	0.299 69.375		0.264 76.375
	64	0.451 70.438	0.348 68.438	0.311 67.438	0.308 66.938	0.347 66.688	0.266 73.312
128K	16	0.363 154.500	0.284 150.500	0.257 148.500			0.237 164.500
	32	0.366 145.250	0.287 141.250	0.260 139.250	0.261 138.250		0.231 152.250
	64	0.369 140.625	0.289 136.625	0.261 134.625	0.263 133.625	0.299 133.125	0.231 146.375

Table 8: Relative execution time (1.0=4 Kbyte cache, 64-byte block, 64-byte subblock) and number of bits required for implementation (in Kbytes), tabulated information from Figures 9, 21, 22.

Breakdown of Relative Processor Execution Time (64K Per-Processor Cache, 64-Byte Blocks)														
		Programs												
Protocol	Description	barnes	cholesky	fmm	locus	mp3d	ocean	raytrace	pthor	pverify	topopt	volrend	water	Average
No Opt	uniexec	0.061	0.400	0.051	0.353	0.091	0.001	0.473	0.092	0.003	0.001	0.276	0.164	0.164
	uniread	0.004	0.118	0.017	0.022	0.009	0.001	0.100	0.058	0.001	0.000	0.112	0.002	0.037
	multiexec	0.422	0.072	0.500	0.258	0.036	0.096	0.150	0.030	0.033	0.044	0.383	0.558	0.215
	multisync	0.176	0.177	0.350	0.142	0.011	0.202	0.221	0.154	0.043	0.169	0.103	0.083	0.153
	multiread	0.303	0.133	0.073	0.157	0.395	0.498	0.052	0.497	0.686	0.674	0.121	0.140	0.311
	invstall	0.034	0.079	0.006	0.051	0.330	0.162	0.004	0.153	0.232	0.111	0.003	0.053	0.102
	writebuffer	0.000	0.020	0.003	0.017	0.128	0.040	0.000	0.016	0.003	0.000	0.001	0.000	0.019
Read Sharing	uniexec	0.061	0.400	0.051	0.353	0.091	0.001	0.473	0.092	0.003	0.001	0.276	0.164	0.164
	uniread	0.004	0.118	0.017	0.022	0.009	0.001	0.100	0.058	0.001	0.000	0.112	0.002	0.037
	multiexec	0.422	0.072	0.500	0.258	0.036	0.096	0.150	0.030	0.033	0.044	0.383	0.558	0.215
	multisync	0.156	0.176	0.350	0.139	0.011	0.195	0.222	0.146	0.043	0.152	0.104	0.084	0.148
	multiread	0.259	0.132	0.068	0.126	0.391	0.442	0.052	0.428	0.657	0.512	0.120	0.139	0.277
	invstall	0.032	0.079	0.006	0.043	0.328	0.151	0.004	0.139	0.224	0.094	0.003	0.053	0.096
	writebuffer	0.000	0.020	0.003	0.014	0.127	0.040	0.000	0.015	0.003	0.000	0.001	0.000	0.019
Snarfing	uniexec	0.061	0.400	0.051	0.353	0.091	0.001	0.473	0.092	0.003	0.001	0.276	0.164	0.164
	uniread	0.004	0.118	0.017	0.022	0.009	0.001	0.100	0.058	0.001	0.000	0.112	0.002	0.037
	multiexec	0.422	0.072	0.500	0.258	0.036	0.096	0.150	0.030	0.033	0.044	0.383	0.558	0.215
	multisync	0.136	0.178	0.349	0.129	0.011	0.117	0.197	0.140	0.036	0.114	0.102	0.083	0.133
	multiread	0.103	0.130	0.068	0.096	0.393	0.348	0.048	0.380	0.485	0.347	0.119	0.121	0.220
	invstall	0.033	0.079	0.006	0.038	0.329	0.139	0.004	0.147	0.215	0.087	0.003	0.051	0.094
	writebuffer	0.000	0.020	0.003	0.012	0.128	0.041	0.000	0.015	0.003	0.000	0.001	0.000	0.019
Both Opts	uniexec	0.061	0.400	0.051	0.353	0.091	0.001	0.473	0.092	0.003	0.001	0.276	0.164	0.164
	uniread	0.004	0.118	0.017	0.022	0.009	0.001	0.100	0.058	0.001	0.000	0.112	0.002	0.037
	multiexec	0.422	0.072	0.500	0.258	0.036	0.096	0.150	0.030	0.033	0.044	0.383	0.558	0.215
	multisync	0.121	0.181	0.349	0.127	0.011	0.112	0.198	0.134	0.036	0.106	0.103	0.083	0.130
	multiread	0.087	0.128	0.064	0.082	0.389	0.304	0.048	0.340	0.467	0.263	0.117	0.120	0.201
	invstall	0.030	0.078	0.006	0.033	0.327	0.128	0.004	0.135	0.209	0.072	0.003	0.051	0.090
	writebuffer	0.000	0.020	0.003	0.011	0.127	0.040	0.000	0.015	0.003	0.000	0.001	0.000	0.018

Table 9: Breakdown of bus time for simple optimizations (same data as in Figure 2).

Relative Number (by Size) of Fetches (64K Per-Processor Cache, 64-Byte Blocks)														
		Programs												
Protocol	Size (bytes)	barnes	cholesky	fmm	locus	mp3d	ocean	raytrace	pthor	pverify	topopt	volrend	water	Average
Illinois	64	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
8-Byte Subblock	8	1.771	6.003	4.092	4.801	4.195	2.032	2.774	1.645	1.032	0.836	2.557	2.485	2.852
16-Byte Subblock	16	1.305	3.150	2.363	2.662	2.468	1.634	1.775	1.302	1.035	0.967	1.755	1.493	1.826
Minerva	64	0.070	0.304	0.123	0.252	0.381	0.051	0.269	0.191	0.008	0.004	0.237	0.055	0.162
	32	0.034	0.006	0.013	0.011	0.254	0.001	0.000	0.024	0.000	0.006	0.000	0.376	0.060
	16	0.658	1.484	1.420	1.543	0.556	1.060	0.583	0.394	0.730	0.089	0.820	0.372	0.809
	8	0.037	0.081	0.025	0.013	0.172	0.022	0.002	0.002	0.000	0.004	0.001	0.016	0.031
	4	0.615	0.092	0.044	0.084	0.199	0.131	0.164	0.504	0.093	0.305	0.028	0.238	0.208

Table 10: Number and size of data fetches, relative to the number of fetches for standard 64-byte block Illinois protocol, 64K caches (same data as in Figure 10).

Breakdown of Relative Processor Execution Time (64K Per-Processor Cache, 64-Byte Blocks)														
Protocol	Description	Programs												
		barnes	cholesky	fmm	locus	mp3d	ocean	raytrace	pthor	pverify	topopt	volrend	water	Average
Illinois	uniexec	0.084	0.401	0.051	0.398	0.092	0.001	0.486	0.114	0.003	0.002	0.277	0.167	0.173
	uniread	0.005	0.119	0.017	0.025	0.009	0.001	0.103	0.072	0.002	0.000	0.113	0.002	0.039
	multiexec	0.582	0.073	0.505	0.291	0.036	0.141	0.154	0.038	0.044	0.091	0.385	0.570	0.242
	multisync	0.167	0.181	0.353	0.143	0.011	0.164	0.203	0.166	0.048	0.219	0.103	0.085	0.154
	multiread	0.120	0.129	0.064	0.092	0.393	0.445	0.049	0.424	0.621	0.541	0.118	0.123	0.260
	invstall	0.042	0.078	0.006	0.038	0.331	0.188	0.004	0.168	0.278	0.148	0.003	0.052	0.111
	writebuffer	0.000	0.020	0.003	0.012	0.129	0.059	0.000	0.018	0.003	0.000	0.001	0.000	0.020
8-Byte Subblock	uniexec	0.084	0.401	0.051	0.398	0.092	0.001	0.486	0.114	0.003	0.002	0.277	0.167	0.173
	uniread	0.014	0.340	0.034	0.051	0.020	0.002	0.247	0.072	0.004	0.001	0.311	0.006	0.092
	multiexec	0.582	0.073	0.505	0.291	0.036	0.141	0.154	0.038	0.044	0.091	0.385	0.570	0.242
	multisync	0.110	0.361	0.367	0.157	0.009	0.088	0.080	0.088	0.021	0.166	0.095	0.078	0.135
	multiread	0.077	0.276	0.109	0.224	0.738	0.429	0.039	0.247	0.209	0.098	0.044	0.097	0.216
	invstall	0.025	0.160	0.009	0.083	0.385	0.059	0.001	0.071	0.049	0.039	0.001	0.073	0.080
	writebuffer	0.000	0.011	0.001	0.010	0.052	0.027	0.000	0.009	0.005	0.000	0.000	0.000	0.010
16-Byte Subblock	uniexec	0.084	0.401	0.051	0.398	0.092	0.001	0.486	0.114	0.003	0.002	0.277	0.167	0.173
	uniread	0.009	0.212	0.023	0.034	0.014	0.002	0.174	0.064	0.002	0.000	0.198	0.004	0.061
	multiexec	0.582	0.073	0.505	0.291	0.036	0.141	0.154	0.038	0.044	0.091	0.385	0.570	0.242
	multisync	0.119	0.255	0.351	0.138	0.007	0.106	0.073	0.091	0.024	0.163	0.088	0.074	0.124
	multiread	0.067	0.177	0.072	0.120	0.539	0.399	0.031	0.244	0.276	0.180	0.044	0.054	0.184
	invstall	0.020	0.104	0.005	0.043	0.303	0.092	0.001	0.083	0.089	0.068	0.001	0.033	0.070
	writebuffer	0.000	0.012	0.001	0.008	0.067	0.034	0.000	0.010	0.003	0.000	0.000	0.000	0.011
Minerva	uniexec	0.084	0.401	0.051	0.398	0.092	0.001	0.486	0.114	0.003	0.002	0.277	0.167	0.173
	uniread	0.001	0.065	0.010	0.004	0.002	0.000	0.060	0.063	0.001	0.000	0.007	0.001	0.018
	multiexec	0.582	0.073	0.505	0.291	0.036	0.141	0.154	0.038	0.044	0.091	0.385	0.570	0.242
	multisync	0.113	0.203	0.337	0.134	0.015	0.088	0.071	0.082	0.023	0.162	0.081	0.072	0.115
	multiread	0.071	0.148	0.030	0.084	0.355	0.294	0.028	0.203	0.209	0.037	0.052	0.034	0.129
	invstall	0.023	0.114	0.004	0.033	0.310	0.061	0.002	0.074	0.049	0.016	0.001	0.021	0.059
	writebuffer	0.000	0.012	0.001	0.007	0.068	0.042	0.000	0.011	0.004	0.000	0.000	0.000	0.012

Table 11: Execution time, normalized to Illinois protocol execution time (same data as in Figure 7).

Absolute Bus Utilization (64K Per-Processor Cache, 64-Byte Blocks)														
Protocol	Description	Programs												
		barnes	cholesky	fmm	locus	mp3d	ocean	raytrace	pthor	pverify	topopt	volrend	water	Average
Illinois	uniidle	0.004	0.110	0.016	0.024	0.009	0.001	0.093	0.058	0.001	0.000	0.112	0.002	0.036
	multiidle	0.560	0.118	0.601	0.166	0.000	0.056	0.113	0.015	0.002	0.115	0.163	0.304	0.184
	unibusy	0.085	0.410	0.052	0.399	0.092	0.001	0.496	0.128	0.004	0.002	0.278	0.168	0.176
	multibusy	0.351	0.363	0.331	0.411	0.899	0.942	0.298	0.799	0.993	0.883	0.447	0.526	0.603
8-Byte Subblock	uniidle	0.007	0.111	0.018	0.024	0.009	0.001	0.129	0.056	0.006	0.001	0.160	0.003	0.044
	multiidle	0.665	0.122	0.561	0.115	0.000	0.062	0.102	0.036	0.008	0.350	0.238	0.308	0.214
	unibusy	0.103	0.345	0.062	0.345	0.076	0.003	0.598	0.235	0.015	0.005	0.368	0.172	0.194
	multibusy	0.225	0.422	0.360	0.515	0.915	0.933	0.171	0.673	0.971	0.644	0.234	0.517	0.548
16-Byte Subblock	uniidle	0.005	0.112	0.016	0.024	0.009	0.001	0.121	0.059	0.003	0.000	0.142	0.002	0.041
	multiidle	0.673	0.123	0.624	0.141	0.000	0.066	0.110	0.030	0.004	0.246	0.249	0.396	0.222
	unibusy	0.100	0.385	0.058	0.395	0.091	0.002	0.597	0.218	0.010	0.003	0.337	0.187	0.199
	multibusy	0.222	0.380	0.302	0.440	0.900	0.931	0.172	0.692	0.983	0.750	0.273	0.414	0.538
Minerva	uniidle	0.001	0.076	0.011	0.012	0.005	0.000	0.081	0.087	0.002	0.000	0.073	0.001	0.029
	multiidle	0.675	0.116	0.679	0.169	0.000	0.104	0.124	0.031	0.007	0.617	0.275	0.467	0.272
	unibusy	0.096	0.383	0.054	0.411	0.102	0.002	0.602	0.217	0.010	0.005	0.281	0.194	0.196
	multibusy	0.227	0.425	0.256	0.407	0.893	0.894	0.193	0.666	0.981	0.377	0.372	0.339	0.502

Table 12: Absolute bus utilization (same data as in Figure 8).

Miss Ratios Broken Down by Type and Source (64K Per-Processor Cache, 64-Byte Blocks)														
		Programs												
Protocol	Description	barnes	cholesky	fmm	locus	mp3d	ocean	raytrace	pthor	pverify	topopt	volrend	water	Average
Illinois	cache2cache	0.0011	0.0043	0.0006	0.0031	0.0257	0.0090	0.0019	0.0184	0.0473	0.0329	0.0026	0.0018	0.0124
	readshares	0.0002	0.0000	0.0000	0.0003	0.0004	0.0010	0.0000	0.0024	0.0017	0.0098	0.0000	0.0000	0.0013
	unimiss	0.0000	0.0014	0.0000	0.0002	0.0003	0.0000	0.0006	0.0016	0.0001	0.0000	0.0005	0.0000	0.0004
	multimiss	0.0000	0.0015	0.0002	0.0009	0.0088	0.0011	0.0005	0.0001	0.0000	0.0000	0.0010	0.0000	0.0012
8-Byte Subblock	cache2cache	0.0018	0.0173	0.0019	0.0136	0.0809	0.0122	0.0033	0.0282	0.0488	0.0275	0.0031	0.0045	0.0203
	readshares	0.0001	0.0000	0.0001	0.0003	0.0013	0.0008	0.0000	0.0039	0.0011	0.0082	0.0000	0.0000	0.0013
	unimiss	0.0001	0.0090	0.0001	0.0010	0.0018	0.0001	0.0042	0.0048	0.0003	0.0000	0.0043	0.0000	0.0022
	multimiss	0.0001	0.0099	0.0011	0.0055	0.0673	0.0082	0.0020	0.0114	0.0027	0.0000	0.0033	0.0001	0.0093
16-Byte Subblock	cache2cache	0.0014	0.0105	0.0011	0.0076	0.0494	0.0123	0.0025	0.0247	0.0502	0.0318	0.0031	0.0027	0.0164
	readshares	0.0001	0.0000	0.0000	0.0002	0.0007	0.0008	0.0000	0.0033	0.0009	0.0092	0.0000	0.0000	0.0013
	unimiss	0.0000	0.0050	0.0001	0.0005	0.0009	0.0000	0.0024	0.0036	0.0002	0.0000	0.0021	0.0000	0.0013
	multimiss	0.0000	0.0054	0.0006	0.0030	0.0375	0.0041	0.0012	0.0061	0.0009	0.0000	0.0021	0.0001	0.0051
Minerva	cache2cache	0.0016	0.0113	0.0011	0.0070	0.0464	0.0127	0.0025	0.0265	0.0400	0.0134	0.0035	0.0020	0.0140
	readshares	0.0001	0.0000	0.0000	0.0001	0.0007	0.0007	0.0000	0.0042	0.0008	0.0067	0.0000	0.0000	0.0011
	unimiss	0.0000	0.0013	0.0000	0.0002	0.0003	0.0000	0.0007	0.0021	0.0001	0.0000	0.0005	0.0000	0.0004
	multimiss	0.0000	0.0016	0.0002	0.0011	0.0089	0.0009	0.0006	0.0009	0.0010	0.0000	0.0011	0.0000	0.0014

Table 13: Miss ratio data, broken down into unimiss (main memory access during uniprocessor mode), multimiss (main memory access during multiprocessor mode), cache2cache (miss serviced by another processor's cache), and readshare (miss serviced by using data from another processor's fetch transaction) (same data as in Figure 11).

Relative Number (by Size) of Invalidations (64K Per-Processor Cache, 64-Byte Blocks)														
		Programs												
Protocol	Size (bytes)	barnes	cholesky	fmm	locus	mp3d	ocean	raytrace	pthor	pverify	topopt	volrend	water	Average
Illinois	64	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
8-Byte Subblock	8	1.559	4.970	2.934	3.836	3.143	1.340	0.874	1.198	0.494	0.910	1.445	2.426	2.094
16-Byte Subblock	16	1.218	2.688	1.695	2.212	1.929	1.363	0.860	1.074	0.705	1.084	1.246	1.466	1.462
Minerva	64	0.165	0.008	0.110	0.176	0.165	0.683	0.006	0.036	0.001	0.017	0.009	0.059	0.119
	32	0.073	0.009	0.094	0.029	0.368	0.024	0.001	0.038	0.000	0.012	0.004	0.311	0.080
	16	0.251	2.374	1.100	1.371	0.769	0.019	0.047	0.224	0.198	0.010	0.419	0.477	0.605
	8	0.076	0.316	0.178	0.031	0.222	0.062	0.021	0.003	0.001	0.009	0.022	0.040	0.082
	4	0.935	0.196	0.228	0.231	0.283	0.073	0.794	0.794	0.202	0.357	0.444	0.241	0.398

Table 14: Number and size of invalidations, relative to the number of invalidations for standard 64-byte block Illinois protocol, 64K caches (same data as in Figure 12).