Improving Performance of Bus-Based Multiprocessors

by

Craig S. Anderson

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1995

Approved by_______________________________
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree_______________________________

Date_______________________________

University of Washington

Abstract

Improving Performance of Bus-based Multiprocessors

by Craig S. Anderson

Chairperson of Supervisory Committee: Professor Jean-Loup Baer
Department of Computer Science
and Engineering

Processors have become both cheaper and faster in recent years, to the point where it has become practical to use multiple processors in a single system. An important aspect of such systems is how processors are connected to each other and to main memory. The simplest design uses a single, common bus. The main problem with single bus multiprocessors is the lack of bus bandwidth, which limits the number of processors that can be connected effectively. This dissertation addresses this problem by examining protocol improvements for single-bus multiprocessors as well as investigating the use of clustering to increase the number of processors that can be connected together effectively.

We developed a subblock cache coherency protocol to enable programs to take advantage of a program's spatial locality, while avoiding false sharing. In addition, the use of read broadcasting is investigated. In nearly all cases, the subblock protocol alone performed as well or better than the best execution of a standard invalidate protocol. Simulation results also show that using read broadcast substantially reduces the number of block re-reads due to invalidation when false sharing is present. Using read broadcast with the subblock protocol further increases the protocol's performance.

This dissertation also proposes and evaluates a cache coherence protocol that dynamically chooses either an update or invalidate-based coherence scheme on a block-by-block basis based on the application's write patterns. The hybrid adaptive scheme performs as well as or superior to the better of the two standard protocols.

The second part of this dissertation concerns the use of processor clustering to increase the number of processors that can be connected together effectively. A detailed invalidation-based hierarchical bus protocol is presented. Next, the benefits of moving main memory into the clusters and allocating data there are tested using simulation. The results show that cluster allocation had mixed results, improving performance for some applications while showing no benefit for others. The subblock protocol mentioned above is extended to the hierarchical bus system. Simulation results show that the benefits of using the subblock protocol are nearly as good as those obtained on the single bus architecture.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

Foremost, I would like to thank my advisor, Jean-Loup Baer. I appreciate very much his diligence and patience in assisting and guiding me towards the completion of my studies at the University of Washington. I value his willingness to read through many drafts of this document. I'm also grateful that he was always accessible, even when on his hard-earned sabbatical. Without his help, I would not have completed my work at the University. Gaetano Borriello also provided early guidance to my work here.

I would like to thank the staff of the computing laboratory at the University of Washington. Their efforts to answer my many questions speedily and to support the computational requirements of my research are noted. The research in this dissertation required a massive amount of CPU time. A small (but important) fraction of that time was obtained on machines donated by Intel Corporation. The lion's share of time, however, was obtained by using idle cycles on other users' machines. My colleagues' tolerance of my use of their resources is appreciated.

My undergraduate years were spent at a large, impersonal public university. It was a delightful surprise to find such a helpful office staff in the Department of Computer Science and Engineering. Frankye Jones in particular was always there with practical advice, useful information, and skilled action in dealing with the university's bureaucracy. Her advocacy for the department's graduate students is commendable.

> Graduate school takes a sizable chunk of one's life, and is an experience much broader than researching and writing a dissertation.
>
> — *Susan Eggers*

One does not attend graduate school in a vacuum. Many friends and colleagues provided me with good technical advice and encouraged me to persist, both when times were easy and especially when they were not. They include: Robert Bedichek, Bob Benway, Lauren Bricker, Donald Chinn, David Cohn, Susie Hashisaki, David Keppel, Alex Klaiber, Neil Mckenzie, Dylan McNamee, Elizabeth Seamans, and Rick Zucker. I'm especially grateful to those who helped broaden my experiences, both inside and outside of school.

I would like to thank Tekla Haasl, who in many ways made the months leading up to the completion of this document more bearable.

Finally, I would like to thank my family for encouraging me throughout my many years in school.

Chapter 1

INTRODUCTION

In the quest for reducing the time required to execute computationally intensive programs, computer architects have turned to using multiple processors in a single computer system. In theory, running a CPU-bound job on an n processor system with n parallel threads will result in the job executing n times faster than if it is run on an equivalent uniprocessor. In practice, however, applications often fail to achieve such *linear* speedups. One reason is that the problem itself may not have enough inherent parallelism to keep all available processors busy. But even if an application has a great deal of possible parallelism, another reason for non-linear speedups is the delay imposed by a multiprocessor's memory system, which is used not only to connect processors to memory, but also for interprocessor communication (we include synchronization in the communication component). Thus, a great deal of effort has gone into reducing latency and increasing bandwidth in multiprocessor memory systems.

There are many choices to be made when designing a multiprocessor memory system. One of the most important choices is the type of physical interconnect that links processors to other processors and to memory. The type of interconnect, whether it be a bus, hypercube, crossbar, or some other architecture, influences the other choices a memory architect faces. Some of those choices are [HP90]:

Uniform vs. Non-Uniform Memory Access In a Uniform Memory Access machine (UMA), all of physical memory is located in the same location, hence access to it incurs a uniform time cost regardless of which part of physical address space is being accessed[1]. In contrast, in a Non-Uniform Memory Access architecture physical memory is distributed around the machine, which means that some portion of physical memory is closer to a given processor than other

[1] Delays caused by contention for resources can cause variability in memory access times.

parts of the address space. Since it usually requires more time to access physically distant memory modules, memory access times vary depending on which address is being accessed. In order to minimize communication, an application's data should be placed so that a processor accesses local (as opposed to remote) memory as much as possible. This data layout problem creates additional work for the programmer or compiler. NUMA machines are often called distributed memory machines.

Load/store vs. message-passing
Communication to and from remote processors/memory modules can be made *implicit* to the programmer via normal loads and stores, or can be made via *explicit* calls to message-passing routines that send and receive data. Message-passing architectures are usually more difficult to use, because the programmer must directly manage data communication. In a load/store architecture, the hardware performs whatever actions are necessary to access the proper data storage, providing a homogeneous view of memory. While this saves work by the programmer, it also conceals the real cost of a memory reference, which can make it difficult to get optimal program performance.

Coherent vs. non-coherent memory Most multiprocessors cache data to decrease memory latency and load on the interconnect. If data is communicated via load and stores, writes done to shared locations can cause other copies to become out of date. An architecture can provide the programmer with a coherent view of memory, so that processors observe changes to data made by other processors. Coherence may be enforced using hardware, software, or a combination of the two. While providing a coherent view of memory makes the programmer's job easier, it complicates the implementation of an architecture.

Because writing multiprocessor applications is more difficult than writing serial code, architectures that make the programmer's job easier, i.e., those architectures that present a homogeneous, coherent view of memory (using loads and stores) have gained a great deal of popularity. Such architectures are called *shared-memory* machines. Though shared memory has been implemented using rings (e.g. KSR [FIR93]) and meshes (DASH [SJGH93], Alewife [ABC+95]), busses are a common choice because of their simplicity. In addition, busses have an inherent broadcast capability which makes it easier to implement coherent memory through snoopy protocols.

This thesis presents several techniques for improving performance of shared-memory machines based on bus interconnects. We first present our evaluation methodology, plus a brief overview of architectural evaluation strategies in Chapter 2. We follow with Part I of the thesis, which examines improvements to single bus multiprocessors. In Chapter 3 we evaluate two techniques, called subblocking and read snarfing, to improve the performance of snoopy cache coherence protocols. The subblock protocol addresses the problem that the best size for cache blocks varies from application to application. Read snarfing reduces the cost of keeping data coherent by taking advantage of busses' ability to do inexpensive broadcast. The two techniques are studied both separately and in combination.

Chapter 4 describes and assesses two algorithms that adapt their strategy for maintaining cache coherence based on the application's reference behavior. The two strategies alternate between updating and invalidating remote copies of cache blocks, depending on the past reference behavior for each block to pick the correct strategy.

In Part II of the thesis, we examine hierarchical bus systems. Hierarchical systems take advantage of the fact that often data is shared between only a few processors. Hierarchical bus systems exploit this property (called communication locality) to increase the number of processors that can be effectively connected together. In a hierarchical system, processors are grouped into clusters. Each cluster has its own bus to which processors are connected. In turn, clusters are connected to each other via an inter-cluster bus. In a three (or more) level system, groups of clusters are connected via a higher level bus. Connection between two bus levels is made using a combination bus cache/connector which not only connects the part of the system below the connector to the rest of the machine, but also provides additional cache space for the elements below.

In Chapter 5, we present a cache coherence protocol for a hierarchical bus system. Such a protocol is more complicated than one for a single bus, since there is no single serializing resource as there is in a single bus machine. In Chapter 6 we evaluate the two-level hierarchical architecture running our coherence protocol. We also extend the subblock protocol and read snarfing to the hierarchical architecture, and assess what effects they have on system performance when compared against the base coherence protocol.

Chapter 7 examines the problem of data placement in a hierarchical system in which main memory is distributed among the system's clusters. We investigate the efficacy of giving the programmer control over where data is allocated in the system. We then evaluate programmer-controlled data allocation against essentially random allocation, and also compare the performance of the distributed memory architecture against the base architecture as detailed in Chapter 5.

Finally, in Chapter 8, we summarize the results of this dissertation and state our research contributions.

Chapter 2

METHODOLOGY

2.1 Evaluation strategies

Evaluating new architectures, especially multiprocessor architectures is difficult. Multiprocessor architectures are much more complex than single processor systems. Multiprocessors often include features like cache coherence, complicated interconnects, and resource contention that are not usually found in uniprocessors. Because of this, evaluating multiprocessors requires more complicated models or simulators than single processor systems. Before introducing the methodology used in this thesis, we will briefly examine general evaluation strategies.

2.1.1 Analytical modeling

Perhaps the fastest way to evaluate a new system is to use an analytical model, in which mathematical equations are used to generate architectural performance estimates. A common methodology used to generate such equations is *queueing network models* [LZGS84]. In the context of evaluating multiprocessors systems, a model's service centers represent the various components of the multiprocessor, and queueing theory is used to solve for the delays imposed by these service centers. The great advantage of modeling is speed — most analytic models require no more than a few minutes to iteratively solve a model's equations. Unfortunately, there are drawbacks to modeling that limit its usefulness. One of the problems is determining what input values should be used. Model input values typically include processor request and cache miss rates, proportion of writes versus reads and the fraction of accesses that are to shared variables. Estimating what input values should be used is a difficult problem. Another problem with models is that in order to make them simple enough to be evaluated analytically, many hardware elements, such as cache size, associativity, block size, and protocol type must be abstracted away. Since varying these elements often dramatically affects system performance, analytical models cannot be used to evaluate the performance effects of changing such hardware elements.

Despite such drawbacks, analytical models are useful at quickly providing initial performance estimates for simplified systems.

2.1.2 Hardware prototypes

Another approach is to use actual hardware to evaluate a proposed architecture, as in [LLJ+92]. Prototypes have a distinct advantage over using simulation or analytical models; namely that applications and the underlying operating system run at full speed with no inaccuracies in the performance results due to the use of simplified models. In contrast, both analytical models and simulators have been criticized for being either slow or of questionable validity (or both). Building a prototype also can expose design problems that aren't found when using a simulator.

Regrettably, there are major drawbacks in building a prototype. First, building a prototype machine requires much effort (typically tens of man-years). Second, once the hardware is built it is difficult to change it, which limits the exploration of alternative designs. Third, monitoring the performance of the hardware requires that extra hardware be used — for the DASH prototype, 20% of the directory board is devoted to performance monitoring hardware. Lastly, not all architectural parameters can be measured on the hardware. For example, the first level cache hit rate and processor utilization cannot be measured in the DASH prototype.

2.1.3 Simulation

A much more widely used technique is architectural simulation. Simulation has several advantages over both analytical models and actually building hardware. First, a simulator can model very detailed workings of the target system, albeit at a cost in both programming effort and speed of simulator execution. Simulators usually don't require estimates of cache miss rates or workload characteristics that analytical models require. It is also relatively easy to change various target architectural parameters, such as cache size and associativity, interconnect size and speed, and other variables, which is very difficult to do in hardware. Simulators also can be used to evaluate future or idealized hardware, which is impossible to build.

Unfortunately, the flexibility of simulation comes at a price. Simulators often are complicated to write and debug. In addition, simulations often require large amounts of resources in order to produce meaningful results. What resources are required depends on the type of simulation performed. Two major types of simulation are *trace-driven* and *execution-driven* (instruction-level) simulation.

Trace Driven Simulation

Trace driven simulation is often used when evaluating memory systems [SJF91]. In trace driven simulation, a list, or trace, of memory references is gathered from an program, usually while it is executing on real hardware. The trace is stored on disk, and is then read by the simulator. One drawback of trace-driven simulation is that the trace files are often very large, which means that storing traces is sometimes a problem.

Gathering accurate traces is often difficult. Gathering a trace using dedicated hardware has the advantage that the tracing activity minimally disturbs the system as a whole; unfortunately, such hardware is often expensive, non-portable, and sometimes cannot observe all memory events, especially if the processor has on-chip caches. Traces generated using software techniques avoid the problems of hardware generated tracing, but have their own problems, including time and memory dilation. Time dilation refers to the slowdown in the traced application because of the execution of tracing instructions. A large slowdown can drastically change the behavior of a program, because the time spent in trace generation overwhelms the normal timing differences between the execution of different instructions on different processors [KKEL90]. Koldinger *et al.* examined the problem of trace dilation, and concluded that there was little difference in aggregate behavior of course-grained programs when their tracing tool's time dilation was artificially increased from 2 to 250 [KEL91]. However, they caution that dilation may be more important with medium or fine-grain programs, and that even a dilation factor of 2 may be enough to significantly affect the validity of a trace.

Memory dilation is another source of inaccuracy in traces that include operating system references [BLRC94]. Memory dilation occurs when the traced program puts an additional load on the virtual memory system because it is larger than the original untraced program. One way around this problem is to collect the trace on a machine with a large amount of physical memory (to avoid paging), and to simulate, rather than trace translation-lookaside buffer behavior.

Another problem with traces in general is that a particular trace records a particular order of memory reference events. Unlike the case for uniprocessors, changing hardware parameters can change both the type and order of memory events. Unfortunately, this change in program behavior will not be reflected in the trace.

Execution-Driven Simulation

An alternative to trace-driven simulation is execution-driven simulation, in which the simulation is performed as the target program executes. This allows the memory system simulator to control when the target program runs, because the processor can be stalled until the memory reference completes (according to the memory consistency model). In this way, the effect of memory system behavior on processor execution can be modeled. Another advantage of execution-driven simulation over using traces is that since the memory reference stream is generated and consumed at the same time, no need for large amounts of storage for traces is needed. The downside to these advantages is that the memory reference stream must be generated each time a simulation is run, which costs additional time over that needed by the memory simulator.

There are a wide variety of methods used to simulate program execution [CK94]. Some methods used, along with multiprocessor simulators that use the method are:

Decode and Dispatch Interpreter The target processor is simulated by fetching, decoding, and executing each instruction as it is encountered (Cerberus [BAD89], CacheMire [BDNS93]).

Pre-decode interpreter Target instructions are pre-decoded into an intermediate representation that execute more quickly than undecoded form. The predecoded instructions are usually cached for reuse (Talisman [Bed95], MINT [VF94a]).

Note: The above techniques are often classified as program-driven simulators. Program-driven simulators interpret the instructions of the target machine. While this can be more costly than direct execution, program-driven simulators can run on a variety of host architectures, and can simulate architectures which haven't been implemented in hardware.

Direct execution with annotated binaries This technique can be used when the host and target instruction sets are the same. The target binary is instrumented so that references to shared memory locations call the memory simulator (Proteus [BDCW91], Tango Lite [Her93]).

Direct execution using traps Instead of augmenting the target binary to call the memory simulator, the host hardware traps to the memory simulator when necessary, usually on a simulated cache miss (Wisconsin Wind Tunnel [RHL+93]).

Other aspects of various simulators include whether or not the simulator can run system-level code; whether it will handle program-generated signals correctly; and whether the simulator is robust in the face of program errors like dividing by zero [CK94]. Some multiprocessor simulators can take advantage of multiple host processors to speed up the simulation process. Unfortunately, parallel architecture simulators usually have a great deal of interprocessor communication and synchronization which limit the speedup of the simulator [Fuj90].

2.2 Our evaluation methodology

We chose a program-driven simulator, Cerberus [BAD89], for our architectural research. Cerberus was already being used by another student in the department, which made it easy to learn about the tool. Though Cerberus is an old simulator that lacks features and time-saving optimizations common to current simulators, it has been a valuable tool for performing architectural research.

The original Cerberus, as well as its enhancements to accommodate various memory consistency models [ZB92], simulates a shared-memory multiprocessor with processor-cache pairs connected to shared-memory modules via a multi-stage interconnection network. We retained the processor simulation component of Cerberus and replaced the cache, interconnection network, and memory module packages. Replacing the interconnection network had the added benefit of speeding up simulations by an order of magnitude, because simulating even a hierarchy of busses is much faster than simulating an interconnection network cycle by cycle. The emulated processor is similar to a Ridge-32, a RISC-like processor which only uses loads and stores to access memory [Rid]. The simulator executes Ridge binaries which are produced using a C compiler included in the simulator package.

The processor module simulates (at a crude level) the pipeline delays of a RISC processor. Simple instructions, like integer addition, logical operations, and register-to-register moves require one cycle. Integer multiplies and floating point additions require 2 cycles, as do branches that are incorrectly predicted by the compiler (branch instructions include a static-prediction bit). Floating point multiplies and integer divides require 4 cycles, while floating point divides require 10 cycles to complete.

All instruction references are assumed to hit in the instruction cache with no additional delay. All load and stores to non-shared data are assumed to hit in a separate, private cache and take one cycle to complete. Shared references that hit in the simulated cache and require no further action by the memory system also require a single cycle to finish. If a shared reference requires the use of the memory system, the delay is determined by simulating the memory system. Caches are assumed to have dual-ported tag structures so that snooping accesses don't interfere with the processor's cache access.

All busses in our simulations are split-transaction on non cache-to-cache transactions. Arbitration for bus access is done on a round-robin basis, and requires one cycle. We assume the bus has separate address lines so that addressing information may be transmitted at the same time as data for the same transaction. The width of the bus is a parameter to the simulator; in our simulations, we used 32 or 64 bit data busses.

Main memory is assumed to require 10 cycles to produce the first word, plus an additional cycle for every word (or doubleword, depending on the simulation parameters). The number of independent memory banks is another simulation parameter, but in most cases we used 4 memory banks.

2.2.1 Workings of the simulator

After the simulator completes its initialization, either the first CPU or all CPUs (depending on a command-line flag) begin executing code from the code file (Figure 2.1). Each CPU executes some number of instructions and increments its own clock until it executes a load or store instruction. Since the effect of a load or store is visible to the rest of the simulated system, a CPU may not proceed with such an instruction until the global simulation clock (which is the true system time) catches up to the CPU's clock time. If the load/store is to a non-shared location, or hits in the simulated cache (and the block is in the appropriate state), the CPU will complete the memory operation and continue executing until the next load/store instruction. However, if the memory instruction requires action by the memory system, a bus request will be formed and sent to the bus module. The bus module is responsible for satisfying all bus requests, and models delays due to arbitrating for the bus, checking other processor's caches, and the transmission of bus requests and replies. If the bus request cannot be satisfied by the other caches attached to the bus, the request is sent either to the memory module in a single-bus system, or to the cache on the next highest level in a hierarchical system. After a bus transaction successfully returns from the bus module, the cache block is updated, and the processor is restarted. If a request is unsuccessful, the request is returned to the originating cache, and the load/store is retried.

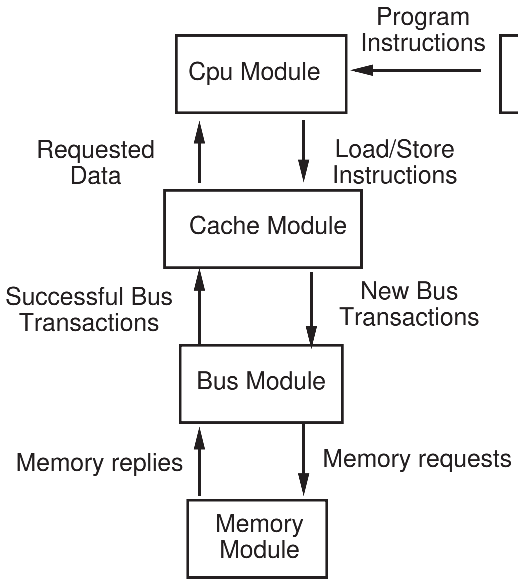


Figure 2.1: Simulator flow

2.2.2 Protocol correctness

Designing a cache coherence protocol that works correctly in all circumstances can be a difficult task. Not only must a protocol correctly enforce the appropriate memory consistency model, but also it must avoid deadlock and livelock conditions. One way of debugging and verifying a cache coherence protocol is to use an automatic protocol verifier, such as Murφ [DDHY92]. Unfortunately, time demands and the difficulty of correctly specifying a non-trivial protocol to a verifier precluded us from using such a tool. Moreover, the tool might not be capable enough to handle the more complex protocols developed in this dissertation. Instead, we used the simulator itself to find errors not only in the protocol, but also in the implementation of the design in the simulator code. The first technique we used took advantage of the fact that we used a program-driven simulator in our studies. The simulated CPUs used the data values returned by the simulated memory system to perform the program's computations. Therefore, protocol errors which returned the wrong data to the processor often showed up as incorrect program results. This was discovered when the results of the simulated program were compared against the results produced by running the program directly on a uniprocessor workstation.

Since protocol errors don't always manifest themselves in a program's final results, we implemented a finer-grain checker inside the simulator. The fine-grain checker allocates a shadow copy of main memory at simulator start-up. When a processor completes a store instruction to a shared location, the data value is sent not only to the simulated memory system, but also directly to the shadow memory. On every completed load, the check verifies that the data returned from the simulated memory system matches that kept by the shadow memory.

Lastly, we inserted code into the simulator to check for illegal conditions, such as cache lines that are exclusive in one cache but are also present in another cache, or lack of inclusion in upper level caches. When a check is violated, the simulation run is aborted. Though these measures don't guarantee correctness, they do give us a good degree of confidence that the protocols we developed work correctly.

2.3 Benchmarks

In our studies, we used 7 applications to test and evaluate our proposed design. We chose these benchmarks because together they exhibit a wide variety of reference patterns. In all cases, we gathered statistics only for the parallel section of the program.

The Gauss, Relax, and Cholesky benchmarks all have good spatial locality and exhibit high hit rates. MP3D exhibits a great deal of true sharing, and was written to avoid false sharing. Both the Topopt and Pverify applications exhibit both true and false sharing, even using moderate (32 byte) block sizes. Lastly, the Barnes application does relatively few shared accesses. In the following paragraphs, we briefly describe each application.

Gauss

The Gaussian elimination program solves without pivoting the system of linear equations $A \times x = b$. This computation lends itself well to parallelism [Dar88]. Each processor is assigned (statically) a set of rows, and triangularization proceeds column by column. Computation for a new column cannot proceed until all calculations for the previous column have been completed. The amount of work performed by each processor varies slightly depending on the dimension of the A matrix. The second phase of the procedure is a classical back substitution and takes much less time than the triangularization. Back substitution takes place in parallel as each element in the solution vector is computed and substituted into the equations for the other solution elements.

The main data structures in Gauss are the A matrix and the b vector. Both shared structures are allocated dynamically using the general purpose memory allocator. The A matrix is constructed as a one dimensional array of pointers, each pointing to a single dimension array of double precision floating point numbers (8 bytes). Because the memory allocator rounds up allocation requests to the next higher power of two, successive rows of the matrix will likely not be laid out contiguously to one another in memory. Hence it is unlikely that elements from two adjacent rows of the A matrix will end up in the same cache block. This is important, because rows are statically assigned in a round-robin fashion to the processors. As the triangularization process proceeds, each row is written some number of times by the same processor, without any intervening reads or writes by another processor. Once work on a row is finished, it is read one time by all the processors. Subsequently, the row is not written by any processor. Because of this data access pattern, there is very little true or false sharing involving the elements of the A matrix.

The b vector is contiguously allocated. Because each successive element of the b vector is assigned to a different processor, false sharing will occur when the line size is greater than the size of a b element (8 bytes). However, since b vector elements are accessed a relatively small amount of time, the effect of the false sharing is minimal.

Overall, Gauss exhibits a large degree of spatial locality and a low degree of true and false sharing (for “reasonable” choices of block size).

A Gaussian elimination problem of NxN will require storage of approximately 8 * (N * (N + 1)) bytes. For our tests we used a 250 by 250 matrix, which works out to be about 490K. The approximate cache size where there won't be many capacity misses, for a given number of processors, is:

Table 2.1: Working set size for Gauss

# of processors	size per processor
1	490K
2	245K
4	122K
8	61K
16	31K
24	20K
32	15K

Due to data layout, the Gauss application suffers from conflict misses. These conflict misses are present even when 32 128K 4-way set associative caches are used. On a single processor, Gauss running with a 250 by 250 matrix requires about 161 million instructions to complete.

Relax

The Relax program computes the successive over-relaxation over a matrix of numbers. For each iteration, the new value of a given matrix element is computed based on the value of the element itself, plus its eight nearest neighbors. We ran the application for a predetermined number of iterations, though in practice, the program would be run until the matrix values converge. In the parallel version of the program, each processor is statically assigned to compute a portion of the matrix.

Relax's matrix is a NxN array of double precision floating point numbers. A single processor allocates and initializes the original array. Each processor then allocates a smaller array of single precision floats (4 bytes) that holds its intermediate results.

The program then begins the first iteration. Each iteration proceeds in phases: in the first phase, each processor reads the appropriate values from the main array and calculates its results into its local submatrix. In the second phase, each processor writes its results from its submatrix back to the main matrix. Barriers are used to synchronize between phases and iterations.

Overall, Relax exhibits good locality. Since a processor computes from the same sub-matrix iteration after iteration, data reuse is common, assuming that the processor cache is large enough to avoid replacements. In addition, Relax's pattern of accessing an element's nearest neighbors also tends to produce high cache hit rates. However, because matrix elements on the boundary between two sub-matrices are accessed by two (or more) processors, Relax does exhibit some true sharing. In addition, the application is susceptible to a small degree of false sharing, since a large cache block may contain elements from two different sub-matrices.

As stated above, data reuse in Relax will be high, as long as cache capacity is sufficient for the data set. For each element a processor is responsible for, 12 bytes of cache storage are needed (8 for the main matrix, 4 for the local sub-copy). In addition, space is needed for boundary elements. For a 256x256 array, which is what we used in our simulations, Figure 2.2 shows roughly how big a cache would have to be to avoid most capacity misses.

Table 2.2: Working set size for Relax

# of processors	size
1	776K
2	388K
4	194K
8	97K
16	50K
24	37K
32	25K

Despite these figures, Relax still suffers from a non-trivial number of capacity misses even when a megabyte cache is used with a single processor. Unlike Gauss however, Relax does not suffer from a significant amount of conflict misses when caches with associativity of two are used.

A uniprocessor run of Relax on a 256x256 matrix requires over 62 million instructions.

Cholesky

Cholesky is a benchmark from the SPLASH suite used for sparse matrix factorization. The principal data structure is the representation of the sparse matrix, which is stored by columns. Columns of the matrix are grouped into supernodes to obtain adequate granularity. Associated with each supernode is a count of how many modifications are to be done to the columns of that supernode. When the count drops to zero, that supernode is placed on the global task list. Each free processor takes a supernode off the task list and modifies the appropriate parts of the main matrix. Processors interfere with one another when attempting to take work off the queue, and when making modifications to the same destination column. The non-zero elements of the matrix are stored contiguously by column in a single-dimensional array. Because Cholesky accesses the matrix in supernode chunks, it exhibits good spatial locality. Because of time constraints, we used a relatively small input matrix, BCSSTK14, which limited speedups due to the lack of available concurrency. Processing the BCSSTK14 matrix required 65 million instructions on a single processor. Cholesky has a fairly large working set — the factored matrix requires 1.4 megabytes of storage. Because of the large size of the matrix, conflict misses were much less important in our simulation runs than capacity misses.

Topopt

Topopt performs topological optimization on VLSI circuits using a parallel simulated annealing algorithm [DN87]. The circuit is represented in the form of an array, which the algorithm repeatedly folds to minimize the circuit's area. Because the assignment of work to processors changes dynamically, and gates and signals are shared between processors, Topopt has a great deal of true sharing [Egg89]. In addition, Topopt also exhibits false sharing when larger line sizes are used [EJ91]. A notable feature of Topopt is that it has 4 barriers in its inner working loop. For all but the master processor, the amount of work performed between successive barriers is the same.

This leads to contention for the lock guarding the barrier entry code, and (depending on the implementation) the variables used to implement the barrier.

In our experiments, we used the cpla1.lomim input file. A single processor executes about 1.7 billion instructions in the (measured) parallel section of the program. In uniprocessor runs, Topopt's working set size is about 128K. However, using a cache with associativity of 2 will incur conflict misses unless it is at least 256K in size.

Pverify

Pverify is a verification program which compares two circuits and reports if they are functionally equivalent [Egg89] [MDWS87]. The program consists of two phases. In the first phase, the program determines which input combinations produce an output of one (or zero) for the first circuit, using a depth first search of the circuit's graph representation. In the second phase, the second circuit is simulated using the input combinations used in the first phase. The outputs of the second circuit are verified against the outputs of the first circuit [Jer95]. In the both phases of the algorithm, work is assigned dynamically to threads. Pverify exhibits a great deal of false sharing, event when moderate-sized blocks are used [EJ91]. We ran Pverify on the circuits C880.21.berk1 and C880.21.berk2 which used about 1 billion instructions on a single processor.

Using the above input files, Pverify's uniprocessor working set size is about 256K. However, cache conflict misses still occur in a 256K cache when low associativities (like 2) are used, though they disappear when a 2-way 512K cache is used.

MP3D

MP3D is a 3-dimensional molecule simulator from the SPLASH suite [SWG92]. The program evaluates the positions and velocities of molecules over a sequence of time steps. The two principal data structures in MP3D are two arrays, one for the molecules, and another which stores the properties of the space cell. Work is partitioned by molecules, which are statically assigned to processors in groups to avoid false-sharing problems. Each time step, processors move the bodies assigned to them, then check for collisions with other particles and the boundaries. Although it appears that the program is well suited for parallelization, speed-ups are hampered by changes in localities when molecules move from one "space" cached in one processor to another. Much of the shared data in MP3D migrates from one processor to another as the program proceeds [GW92]. Thus the number of invalidates does not substantially increase as the number of processors increases. For our benchmark runs, we simulated 20,000 particles for 100 time steps, which requires 497 million instructions by a single processor. The uniprocessor working set requires between 512K and 1024K of cache. Most of the static data space is occupied by large arrays of particles and data spaces. Each particle requires 36 bytes per cell, while each space cell requires 40 bytes.

Barnes

Barnes is another benchmark from the SPLASH suite. It simulates the effect of gravity on a system of bodies. The main data structure is the Barnes-Hut tree, which represents the hierarchical division of the bodies in space. As the algorithm proceeds, the amount of work done on behalf of each particle is profiled, and used as an estimate of how much work will be required for that particle for the next time step. The tree's interior nodes record the amount of work required for its children. The tree is then disjointly partitioned and assigned to processors for the computation phase. The partitioning algorithm strives to maintain cost balance and tree contiguity, which in most cases preserves data locality. Because of the good partitioning algorithm, Barnes exhibits a very low coherence miss rate, even when 64 processors are used. Consequently, the application exhibits near linear speedup on most shared-memory architectures. In order to keep simulation time down, we ran Barnes with only 128 bodies, which requires 111 million instructions to execute on a single processor. The size of the working set is very small — less than 128K bytes.

Part I

Single Bus Improvements

Introduction

Because of their simplicity and low cost, single bus shared-memory multiprocessors have enjoyed a great deal of commercial success. Machines like the Sequent Symmetry [LT88], SGI Challenge [GW94], and Sun's SPARCcenter 2000 [CYS+93] all use commodity processors and bus technology to reduce design complexity while still achieving good performance. Such designs leverage the investment made in uniprocessors to keep costs down.

Like uniprocessors, single-bus multiprocessors generally use per-processor (or private) caches in order to reduce memory latency and bus traffic. Such caches are very effective in increasing system performance. Unfortunately, private caches introduce a problem when data that is present in multiple caches is modified by a processor. The problem of providing all processors a consistent view of memory is called the *cache consistency* problem. There are many solutions to this problem. The simplest solution is to disallow caching of all write-shared data in a program [WB72]. A disadvantage of this solution is that the compiler must conservatively determine which blocks might be write shared. A bigger drawback is that the additional bus traffic generated by accesses to non-cacheable data substantially reduces system performance. If shared-writable data is allowed in private caches, some mechanism must exist to provide processors a consistent view of memory. In general, the mechanism can be provided in software, hardware, or some combination of both.

One hardware strategy is to maintain coherence information with every memory block in structures called directories [Lil93]. When a cache miss occurs, the directory is consulted and the appropriate action taken to maintain coherence. Directory-based coherence protocols don't rely on inexpensive broadcast mechanisms, so they are often a good protocol choice for non-bus based systems.

For single bus machines, however, a cheaper hardware mechanism called *snoopy caching* is generally used. In a snoopy cache coherence protocol, caches observe (snoop on) bus transactions and take action to keep the cache consistent. States are associated with each cache block. The two major types of snoopy protocols are *write-invalidate* and *write-update*. Under a write-invalidate scheme, a processor performing a write to a block which is currently shared must first erase, or invalidate all other copies of the block. Under write-update, the writing processor broadcasts the new data value, which other processors that currently have the block valid in their cache then read to keep their cache consistent. Because a write-update policy updates, rather than invalidates cache data, a multiprocessor employing write-update usually has lower miss rates than one using write-invalidate, but often at a cost of increased bus traffic.

Several examples of both types of protocol exist. A good example of a write-update (WU) protocol is the Dragon protocol [McC84]. A feature of this protocol is the addition of a *sharing line*. When a processor performs an update transaction on the bus, other caches signal that they read the data by raising the sharing line. If no other processor currently shares the data, the writing processor will observe that the sharing line is not asserted and will cease sending updates until the block becomes shared again. This optimization conserves bus bandwidth and decreases write latency.

Two examples of invalidate protocols are the Illinois protocol [PP84] and the Berkeley protocol [KEW$^+$85]. Both protocols feature cache-to-cache transfers on read misses and three common cache block states (Invalid, Clean-Shared, and Dirty). The Illinois protocol differs from the Berkeley protocol in that it assumes that a cache can determine the source (memory or another cache) of a block that has been read. This information is used to determine if the VALID EXCLUSIVE state can be used, which allows processors to write to clean blocks without a bus transaction. The Berkeley protocol uses its fourth state to encode a DIRTY SHARED state, which avoids writing modified cache blocks back to memory unless they are replaced. Because the Illinois protocol is the more common of the two protocols, we will be using it as our standard invalidate protocol.

Choosing the best policy between update and invalidate protocol is difficult, because the best protocol depends not only on the application, but also on the machine's hardware configuration. Chapter 4 addresses the issue of choosing the best protocol.

Among the two policies, write invalidate has been the favorite among both researchers and hardware designers. One design choice in an invalidate-based system that can drastically affect performance is the size of the cache block. In most systems, the block size determines how much data is transferred on a read miss, and also how much data is invalidated when a processor does a write. A problem with using a single block size is that some applications perform best with a large block size, while other do better using a small block size. Chapter 3 examines the problem of using the best block size for each application. It also evaluates a technique called *read snarfing*, which reduces the number of times a block is re-read following an invalidation.

Chapter 3

READ BROADCAST AND THE SUBBLOCK PROTOCOL

3.1 Introduction

In this chapter we investigate two techniques to reduce memory traffic in a single bus, cache coherent shared-memory multiprocessor architecture. For the first technique, we focus on the relationship between the coherence protocol and the cache block size. While some applications run better when small cache block sizes are used because of the presence of migratory data or because false sharing is avoided, others execute more quickly with larger block sizes because they exhibit good spatial locality. Most often, an architecture is implemented with the same block size used for both transfer (memory to cache or cache to cache) and coherence. Our technique is a snoopy cache coherence protocol (called the subblock protocol) that uses different block sizes for transfer and coherence. The goal of the subblock protocol is to obtain the advantages of using large block sizes for those programs that have good spatial locality while avoiding many unnecessary invalidations that result from migratory data and false sharing. The subblock protocol performance is compared to that of an invalidate protocol, namely the Illinois protocol.

The other technique we evaluate is read snarfing. Read snarfing is an enhancement to snoopy cache coherence protocols that takes advantage of the inherent broadcast nature of busses. In a system with read snarfing, caches snoop the bus not only for coherence transactions, but also for potentially useful data that is being read by another processor. In this chapter, we evaluate the impact that snarfing has on the performance of the Illinois protocol and on our newly developed subblock protocol.

In Section 3.2 we present our model architecture, in particular a sector cache organization that allows subblocks. Sections 3.3 and 3.4 describe the two techniques, subblock protocol and read snarfing, in more detail. Section 3.5 presents our evaluation methodology for this particular study, with Section 3.6 giving simulation results. Suggestions for improvements and further study are given in the concluding section.

3.2 Architectural model

Our base architecture is a single, shared bus, cache coherent multiprocessor. We consider two types of caches: "usual" caches and "sector" caches. Usual caches have a capacity C, a block size L, and an associativity k. Corresponding sector caches have the same (C, L, k) characteristics but in addition the blocks are divided into subblocks of size b. The units of coherence and transfer are the same for the usual caches, namely L, while these units can be L or b depending on specific protocols in the case of sector caches. In the sector caches, both blocks and subblocks have states. A sector cache (C, L, k, b) will therefore require $2L/b$ extra state bits/block, assuming that subblocks have a maximum of 4 states. Thus an $(C, L/2, k)$ usual cache and a (C, L, k, b) sector cache require approximately the same number of tag and state bits for L between 32 and 128 bytes and $b = 8$ bytes. Note also that a usual cache with $L = b$ requires more memory than a sector cache with L and b as above [Sez94].

Systems with sector caches also require more bus lines to transmit bitmasks corresponding to the status of the subblocks in a particular block. However, the number of additional lines will be comparatively small, since the number of extra lines required is equal to L/b, the number of subblocks in a block.

As stated in Section 2.2, we assume caches with dual ported tags. This is particularly important when snarfing is implemented so that read snarfing snoop checks don't interfere with a processor's access to the cache.

3.3 The subblock protocol

3.3.1 Subblock protocol

The motivation behind sector caches is to design a coherence protocol that takes advantage of applications' spatial locality by fetching large sized blocks while avoiding invalidations and migrations of falsely shared data by having a smaller coherence unit, namely a subblock. The protocol is snoopy-based [AB86]. It incorporates features from the Illinois protocol [PP84] and from protocols or write policies with subblock (in)validations [CD93, Jou93].

The basic philosophy behind the protocol is as follows. As much as possible, we favor cache to cache transfers. On read misses, we transfer as many valid subblocks in the block as possible. If the needed subblock is in another processor's cache, that cache sends along not only the cached subblock, but all other read-shared subblocks (i.e., subblocks in states *Clean Shared* and *Dirty Shared*, see below) in that block. In contrast with the Illinois protocol, dirty subblocks that are transferred on read misses are not copied to memory (hence the *Dirty Shared* state for ownership). On the other hand, if the subblock is located only in main memory, then memory will respond with the requested subblock, plus any subblocks which are not currently cached. The information about which subblocks are currently cached is determined when the request goes out over the bus. On writes to clean subblocks and write misses, we invalidate only the subblock to be written. We do this in order to avoid the performance penalties associated with false sharing.

To implement our protocol, we used 4 block states, plus 4 subblock states. The block states are useful when there is only one cache that contains a copy of the block (state **VALID EXCLUSIVE**) thus avoiding invalidations, or when all subblocks in the block are either clean – and not owned – or invalid (state **CLEAN SHARED**), thus avoiding write-back upon replacement. The block states are as follows:

INVALID All subblocks are *Invalid.*

VALID EXCLUSIVE All valid subblocks in this block are not present in any other cache. All subblocks that are *Clean Shared* may be written without a bus transaction. Note that any (or possibly all) subblocks in the block may be *Invalid.* There also may be *Dirty* blocks which must be written back upon replacement.

CLEAN SHARED The block contains subblocks that are either *Clean Shared* or *Invalid.* The block can be replaced without a bus transaction.

DIRTY SHARED The subblocks in this block may be in any state. There may be *Dirty* blocks which must be written back on replacement.

The subblock states are as follows:

Invalid The subblock is not valid.

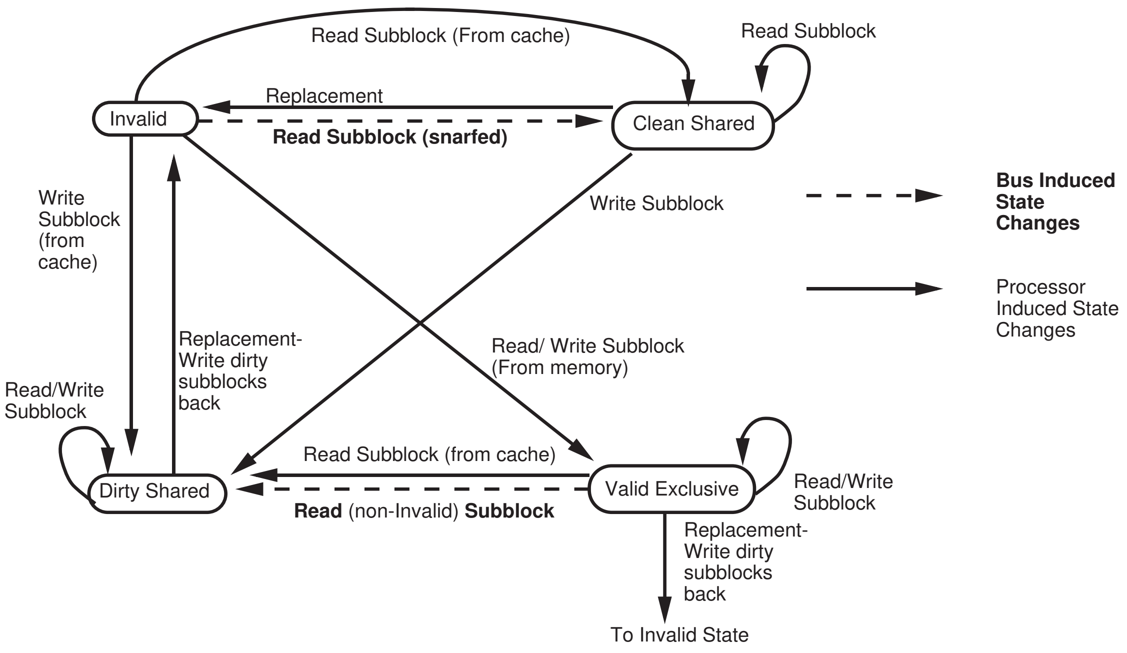


Figure 3.1: Block Protocol

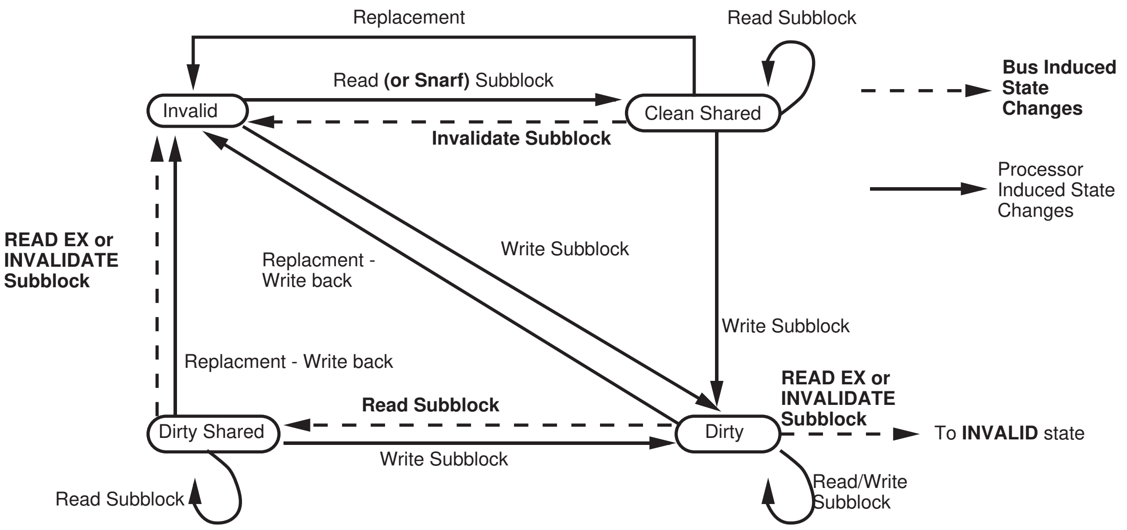


Figure 3.2: Subblock Protocol

Table 3.1: Allowable states for subblocks

Block Status	Subblock Status			
	Invalid	*Clean Shared*	*Dirty Shared*	*Dirty*
INVALID	OK	Error	Error	Error
VALID EXCLUSIVE	OK	OK	Error	OK
CLEAN SHARED	OK	OK	Error	Error
DIRTY SHARED	OK	OK	OK	OK

Clean Shared A read access to the subblock will succeed. Unless the block that the subblock is a part of is in the **VALID EXCLUSIVE** state, a write to the subblock will force an invalidation transaction on the bus.

Dirty Shared The subblock is treated like a *Clean Shared* subblock, except that it must be written back on replacement. At most one cache will have a given subblock in either the *Dirty Shared* or *Dirty* state.

Dirty The subblock is exclusive to this cache. It must be written back on replacement. Read and write accesses to this subblock hit with no bus transactions.

Table 3.1 shows the states a subblock may be in given a particular block status.

Figure 3.1 gives the state transitions for cache blocks, including transitions induced by read snarfing (see Section 3.4). Note that there is only one transition (from **VALID EXCLUSIVE** to **DIRTY SHARED**) that is induced by requests from other processors that the cache observes. All other bus induced state changes change subblock states only. Figure 3.2 shows the state changes at the subblock level. For a complete description of the protocol, see Appendix A.

To illustrate the protocol, we give an example of its workings on a particular cache block. In Figure 3.3 we show the status of a cache block, say L, which contains 4 subblocks. The subblocks will be referred to as a through $a+3$. Initially, the block is invalid in all three caches of the processors P1, P2 and P3.

At step one, P1 reads subblock $a+2$. Since no other cache has the block, the request goes to memory. Memory returns all subblocks in the block. The state of the block is **VALID EXCLUSIVE**, which signifies that all valid blocks in L exist only in this cache. The four subblocks are loaded in the *Clean Shared* state.

At step two, P2 writes subblock $a+1$. A combination read/invalidate message is sent over the bus, which invalidates only subblock $a+1$ in P1's cache. Note that P1 remains **VALID EXCLUSIVE** because all of its remaining subblocks are still unique to it.

On the third step, P3 reads subblock a. P1 supplies subblock a as well as all other valid, non-dirty subblocks it possesses. The state of the block in P1's cache is changed to **DIRTY SHARED** because other copies of L's subblocks exist in the system. Note that P2's cache snarfs the 3 valid blocks.

The next operation is a write to subblock $a+3$ by P1. This operation invalidates the subblock in P2's and P3's caches.

Following this operation, let us assume that a request from P1 induces a replacement of L in P1's cache. A bitmask is transmitted with the write back request so that only subblock $a+3$ is updated in memory.

In the final operation, P3 reads subblock $a+3$. When the request is broadcast over the bus, each cache responds with a bitmask indicating which subblocks of L (if any) it has stored in a valid state. These bit-masks are OR'ed together to give a complete view of which subblocks of this particular block are cached – and valid – in the system. Because no other cache has the subblock $a+3$, the request must go to memory. The memory unit returns the entire block, but only those subblocks which are not cached elsewhere in the system are marked valid in the requester's cache (in addition to those subblocks, like a and $a+2$ that were already valid). This is to prevent stale data from memory from being marked valid in a cache. In our example, the $a+1$ block is not marked valid in P3's cache, because it is cached and dirty in P2.

3.3.2 Related work

Goodman introduced the concept of a *coherence* block which can be different in size from either an address block (the amount of storage associated with a cache address tag) or a transfer block, which is the amount of data transferred from memory on a miss [Goo87]. Goodman advocates using a large size for the coherence block to reduce the number of coherence operations that must be done to read or write a

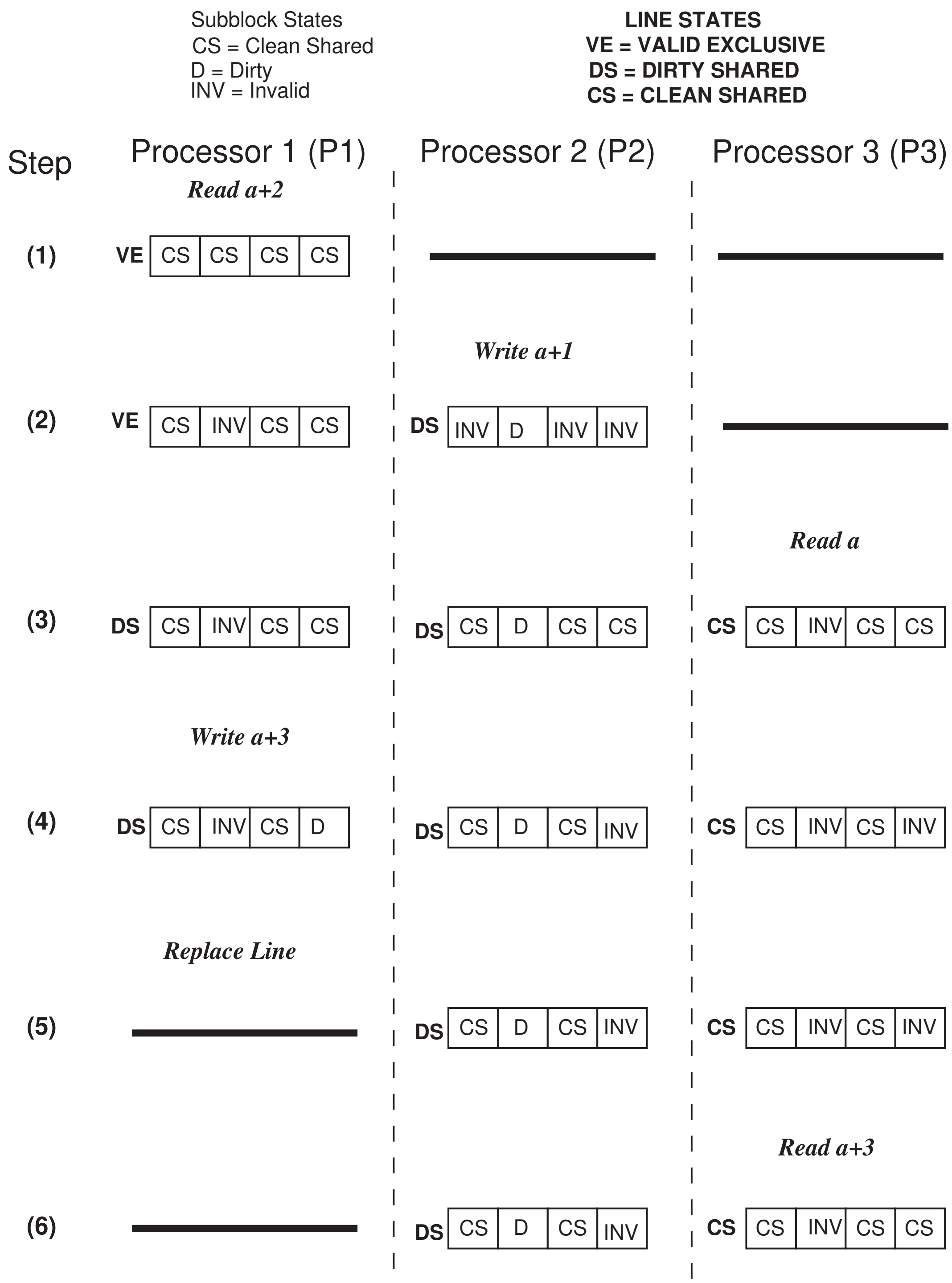


Figure 3.3: Subblock Protocol Example

given amount of data. However, at some point increased traffic from false sharing misses will cause an overall increase in memory latency and bus traffic.

Other authors have proposed *directory* based schemes in which the unit of coherence is smaller than an address block. Chen and Dubois [CD93] describe an extension to a full-map directory protocol in which a valid bit is associated with each invalidation block. They show a substantial decrease in both miss rates and memory traffic when partial invalidations are used. Dubnicki and LeBlanc [DL92] propose adjustable block size caches where the size of an address block dynamically grows or shrinks, in a buddy-system like fashion, in response to various patterns of write sharing. Unlike Chen's protocol, Dubnicki's protocol is not easily adaptable to non-directory based coherence protocols. Tomašević and Milutinović [TM91] advocate a Word Invalidate Protocol (WIP) that uses per-word valid bits and additional block states to reduce the "pollution" of valid blocks with invalid words. Using a synthetic benchmark to evaluate their protocol, they found that the WIP protocol overall performed slightly better than the write-update Dragon protocol for their parameters.

3.4 Snarfing

Read snarfing is an enhancement to snoopy-cache coherence protocols that takes advantage of the inherent broadcast nature of the bus. Under read snarfing, the cache snoop mechanism of every cache, and not only of the requesting cache, checks the address associated with all completed read operations on the bus. If the address of the data matches in the cache, but the state of the block (subblock) is invalid, the cache will read the value of the block from the bus, and set the state of the block (subblock) to clean shared. The main benefit of read snarfing is that it reduces the number of read misses for blocks that were invalidated because of coherence actions. The first processor to reload a block because of an invalidation miss will effectively supply the block to other processors whose blocks were invalidated in the past, assuming that the invalidated blocks were not overwritten by other data.

We expect that read snarfing will enhance the overall performance of the system (reduction in elapsed execution time and in bus utilization). Note however that snarfing can produce an "anomaly", namely an increase in the miss rate (this actually happened in two of the benchmarks discussed later). To see how this can happen, consider a system that uses test and test and set (TTS) locks. The code for TTS locks is as follows:

L1: (1) Load L (L is the lock location)
(2) If L != 0, goto L1
(3) Load 1 into a register
(4) Atomically swap L with register
(5) If register != 0, goto L1
(Lock succeeds)

To unlock the lock, the holder simply swaps a 0 into the lock location.

Each time a lock is released and acquired, the value of the lock transitions from 1 to 0, then back to 1. A processor unsuccessfully contending for the lock would see 2 misses — one for each transition. One miss would be a read miss at statement (1) which is caused by the transition from 1 to 0, and the other would be a swap (write) miss at statement (4) since the processor successful in getting the lock would have invalidated all other copies. This is the case when bus and lock contention are low.

However, when bus contention is high, it might take a long time for the load that results from the first miss to be completed. The load could be delayed so long that the value loaded is 1, not 0. The processor would only tally one miss (a read miss), not two, as a result of the lock being released and acquired.

When snarfing is used, the processor's read request is satisfied quickly, since one of the contending reads succeeds even under high bus loading. Because of this, the processor will tally more misses under snarfing. It tallies more write misses, since all processors contending for the lock fall through to statement (5), unlike the case when there is no snarfing and high bus contention. It also tallies more read misses, because under snarfing, all processors contending for the lock reach statement (4) at about the same time. The first contending processor to get the bus will succeed in taking the lock; all others will fail. The first processor to fail (call it A) will branch back to L1, where it spins. The second processor to fail will then execute statement (4), invalidating the copy in A's cache, which causes a read miss. Depending on the interleaving of requests on the bus, processor A may experience one or more read misses due to failed attempts to take the lock. This undesirable behavior could be avoided by using a queue lock which works well under both heavy and light contention.

Despite the additional misses caused by snarfing, using it can reduce overall execution time and bus loading, because most of the extra misses incurred do not require a bus access to satisfy them; the value of the lock is reloaded through snarfing.

Read snarfing was discussed in previous work by Rudolph [SR84], Goodman [GW88], and Eggers [EK89]. Of these, only Eggers evaluated the performance benefits of read snarfing. Under a single-ported tag directory assumption she found that although read snarfing resulted in a decrease in invalidation misses and bus utilization for most applications, total execution time increased, due to processor lock-out from the cache and the need for caches to acknowledge that they have completed a snoop. With our assumption of dual-ported tag directories the processor lock-out problem is no longer present.

3.5 Methodology

We modified the Cerberus simulator to evaluate both the sector cache protocol and snarfing. The basis for comparison (with usual caches) is the Illinois protocol. The width of the bus in all simulations was 64 bits. The other simulation parameters are as given in Section 2.2.

Our experiments simulated both usual and sector caches. In all experiments the subblock size b was 8 bytes and caches were two-way set-associative ($k = 2$). In order to minimize simulation time, we only simulated relatively large caches ($C = 128K$). Previous work [AB94] indicates that using a smaller cache size does not significantly change the results. The block size was either 8 or 64 bytes. For the usual caches we used the Illinois protocol. For the sector caches we simulated the protocol described previously (of course there was no simulation of sector caches with a block size of 8 bytes).

Finally, we simulated the applications on systems with 1, 4, 16, and 32 processors.

3.5.1 Benchmarks

We chose six applications to evaluate our protocol. As stated in Section 2.3, the applications' reference behaviors fall into four categories:

- An application that have few shared accesses, so the choice of block size is less important (Barnes)

- Applications that have good spatial locality and exhibit high cache hit rates, and therefore do well with large block sizes (Gauss, Cholesky)

- An application which exhibits mostly true sharing, and performs somewhat better with large block sizes (MP3D)

- Applications that exhibit both true and false sharing, that perform better with small block sizes (Pverify, Topopt)

3.5.2 Metrics

The principal metric we used to evaluate the protocols was execution time in simulated clock ticks. For all programs, all measurements were done only on the parallel section of the application; for the programs studied, the execution time of the serial section of the code is negligible compared to the time spent in the parallel section.

We also gathered other performance metrics that allowed us to interpret the simulation results. The two most important ones are the number of (shared) data bytes transferred during execution, and the number of bus transactions needed. These metrics are important not only because they allow us to understand why one protocol performs better than others, but also because they can be used to project how technology advances will affect our results. For example, we could project the impact of faster processors or of wider busses by expressing the total elapsed time as the sum of the "computational cycles" and of the "bus cycles", with differing values for these "cycles" and a factor for varying bus contention.

3.6 Results

We report our results in four groups, corresponding to the classification used in section 3.5.1. First, we present results for the three applications (Gauss, Cholesky, and MP3D) that do well using large block sizes. Next, we present data for the two applications (Topopt, and Pverify) that perform better using smaller block sizes. Finally, we report on our results running Barnes, for which the choice of block size is not very important. Note that in the following discussion, "8I protocol" refers to the Illinois protocol used with an 8 byte block size; similarly, "64I protocol" refers to the Illinois protocol and a 64 byte block size.

Barnes

Overall, the Barnes-Hut application exhibits good locality, primarily because it does relatively few shared read (10-15%) and shared write references (.1%). Because of this, varying the block size, or using the subblock protocol or snarfing had little effect on overall speedup (Figure 3.4(L)). However, if we look at the number of bytes transferred (Figure 3.4(R)), and the number of bus transactions (Figure 3.5(L)), then we do see differences between the tested configurations. Since the working set for the application fits in a single 128K cache, most of the bus traffic is due to invalidations and the subsequent misses due to invalidations. As we can see in Figure 3.4(R), using the 64I protocol resulted in over 5 times more bytes transferred than the 8I protocol. While using the 64I protocol did reduce the number of bus transactions by about 25%, the large number of bytes transferred under the 64I protocol resulted in much higher bus utilization than the 8I protocol.

The subblock protocol performed well on the Barnes application. It substantially reduced the number of bytes transferred over the 64I protocol, approaching the performance of the 8I protocol. It used fewer bus transactions than either the 64I or 8I protocols. And its bus utilization was lower than either of the two Illinois protocols.

Snarfing produced large benefits for all three protocols studied. Using snarfing with the 64I protocol reduced the number of bytes transferred to nearly the level of the 8I protocol which did not use snarfing. Adding snarfing to the 8I protocol reduced the number of bytes transferred by about 75% for the 16 and 32 processor systems. Snarfing also reduced the number of bus transactions. For 32 processors, snarfing reduced the number of transactions by about 70% for both the 8I and 64I protocols, and by 57% for the subblock protocol. Snarfing dramatically reduced bus utilization for the 8I (from 17% to 4% for 32 processors) as well as that of the 64I protocol (from 31% to 7%) and the subblock protocol (14% to 4%).

Gauss

The Gaussian elimination application has little write-shared data, and good spatial locality. Therefore, using a large block size provides important performance benefits (see Figure 3.6(L)). Running Gauss on the 64I protocol results in linear speedup up to 16 processors, and a speedup of about 21 on 32 processors. In contrast, the 8I protocol peaks at a speedup of about 10 using 16 processors. The reason for this is

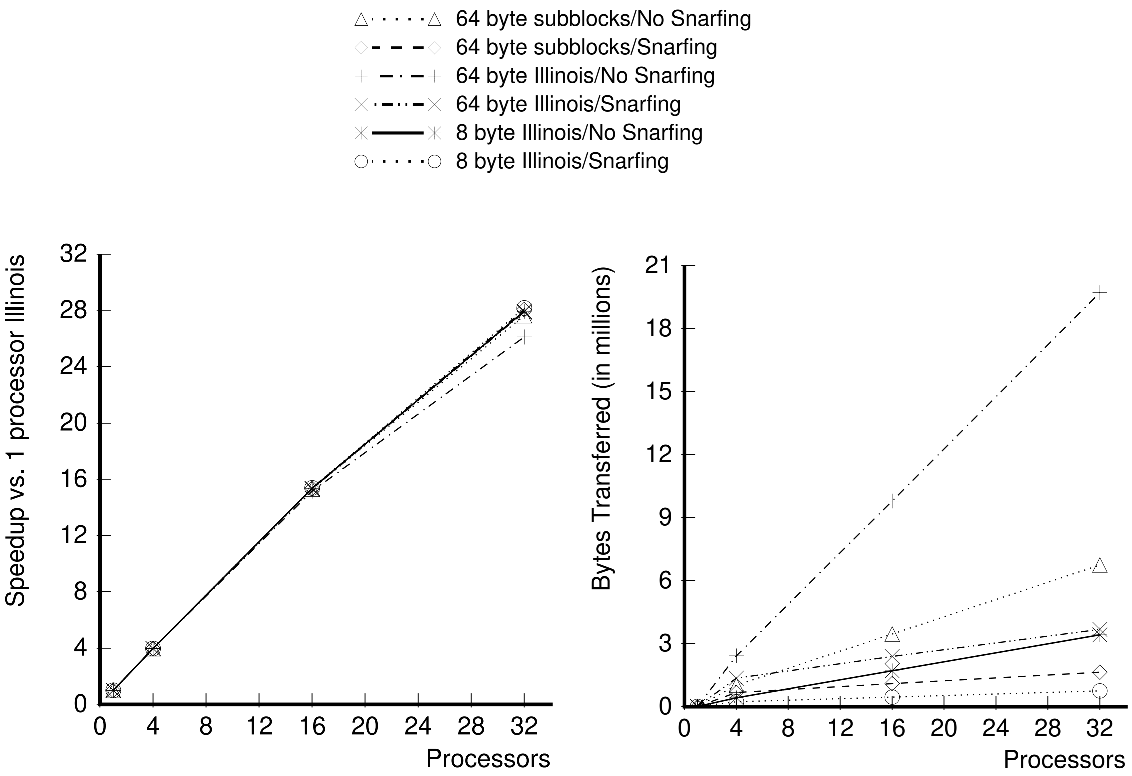


Figure 3.4: Speedup (L) and Bytes Transferred (R) for Barnes (128K Caches)

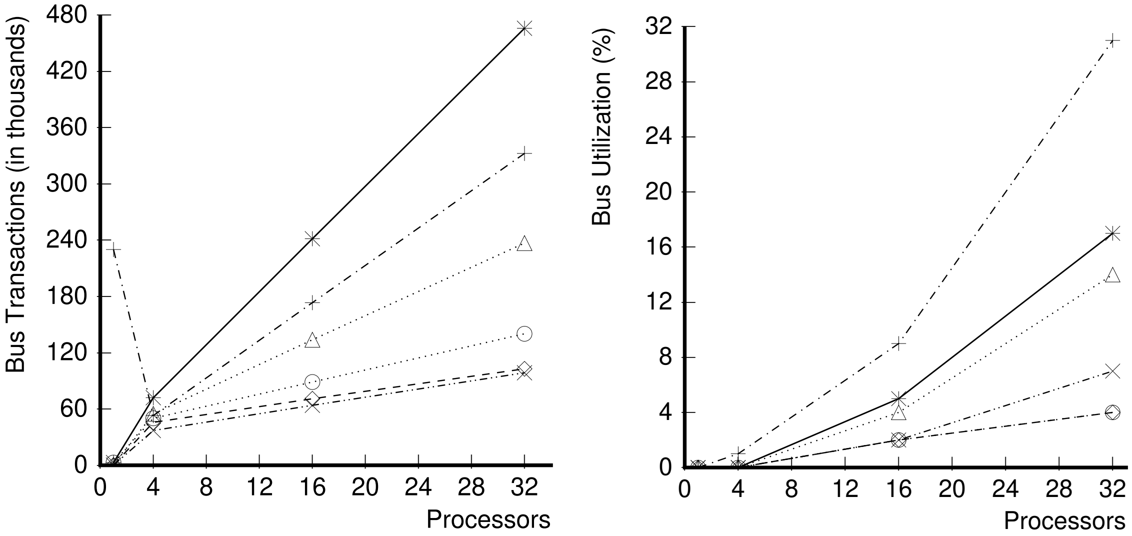


Figure 3.5: Bus Transactions (L) and Bus Utilization (R) for Barnes (128K Caches)

that the 8I protocol generates over 7 times as many bus requests as the 64I protocol, both for small and large numbers of processors (Figure 3.7(L)). This more than makes up for the small reduction (from 7% to 13%) in the number of bytes transferred when using the 8I protocol (Figure 3.6(R)).

The 64 byte subblock protocol outperformed even the 64I protocol on the Gaussian application. Because the subblock protocol transfers as much of a cache block as possible when satisfying read requests, it took advantage of the spatial locality in Gauss, which reduced the number of bus transactions needed. At the same time, the subblock protocol reduced the number of bytes transferred when compared to the 64I protocol, to a level in between that of the 64I and 8I protocol.

Snarfing had a small but beneficial impact on program performance, mostly for the 32 processor configuration. Snarfing reduced the number of bytes transferred for the 64I configuration by 5%; for other configurations, the figure is smaller. This is not surprising, given the limited number of opportunities available for snarfing.

Cholesky

While linear speedup was not reached in Cholesky, this application actually exhibits good spatial and processor locality; the lack of speedup is due to the small example that was simulated (Figure 3.8(L)). Because of its good locality, there was little difference in overall speedup between the various protocols simulated. The 64I protocol did run slightly faster than the 8I protocol, primarily due to the fewer bus transactions needed by the 64I protocol (Figure 3.9(L)). The 8I protocol required about 5 times more bus transactions than the 64I protocol to complete the program, whether for 1 processor or 32 processors. These additional bus transactions caused bus utilization to roughly double for the 8I protocol, despite the fact that using a smaller block size reduced the number of bytes transferred by at least 33% when using 4 or more processors (Figure 3.8(R)).

The subblock protocol performance was closer to that of the 8I protocol than to that of the 64I protocol. The reason for this is the large number of bus transactions required by the subblock protocol: 20% less than the 8I protocol but up to 4 times more than the 64I protocol. The subblock protocol required many more invalidations than the 64I protocol, since invalidations were done on the (smaller) subblock level. It also required more read requests than the 64I protocol. These additional requests

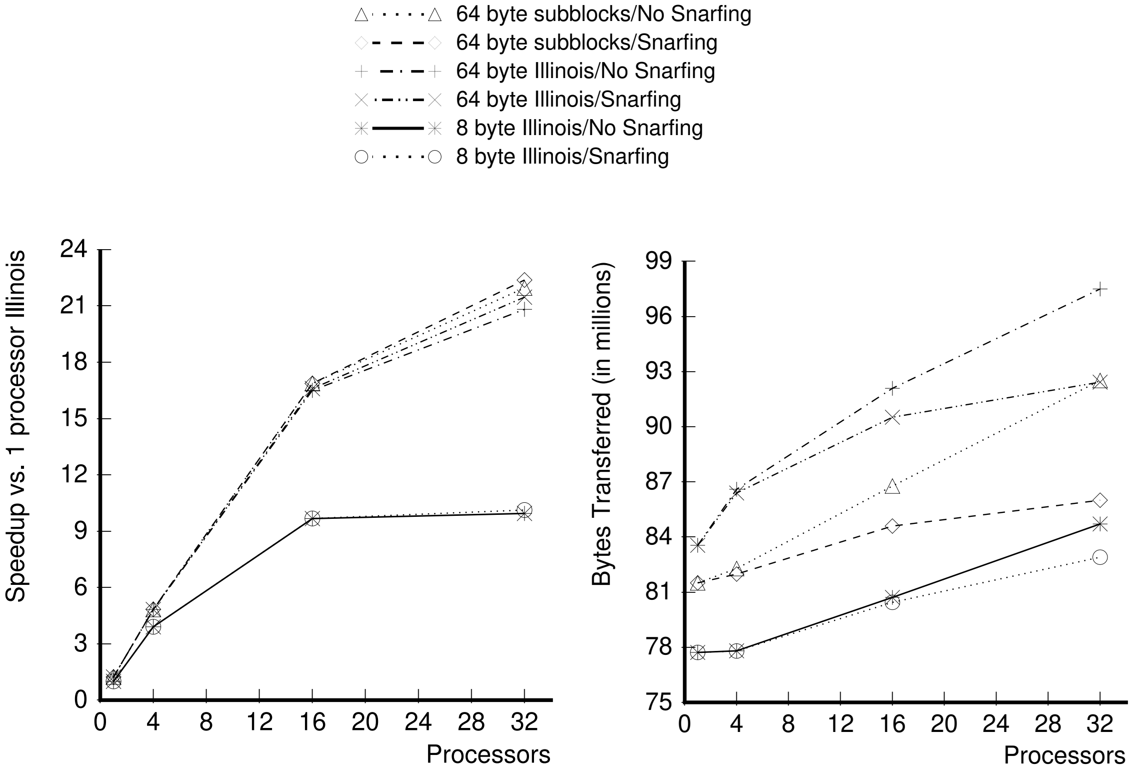


Figure 3.6: Speedup (L) and Bytes Transferred (R) for Gauss (128K Caches)

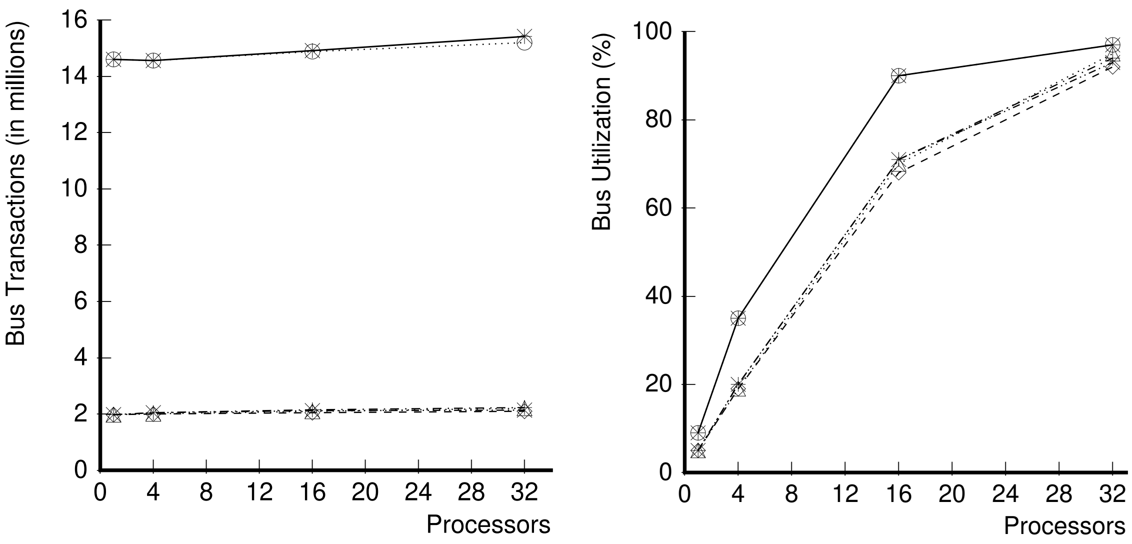


Figure 3.7: Bus Transactions (L) and Bus Utilization (R) for Gauss (128K Caches)

were needed on cache to cache transfers, despite the fact that when a cache supplies a subblock in response to a read request, it sent along all other read-shared subblocks. From this we can infer that a great many of the subblocks in each cache were owned by that cache, necessitating a separate read request for each subblock needed.

Overall, snarfing provided relatively small benefit to this application. It did reduce the number of bytes transferred by all three protocols, but only by 13% in the best case. It slightly reduced the number of bus transactions, and had a very small impact on bus utilization (Figure 3.9(L)).

MP3D

As discussed in Section 2.3, MP3D has a great deal of true sharing. Such sharing tends to limit speedups as more processors are devoted to the problem. Because of this, the maximum speedup on our simulator runs was about 17 using 32 processors. At the same time, MP3D was structured to avoid false sharing, which tends to favor large block sizes. As seen in Figure 3.10(L), the 64I protocol slightly outperformed the 8I protocol. Using the larger block size had both an advantage and disadvantage over using the 8I protocol. The advantage of using the 64I protocol is that substantially reduced (compared to the 8I protocol) the number of bus transactions required to complete the program. The reason for this is MP3D's two main data structures span multiple 8 byte blocks, which required multiple read or invalidate transactions in order to read or write each structure. The 64I protocol needed about 50% fewer invalidations, and about 60% fewer reads than the 8I protocol. The combined effect of these reductions can be seen in Figure 3.11(L), which shows the total number of bus transactions required by each protocol. The disadvantage of the 64I protocol over the 8I protocol is that it transferred more data than the program actually required; the 64 byte transfer size was, in this case, too big. This effect became worse as the number of processors was increased. When a single processor was used, the 64I protocol transferred 23% more bytes than the 8I protocol; at 4 processors, the increase was 147%, and at 32 processors it was 239%. The additional time required to transfer the unneeded bytes offset to a large degree 64I's performance advantage gained by reducing the number of bus transactions.

The subblock protocol overall performed nearly as well as the 64I protocol, and better than the 8I protocol. For both the bus transaction and bytes transferred

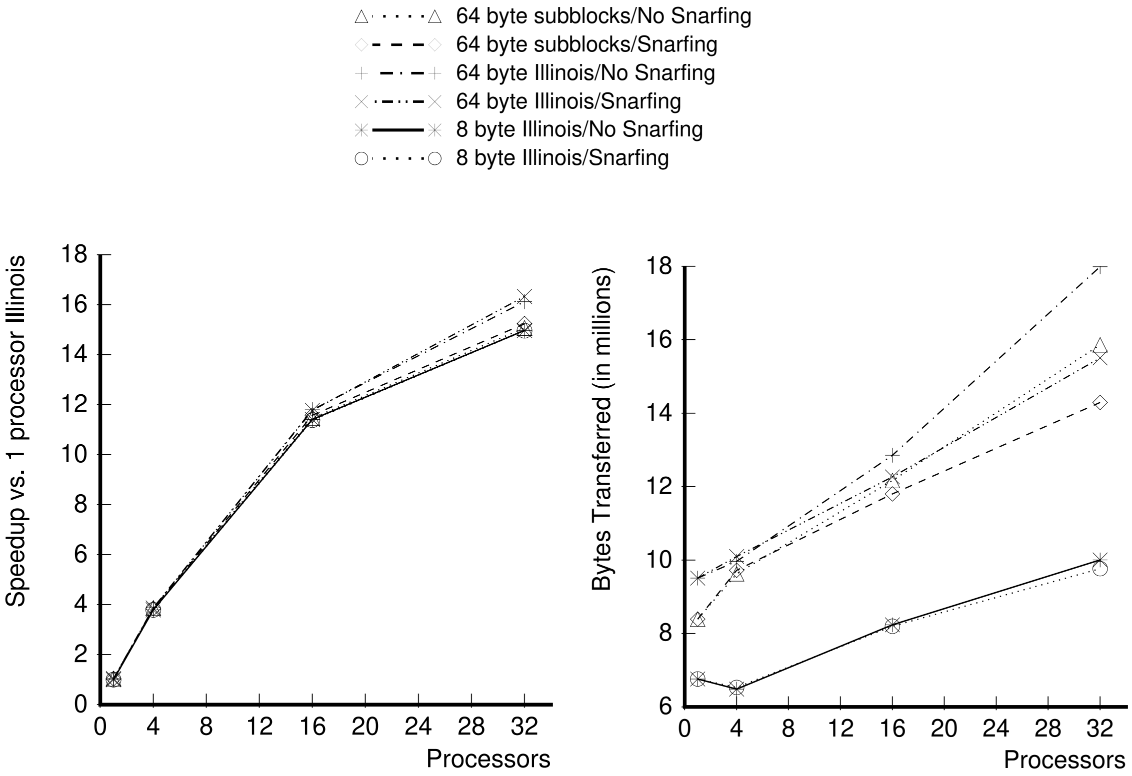


Figure 3.8: Speedup (L) and Bytes Transferred (R) for Cholesky (128K Caches)

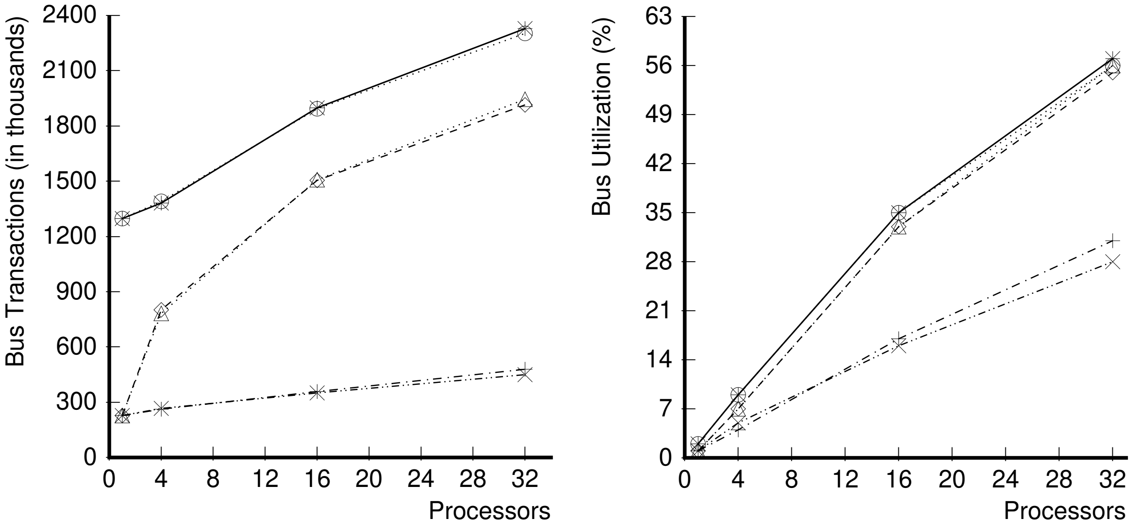


Figure 3.9: Bus Transactions (L) and Bus Utilization (R) for Cholesky (128K Caches)

metrics, the subblocks protocol performed in between that of the 64I and 8I protocols. Because the subblock protocol attempts to transfer as much of a block as possible, it reduced the number of read transactions over the 8I protocol, though not as much as the 64I protocol. At the same time, the subblock protocol did not transfer as much unneeded information as did the 64I protocol, though again it did not match the good performance of the 8I protocol.

Snarfing had little effect on MP3D, except on the 64I protocol when using 16 or 32 processors. Snarfing increased 64I's speedup by 6% on 16 processors, and by 22% on 32 processors. The performance increase was due to a reduction in bytes transferred under 64I (11% for 16 processors, 20% for 32 processors), and smaller decreases in the number of bus transactions. Snarfing had very little effect on either the number of bus transactions or bytes transferred for either the 8I or subblock protocols.

Pverify

Pverify has great deal of false sharing, even for moderate blocks sizes [EJ91]. It is not surprising, therefore, that using a large block size substantially degrades performance, as can be seen in Figure 3.12(L). In contrast, using a small block size leads to near-linear speedup. The poor performance of the 64I protocol is due to the increased number of invalidations, and the bus traffic needed to satisfy the read misses which result from the invalidations. As can be seen in Figure 3.12(R), using the 64I protocol (without snarfing) transfers over 16 times as much data as the corresponding 8I protocol when 32 processors are used. Using a large block size also increases the number of bus transactions (by 76% for 32 processors) required to complete the application (Figure 3.13(L)).

Using the subblock protocol substantially improved the performance of Pverify as compared to the 64I protocol, nearly matching the 8I protocol in speedup. The reason for this can be seen in Figures 3.12(R) and 3.13(L), where the subblock protocol reduced both the number of bytes transferred by 65% and the number of bus transactions by 29% (for 32 processors) when compared to the 64I protocol. One interesting aspect to Figure 3.12(R) is the drop in the number of bytes transferred under the subblock protocol as the number of processors increases. This is due to the fact that as the total amount of cache in the system increases, capacity and conflict misses decrease, necessitating fewer reads from memory. While this phenomenon also

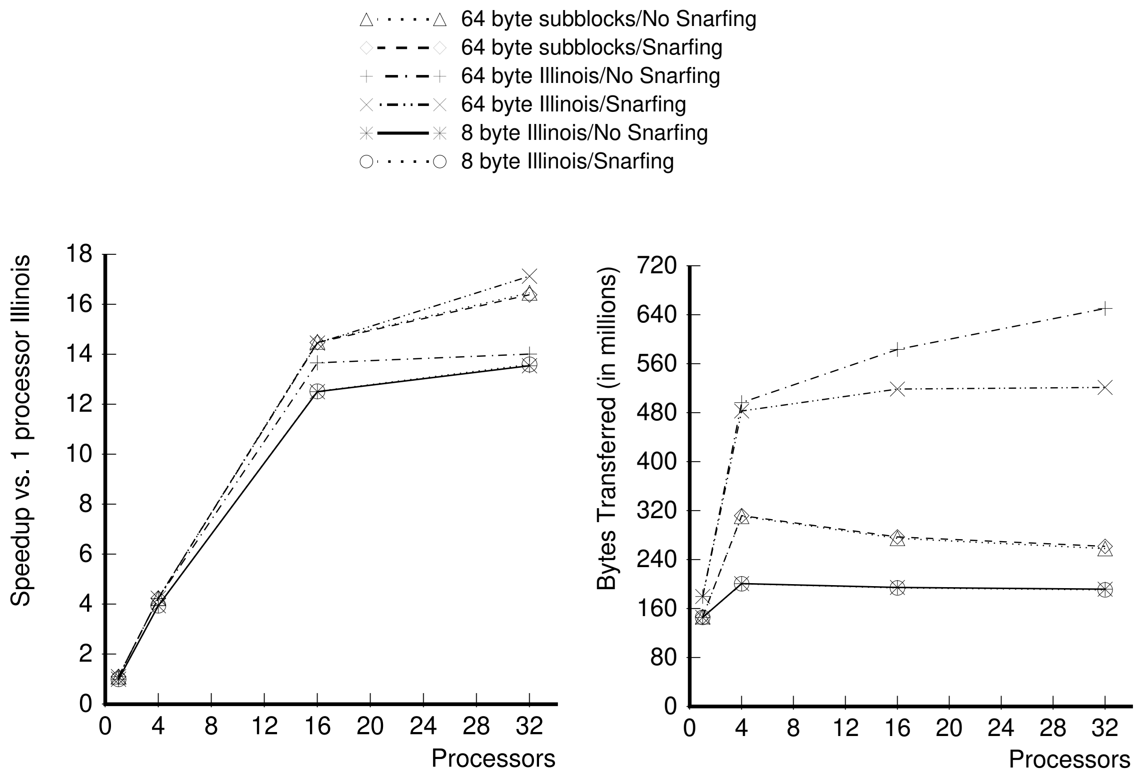


Figure 3.10: Speedup (L) and Bytes Transferred (R) for MP3D (128K Caches)

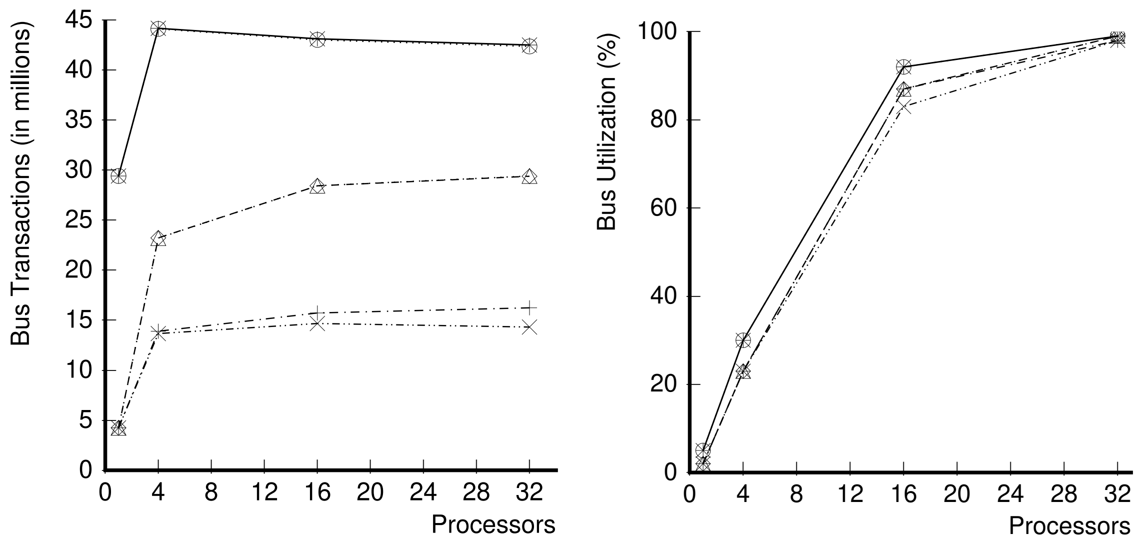


Figure 3.11: Bus Transactions (L) and Bus Utilization (R) for MP3D (128K Caches)

occurs under the 8I and 64I protocols, the effects are masked by the increase in reads due to invalidation misses.

Snarfing was a large benefit to the 64I protocol, while it was only a small benefit for the 8I protocol. For 16 processors or less, the speedup benefit was small for all protocols (at most 10%). However, at 32 processors, snarfing improved the speedup of the 64I protocol by 67%, which was still not enough to make the 64I protocol competitive with the other two protocols. The improvement was due to large reductions in both the number of bytes transferred, and the number of bus transactions. Snarfing reduced the number bytes transferred by 35% for 16 processors, and 43% for 32 processors. Improvements for the number of transactions was smaller: 24% for 16 processors, and 33% for 32 processors. Gains for the 8I protocol were much smaller. There was little difference in speedup, except for 32 processors (6%). Snarfing did improve the number of bytes transferred for the 32 processor case by 24%, and the number of transactions by 20%. We observed an anomaly in our results when we looked at snarfing under the subblock protocol. Contrary to what we expected, snarfing slightly decreased performance in all three categories of speedup, bytes transferred, and bus transactions. This occurred because snarfing increased the average number of valid subblocks in each block. Because the snarfed subblocks were clean-shared, they were sent along with the requested subblock when a cache responded to a read request. Unfortunately, for this application most of the additional subblocks were not needed. This increased bus traffic, and thus led to decreased program performance.

Toptopt

The Topopt application exhibits both true and false sharing. Because of this, using an 8 byte block size resulted in much better speedup performance than using a 64 byte block size (see Figure 3.14(L)). Using a small block size resulted in near-linear speedup, while using a large block size achieved a speedup of only 10 on 16 processors, and 17 on 32 processors. The effect of false sharing can be seen in graphs that show the number of bytes transferred (Figure 3.14(R)), and the number of bus transactions (Figure 3.15(L)). The 64I protocol transferred from 15 to 18 times more data than the 8I protocol when more than 1 processor was used. The 64I protocol also used nearly twice as many bus transactions than the 8I protocol. The decrease in the number

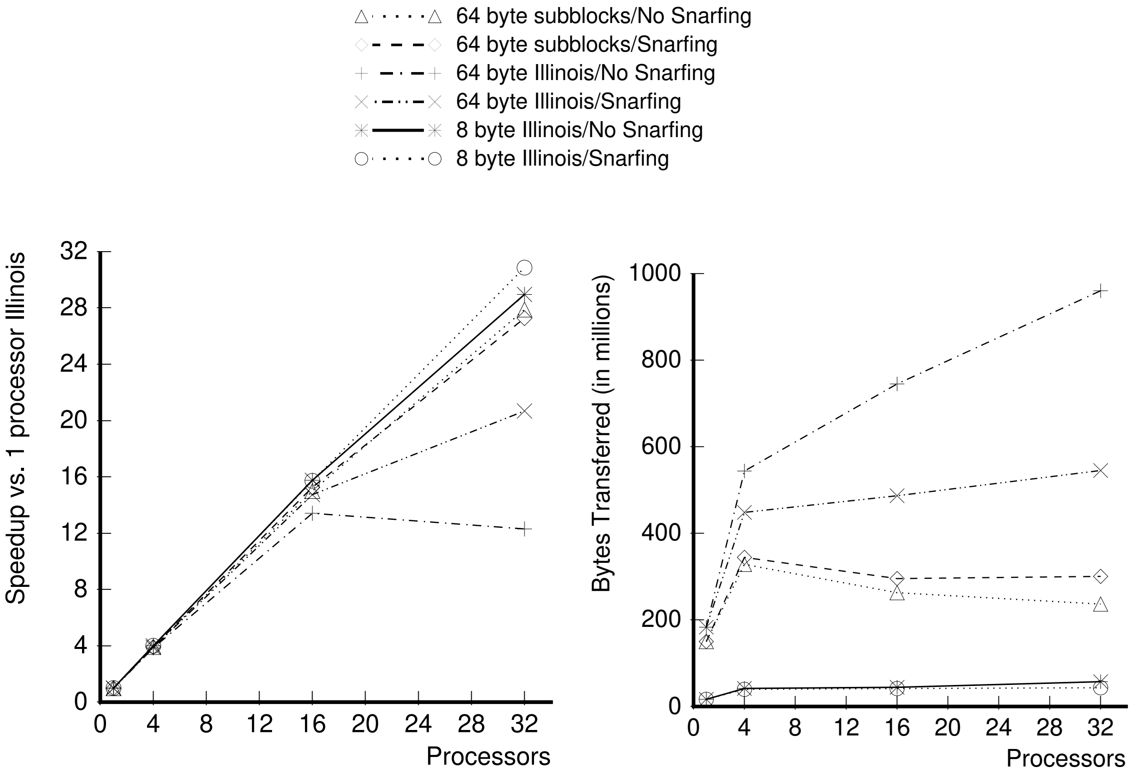


Figure 3.12: Speedup (L) and Bytes Transferred (R) for Pverify (128K Caches)

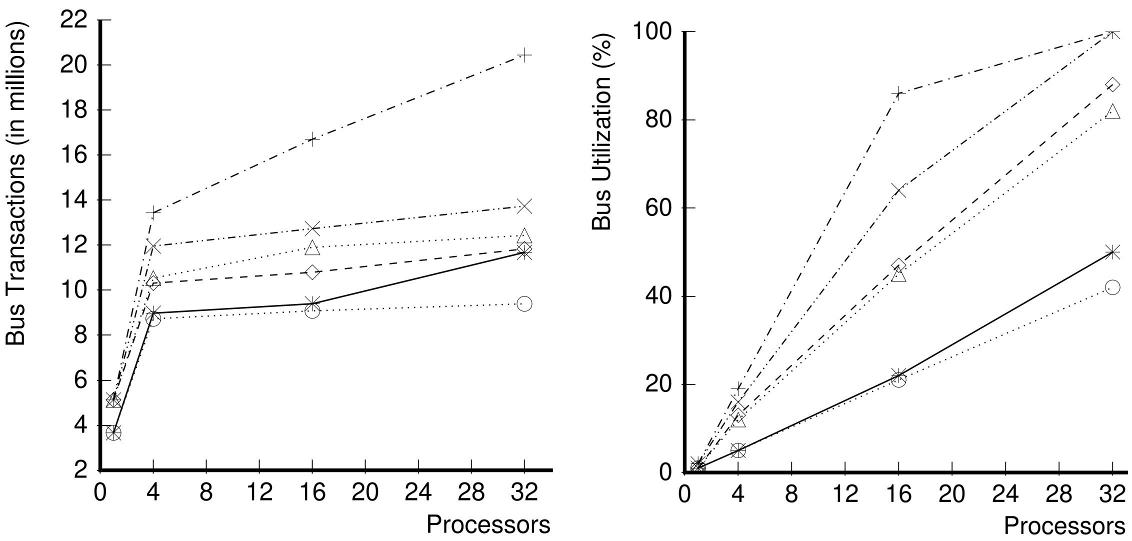


Figure 3.13: Bus Transactions (L) and Bus Utilization (R) for Pverify (128K Caches)

of bytes transferred (and bus transactions) for the 64I protocol when 32 processors were used was due to the decrease in capacity/conflict misses because of the increase in total cache space in the system.

The performance of the subblock protocol on Topopt was mediocre. Without snarfing, the subblock protocol outperformed only the 64I protocol (without snarfing) on speedup. The subblock protocol also transferred far more data and incurred more bus transactions than the 8I protocol, though it did perform better than 64I protocol in these measures. Part of the reason why the subblock protocol performed poorly in the absence of snarfing is its policy of transferring additional subblocks on cache-to-cache reads in addition to the requested subblock.

The addition of snarfing to the 64I protocol was effective in narrowing the performance gap between the 64I and 8I protocols. Snarfing increased the speedup for the 32 processor 64I case from 17 to 28. Snarfing cut the number of bytes transferred by a factor of 2 for 16 processors, and by a factor of 3 for 32 processors. It also significantly reduced the number of bus transactions needed. Snarfing reduced the detrimental effect of false sharing on the performance of the 64I protocol. In contrast, snarfing had little effect on the 8I protocol. Snarfing significantly improved the performance of the subblock protocol. With snarfing, the subblock protocol finished faster than either the 8I or 64I protocols. The subblock protocol also did nearly as well as 8I protocol in minimizing the number of bytes transferred, and had fewer bus transactions than the other protocols.

3.6.1 Results summary

Overall, the subblock protocol performed well. On five of the six applications (Gauss, Cholesky, MP3D, Pverify, and Barnes) it executed about as fast (or faster than) the better of two block sizes used with the Illinois protocol. The other application (Toptopt) performed better using the small block size. However, on Topopt the subblock protocol used with snarfing was faster than all other configurations tested. On every application, the subblock protocol transferred fewer bytes than the 64I protocol, but more than the 8I protocol. Similarly, for bus transactions, the subblock protocol usually performed in between that of the best choice of block size to reduce bus transactions (64I for Gauss, Cholesky, and MP3D; 8I for the Pverify; varying for Topopt) and the worst choice; for Barnes, it actually outperformed both.

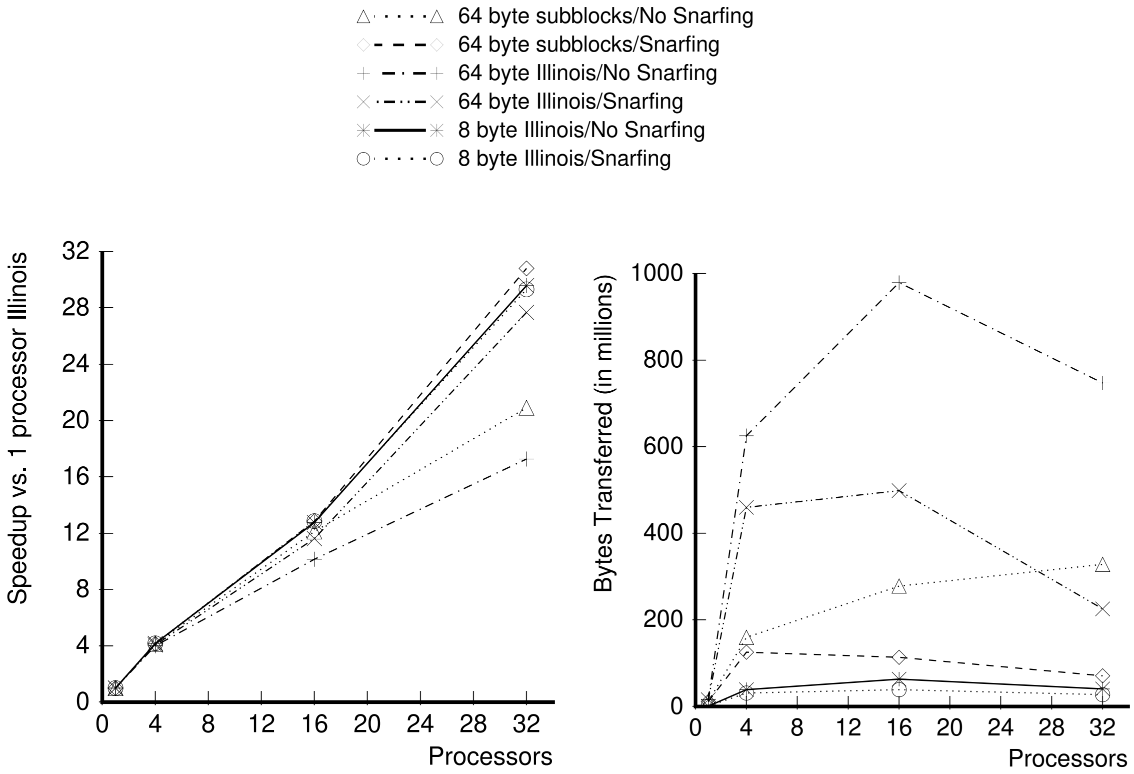


Figure 3.14: Speedup (L) and Bytes Transferred (R) for Topopt (128K Caches)

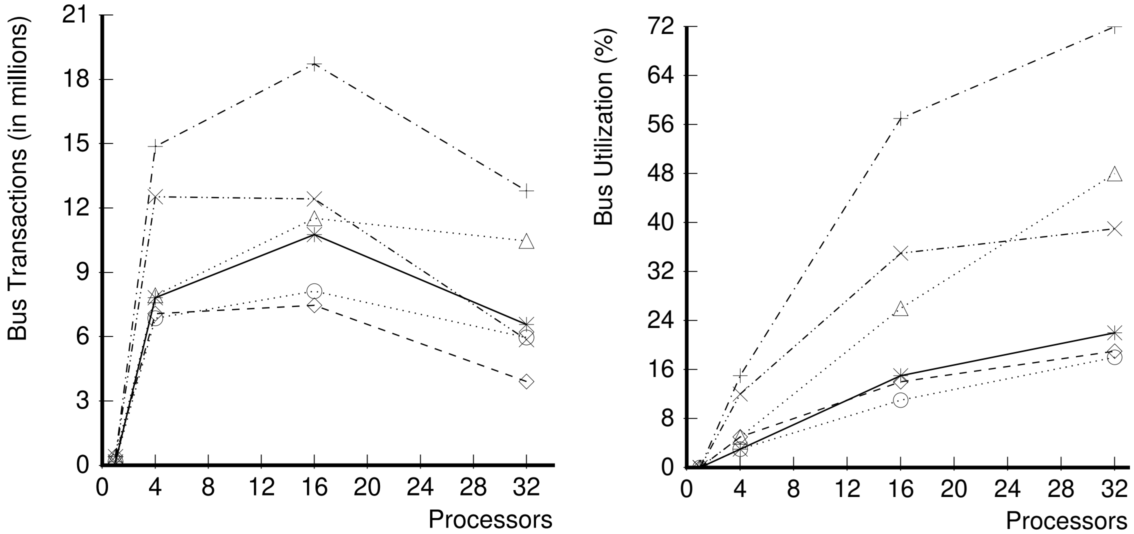


Figure 3.15: Bus Transactions (L) and Bus Utilization (R) for Topopt (128K Caches)

Generally speaking, snarfing is an effective technique to reduce the amount of data transferred and the number of bus transactions, and to improve execution time, especially when the application does not exhibit good spatial locality. Snarfing greatly improved the performance of the 64I protocol on those application with a great deal of sharing (Topopt, Pverify). Snarfing also sharply reduced the amount of data transferred, the number of bus transactions, and bus utilization for Barnes.

Table 3.2 summarizes our results for 32 processors. For each application, the speedup was normalized to the best choice of block size without snarfing. From the table, we can see that neither the small nor the large block size provides the best performance in all cases. In contrast, using subblocks with snarfing had the best performance in three of the applications, and came in second for the other three. The subblock protocol performed more consistently than using a fixed block size; for some of the tested applications, using the wrong fixed block size caused performance to drop 40% or more.

3.7 Conclusion

In this chapter, we have presented two techniques for improving application performance on single bus multiprocessors. The first technique was the use of a subblock cache coherency protocol that takes advantage of the spatial locality found in many programs by using a large sized block for transfers, while avoiding the performance problems caused by false sharing by using a smaller sized subblock for invalidations. The second technique was read snarfing whose intent is to reduce the number of read misses caused by previous coherence actions.

The subblock protocol performed as well or better than the Illinois protocol using the best block size (64 bytes) on those applications with good spatial locality. The subblock protocol also performed well on the applications that exhibited a great deal of true and false sharing. For these applications, it was always better than the Illinois protocol with large block size and was close to the Illinois protocol with small block size. When the subblock protocol did not do as well as the Illinois protocol (when using a small block size), the problem was due to the subblock protocol's use of a large transfer block. Our experience shows the difficulty of creating a non-adaptive cache coherency protocol that works well with all applications. The policy of using a large transfer size in our subblock protocol that helped us get good performance in

Table 3.2: Relative speedups for 32 Processors

Program	No snarfing			With snarfing		
	Small Blocks	Large Blocks	Subblocks	Small Blocks	Large Blocks	Subblocks
Gauss	.47	1.00	1.05	.48	1.03	1.07
Cholesky	.92	1.00	.93	.92	1.01	.94
MP3D	.97	1.00	1.17	.97	1.22	1.17
Topopt	1.00	.58	.70	.99	.94	1.04
Pverify	1.00	.42	.96	1.06	.71	.96
Barnes	1.00	.93	.98	1.00	.99	1.00

The speedup for each application running on 32 processors is normalized to the best choice of block size for that application when running under the standard invalidate protocol. For example, Gauss executes best with the large block size, so its speedups are normalized to the no snarfing, large block speedup.

the Gauss application hurts performance in Topopt. Overall, however, the subblock protocol was either best or close to the best of all protocols.

We also investigated the effects of read snarfing, and found that it is an effective technique in improving application performance on bus-based machines, especially when large block sizes are used and the program exhibits a great deal of false sharing. Snarfing is effective in reducing the amount of data transferred and the number of bus transactions, which in turn leads to lower bus utilization and higher application speedups.

The combination of the subblock protocol and snarfing produced the best results in 2 out of 6 cases; it came in a close second in the other 4 cases.

Chapter 4

ADAPTIVE HYBRID CACHE PROTOCOLS

4.1 Introduction

Applications exhibit a wide variety of reference patterns. It is well known that depending on the application, either write-invalidate (WI) or write-update (WU) performs best [EK88]. Indeed, even within applications, various forms of sharing behavior exist: for example, widely write-shared data (like locks), largely read-only data, and migratory data [CF93, SBS93]. Because of these different sharing patterns, varying the protocol as the application executes has the potential to increase application performance by decreasing both the number of bus transactions and the number of bytes transferred over the bus, leading to decreased bus contention and reduced latency for memory operations.

A key concept in choosing between WI and WU is that of the *write run* [EK88]. Eggers defines a write run as a sequence of write references by one processor to a shared *address*, uninterrupted by any accesses by other processors. We will use a slightly different definition of a write run in our discussions. Our changes are motivated by the observation that in practice, writes to different addresses in the same cache block have the same effect on the memory coherence system as writes to the same address. In addition, replacing a cache block effectively ends a write run. Therefore, we define a write run as a sequence of write references to a shared *block*, uninterrupted by either an access by another processor, or *replacement of the block*. The length of a write run, then, is the number of writes performed in the run. By definition, all writes in a given write run are done by only one processor. Note that our definition reduces to Eggers' when using infinite caches and block size of 1.

The difficulty in choosing between WI and WU occurs because average write run lengths vary from block to block and application to application. If block accesses show long write runs, then WI is best, because only the first write incurs the bus traffic penalty of an invalidation (and subsequent reread by at least one other processor)

[1]; all subsequent writes in the write run hit in the cache with no protocol action required. On the other hand, if write runs are short, WU is best, since the cost of a single update is cheaper than the cost of an invalidate.

The difficulty, then, is choosing the proper strategy. An *off-line optimal* policy examines the future behavior of the program to determine whether to use WI or WU for each write run of a particular block. Since looking into the future is impossible, various methods exist to try to approximate the behavior of the optimal algorithm. Such strategies usually consist of initially using WU on a line, and switching to WI after some number of writes to the block. Hybrid strategies can be divided into two classes: those in which the *sender* of updates (i.e. the writing processor) decides when to switch from WU to WI, and those in which the *receiver(s)* of the update (collectively) makes the decision. A receiver helps to choose the strategy by deciding whether to retain or invalidate a block for which it receives an update. If all receivers of the block invalidate their copies, then the sender changes the block's state to exclusive, and updates are no longer sent. In the following three schemes, the receivers of the block decide when to change to a WI strategy.

Karlin *et al.* proposed an algorithm which was proved to be within a factor of 2 of the optimal off line strategy [KMRS86]. The algorithm works by associating a counter with each cache block. On an update transaction, a receiver of the update (a cache which currently shares the block) is chosen at random, and its associated block counter is decremented. When a block's counter reaches 0, the block is invalidated. When all shared copies of a block have been invalidated, updates are no longer sent, and the block in the writer's cache becomes exclusive. The initial value of a counter is called the *invalidate threshold* (T). A counter is reset to the invalidate threshold whenever the block is read by the processor.

A simpler scheme, called "Snoopy Reading" (which doesn't require choosing a cache at random) involves decrementing the counts for all copies of the shared block. This results in the writer's block becoming exclusive faster, since all caches sharing the block decrement their counts on each write, rather than just one [EK89]. In addition, block counters are reset to the invalidate threshold if a write update is received from a processor which is different from the one that issued the previous write to the block.

---

[1] For the rest of the chapter, *invalidation cost* will refer to the cost of an invalidation transaction, plus all block rereads due to the invalidation.

This is done so that a block which is being written by more than one processor will continue to be broadcast, since the block is being shared. Unfortunately, this requires some hardware mechanism to determine when writes originate from different processors.

Archibald introduced an improved protocol in which a block is not automatically invalidated when its count reaches 0 [Arc88]. Instead, a cache simply doesn't raise the shared line if the relevant block's count is 0. If some cache whose block count is not 0 indicates that it is keeping the block by raising the shared line, then no cache invalidates its block. The block is invalidated in the sharing caches only when all counts for the block are 0. This scheme also has the advantage that a succession of single writes from different processors will not cause other copies to be deleted. It does this without any additional hardware required.

Eggers and Katz evaluated the "snoopy reading" scheme against a pure update scheme [EK89]. Using a value of 3 for the threshold, they found that while the hybrid scheme reduced the number of updates for all 4 benchmarks used, two of the four programs executed slower than WU while using the adaptive scheme. The benchmarks ran slower because the time and bus traffic saved by doing fewer updates were outweighed by the traffic caused by the invalidations.

Veenstra and Fowler evaluated the cache coherence protocol used by the DEC Alpha [TCS92], which uses the Snoopy Reading scheme with an invalidate threshold of 2 [VF94b]. They reported that using a threshold of 2 resulted in shared blocks being invalidated too quickly, thus applications did not gain from selective updating. Increasing the invalidate threshold in order to get more updating behavior does have drawbacks, however. A processor executing a moderate or long write run will have to pay for several unnecessary updates before it can invalidate. This is a problem with using a static value for the invalidate threshold.

4.2 Two dynamic hybrid protocols

In this section, we describe two dynamic hybrid protocols. Both protocols are adapted from competitive thread-spinning algorithms found in the paper by Karlin *et al.* [KLMO91]. In the paper, the algorithms decide when to switch between spinning a thread while waiting for a lock — an inexpensive operation that can be repeated many times — and blocking a thread, which is an expensive operation that is done only once. In our algorithms, WU is analogous to spinning, while WI is analogous to blocking. One difficulty in adapting these algorithms to cache coherence protocols is that unlike the case for blocking or spinning a thread, the cost of an invalidation depends on the number of times a line is re-read due to the invalidation. Depending on the stream of references, the number of rereads could vary between 0 and n-1, where n is the number of processors. However, if read snarfing (as defined in the previous chapter) is used, the first re-read fills all other copies of the invalidated block, making it easier to fix the cost of an invalidation. In the absence of read-snarfing, we must estimate the number of re-reads done. Previous work by Gupta *et al.* has shown that most blocks are not widely shared [GWM90]. Therefore, in our experiments where we did not used read snarfing, we estimated a small number of re-reads for every line.

It is important to know the cost of an invalidation because it is used to calculate the *invalidation ratio*. The invalidation ratio (R) is the ratio between the cost in bus cycles of an invalidate and some number of rereads of the block due to the invalidation, and the cost of an update. If a static hybrid protocol uses the invalidate ratio for the invalidate threshold, it will do no worse than twice the cost of the optimal algorithm [KMRS86]; however it is possible to do better if the value of the threshold is varied as the program runs. The following two algorithms vary the threshold from 0 up to (and including) the value of the invalidate ratio in order to improve performance. In addition, unlike the previously discussed adaptive strategies, the decision to switch to invalidate is made by a block's writer, not the caches that receive the update.

4.2.1 Random Walk

In the Random Walk algorithm, a dynamically variable invalidate threshold is associated with each block of main memory. When the length of a processor's write run exceeds the current threshold, a WI protocol is used. The threshold is adjusted as follows: if the length of a processor's write run exceeds the invalidate ratio, then the current threshold value is decreased by 1, until it reaches 0. If the processor's write run is less than or equal to the invalidate ratio, then the threshold is increased by 1 until it reaches the invalidate ratio. By this mechanism, those blocks which have long write runs will sooner or later have their thresholds reduced to 0, which means that the block will be invalidated immediately, with no wasted updates. Similarly, if a block has short write runs, then the threshold will be adjusted upwards, so that the block will not be invalidated when it is being actively shared.

For example, assume that the invalidate ratio is 5, and the current threshold for a particular block is 3. A processor then begins a write run for the block. If the processor only writes the block 4 times before it is accessed by another processor, then the threshold will be increased by 1 (to 4). On the other hand, if the block is written 10 times before being accessed by another processor, then the threshold will be decreased to 2.

The Random Walk algorithm requires additional memory to store the current threshold. However, since the threshold never exceeds the invalidate ratio, the addition of 3 to 5 bits per memory block is not a large overhead. Given a 64 byte block, the additional amount of storage required even for 5 bits is 1%. Also, we assumed additional bus lines exist to transmit threshold values to and from processors and memory.

4.2.2 Last three samples

The last three samples (LTS) algorithm uses the length of the last three write runs for a block to set the invalidate threshold. The algorithm estimates the distribution of write run lengths for this block by assuming that each of the three previous samples represents one-third of the real distribution. A static table is computed that, given the values of the last 3 write runs, returns which of the three values (or 0) gives the minimum cost when used as the invalidate threshold.

To explain how the table is derived, we first discuss how the threshold would be chosen if the entire write length distribution for the block were known. First, we make the observation that if the write threshold is i, then a processor executing a write run of length w will perform w update transactions, if $w \leq i$. If $w > i$, then i updates will be performed, plus an invalidate transaction and some number of block rereads due to the invalidation. If, for any given block we knew the complete run length probability distribution, we could choose the threshold (i) which minimized the number of bus busy cycles. To do this, we would find the expected cost of using i as a write threshold ($EC(i)$) for all possible values of i, then chose the value which minimizes ($EC(i)$). Given our observations above about the costs of using a given write threshold, the equation for $(EC(i))$ is:

$$EC(i) = \sum_{j=1}^{j \leq i} Pr(w = j) * j + \sum_{j>i}^{\infty} Pr(w = j) * (i + R) \tag{4.1}$$

where:

w	is the length of the write run,
$Pr(w = j)$	is the probability that the write run is of length j, and
R	is the invalidate ratio.

In the above equation, the costs are normalized by the cost of an update. Thus, the cost of an update becomes 1, and the cost of an invalidate becomes R, since $R =$ invalidate cost/update cost. Now let i, j, and k be the values of the last three write runs, and let $0 \leq i \leq j \leq k \leq 2R$. Using equation (4.1), and our assumption about the probability distribution, we obtain:

$$\begin{aligned} EC(0) &= R \\ EC(i) &= \frac{1}{3}i + \frac{2}{3}(i + R) \\ EC(j) &= \frac{1}{3}i + \frac{1}{3}j + \frac{1}{3}(j + R) \\ EC(k) &= \frac{1}{3}(i + j + k) \end{aligned} \tag{4.2}$$

The value which gives the lowest cost is used as the threshold. Note that we limited the values of i, j, and k to $(2 * R)$ to keep the size of the table manageable, though in practice limiting the values further to R would not have affected the actual threshold values determined by the algorithm.

When no write history is available for a block, we used 0 as the threshold. When a block had only 1 or 2 write runs, the thresholds were determined by deriving equations similar to Equation 4.2 under the assumption that the distribution is determined by 1 or 2 samples, rather than 3.

The overhead for the LTS algorithm is substantially higher than that of the Random Walk algorithm. Though the LTS doesn't store the current threshold, it does store the lengths of the last three write runs for every memory block. Additional storage is needed for the threshold lookup table. The additional storage needed makes LTS a less practical algorithm than Random Walk.

4.2.3 Methodology

Using the Cerberus simulator, we simulated both adaptive strategies, as well as a WI and WU protocol. The WI protocol we used is the Illinois protocol [PP84], while the WU protocol we used is the Dragon protocol [McC84]. An important aspect to note about the Dragon protocol is that it uses a shared line to detect if a block being updated is being actively shared. If the block is not being shared by any other processor, it is marked as exclusive in the writer's cache, and no further updates are done for the length of the write run. Thus, an application which exhibits long write runs need not have poor performance under WU, as long as the block is not shared by another processor. The adaptive protocols also use the shared line to stop updates when the block is not actively shared.

Because the benchmarks we used favored WU, we constructed a benchmark that would do poorly with WU. We modified the Gauss benchmark so that all processors access the entire array before starting the row reduction operations. This is done so that the entire array is shared in many processors, forcing an update protocol to do an update transaction for every write. We then ran the modified version of Gauss with a large cache so that the entire array would stay resident during the parallel section of the program. We call this program Badgauss.

Except for the Badgauss benchmark (where we used a very large cache size), we simulated both 32K and 128K processor caches. We used a relatively small bus size of 32 bits so that the cost of an invalidate would be significantly higher from that of an update. The relatively small bus makes it more expensive to read an entire cache block than to update the part of the block which has been written. Using an adaptive scheme is appropriate only when there is a significant difference between the total cost of invalidating and updating. The block size simulated was 64 bytes. Given these parameters, the cost of an invalidate transaction (i) was 2 cycles, the cost of reading a cache block from another cache (r) was 19 cycles, and the cost of an update (u) was 4 cycles.

An important input to the adaptive algorithms is the invalidate ratio, which is the ratio between the price of the high cost operation (invalidation) versus the cost of the low cost operation (update). When read snarfing was used the number of block re-reads due to a given invalidation (n) is 1. Using our formula for the invalidate ratio (found above), we found that the invalidate ratio was 5. Without snarfing, we estimated the ratio to be 18.

For the Random Walk algorithm, the initial value of the invalidate threshold was set at 0 (invalidate immediately). Similarly, the threshold used for the LTS when there were no prior samples was 0. Varying the initial value showed no appreciable affect on the results of the adaptive protocols.

4.3 Computed costs of the schemes

A major reason why programs fail to experience linear speedup on bus-based multiprocessors is bus contention. Thus it is important to reduce the number of bus busy cycles where possible. In this section, we discuss the bus-cycle costs of each coherence scheme. We use the following variables in our discussion:

i The cost of an invalidate transaction.

n The number of times a given block is re-read due to a particular invalidation.

r The cost of reading a cache block.

u The cost of a write update transaction.

w The length of a write run.

Thus, given the above notation, the invalidate ratio R is $(i + n * r)/u$.

For a standard invalidate protocol, the cost of maintaining coherence for a particular write run doesn't depend on the length of the run. Given the above notation, the cost C_i is:

$$C_i = i + n * r$$

While i and r are fixed quantities, the value of n can vary between 1 and one less than the number of processors. If read snarfing is used, n is always 1.

For a Dragon-like update protocol, an update transaction must be done for every write to a block. Therefore, the cost C_w is:

$$C_w = u * w$$

An off-line algorithm is able to look into the future to see how long a given write run will be. An *optimal* off-line policy will use update if the length of the write run will be less than the invalidate ratio (R); if it is greater, then the block will be invalidated immediately, with no updates used. Formally, the cost C_o is:

$$C_o = \begin{cases} u * w & \text{if } w \leq R \\ i + n * r & \text{if } w > R \end{cases}$$

Though the optimal off-line policy is not practical, it is useful as a lower bound on the performance of cache coherence policies.

Because both of the dynamic hybrid policies differ only in how they compute the current threshold T, the method used to compute the cost of the two schemes is the same. Under both policies, a block is updated until the write run length reaches the current threshold; on the next write, the block is invalidated. Thus the cost for both the Random Walk (C_r) and LTS (C_l) schemes is:

$$C_r = C_l = \begin{cases} u * w & \text{if } w \leq T \\ (u * T) + (i + n * r) & \text{if } w > T \end{cases}$$

Because of optimizations in both the invalidate (Illinois) and update (Dragon) protocols studied, the above equations overstate costs to some degree. In the case of the Illinois protocol, blocks read from main memory may be written without an invalidate transaction. For the Dragon protocol, if a processor sends an update for a block which is no longer present in any other cache, the processor will no longer send out updates. Thus in practice, looking at write runs lengths only gives a partial view of a program's write behavior.

To evaluate the theoretical costs of the various coherence schemes, we gathered data about the write run lengths for our benchmarks, as well as information about invalidate thresholds versus write run lengths for the adaptive hybrid protocols. Unfortunately, this information was not sufficient to evaluate the 4 on-line protocols against the optimal protocol, as we will see in the next section.

4.3.1 Evaluating theoretical costs

Ideally, we would like to evaluate the bus costs of the four protocols against the oracle policy. The evaluation would tell us how close each came to the minimum cost of enforcing cache coherence. Unfortunately, because the oracle policy requires knowledge of the future, it would be computationally very expensive to simulate it. In addition, since the threads of an application interact with each other to alter memory access orderings, rolling back a simulation to change the coherence policy for a block could change the access ordering by other threads to other blocks in a program, possibly changing the length of those block's write runs and causing previously optimal decisions to become non-optimal.

Another way of determining the optimal cost would be to use a trace record of the write run lengths for each block. The trace could then be processed, and the bus costs determined by using the above equations. Regrettably, this method also has problems. The main difficulty is that a write trace for a given program, generated using one protocol can give wildly incorrect cost values when used to determine the cost of using another protocol.

Assume that we use a write trace generated when using the Write Invalidate protocol to measure the cost of the Write Update protocol. A naive method for computing WU's cost for a given write run would be to assume that an update would be sent for every write to the block. However, this would in many cases overestimate the cost of WU, because the Dragon protocol (our WU protocol) only sends out updates if the block is present in other caches. To try to correct this problem, a notation in the trace could be made for those situations where a block is not shared at the time of the invalidation, and thus WU would not be charged for the updates (beyond the first) in that write run. Nevertheless, there are circumstances when using an update protocol in which a block is shared at the beginning of a write run, but becomes exclusive part of the way through the run because the block has been replaced in the other caches sharing a block. This situation could not be reflected in a trace gathered using an invalidate protocol.

An alternative would be to use WU to gather the trace, which would then be used to compute the cost of WI, the adaptive protocols, and the optimal policy. Regrettably, there are problems with using this method to compute the cost of using invalidate transactions to maintain coherency. Under a certain sequence of read and write transactions, a block kept coherent using a WU protocol will end up in a shared state, while the same block will end up in a non-shared (Valid Exclusive) state when invalidate transactions are used. The problem is that if we are using WU to generate the write run trace, the trace will indicate that the block in question is shared, which is incorrect when the trace is used to evaluate the cost of WI.

Figure 4.1 shows an example of this phenomenon. At time T0, block **X** is present in two caches. At time T1, processor 1 writes to block **X**, which updates processor 2's cache under WU, but invalidates processor 2's copy under WI. In steps 2 and 3, block **X** is replaced and then re-read by processor 1. At this point, the block is shared under WU, but it is exclusive under WI. The subsequent write to **X** at T4 requires a bus transaction under WU, but no transaction under WI. If a write trace is gathered using WU and used to evaluate the cost of WI, the WI protocol will be incorrectly charged for the cost of an invalidate at T4 (and subsequent re-read of the data), thus overstating the cost of using WI. This sequence of operations occurs quite frequently in the Gauss application. It often arises in applications when one thread initializes all of the shared data structures in an application.

It is unfortunate that we are unable to evaluate the four protocols against the optimal strategy. However, this situation does point out the advantage of instruction-level simulation over trace driven simulation — namely, the actions of changing the cache coherence protocol, or even changing the size of a cache can have a large impact on the type and order of memory events in the system, which will not be reflected in the trace.

4.4 Write runs

In this section we present data for the lengths of write runs for the benchmarks we selected. We used the same bus and cache parameters as for the simulation runs where we evaluated the performance of the four protocols. The graphs are for runs without read snarfing. While read snarfing doesn't directly affect the length of write runs, it sometimes does change the timing of various load/store events, which can have an effect on write run lengths. We show write run length graphs when using 128K caches; in those cases where the run lengths are significantly different for 32K caches, we show those as well.

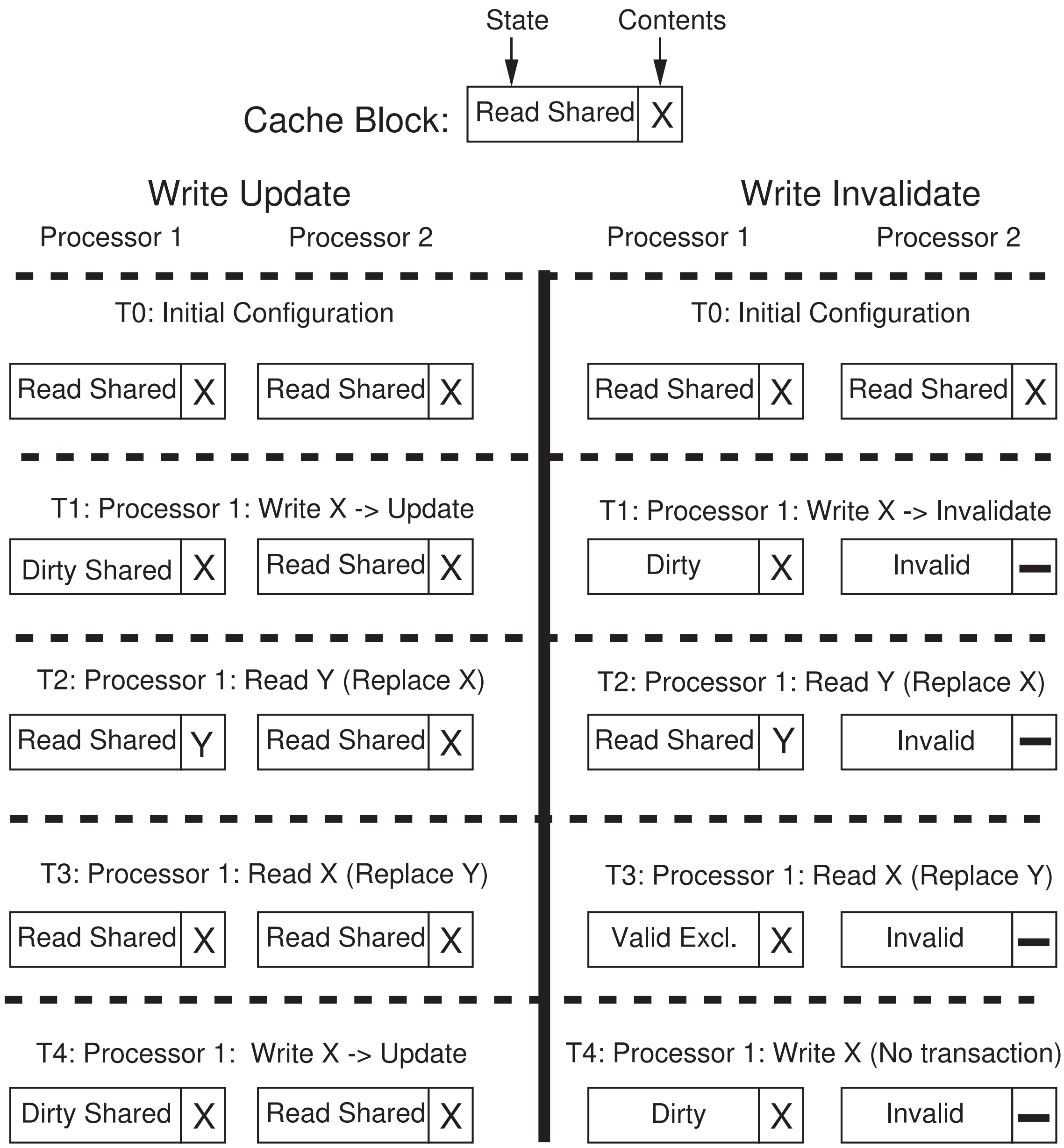


Figure 4.1: Write trace example

Barnes

The Barnes-Hut application has a well-spread distribution of write runs (Figure 4.2). When four processors are used, about 49% of all runs are of length 2 or less, yet 25% are longer than 21. When the number of processors is increased, the total number of write runs increased dramatically, with a higher proportion of short write runs. The use of additional processors caused data to be more widely shared, which caused single long write runs to split into many shorter write runs. There were no significant difference in the distributions when 32K caches were used, since the working set is small.

Given the distributions, one might expect that an update protocol would perform best on Barnes, especially if the long write run blocks were exclusive in the writing processor's cache, which avoids the poor behavior (i.e. many update transactions) of update protocols. However, given that Barnes has relatively low amounts of sharing, the choice of protocol likely will have a small effect on overall performance.

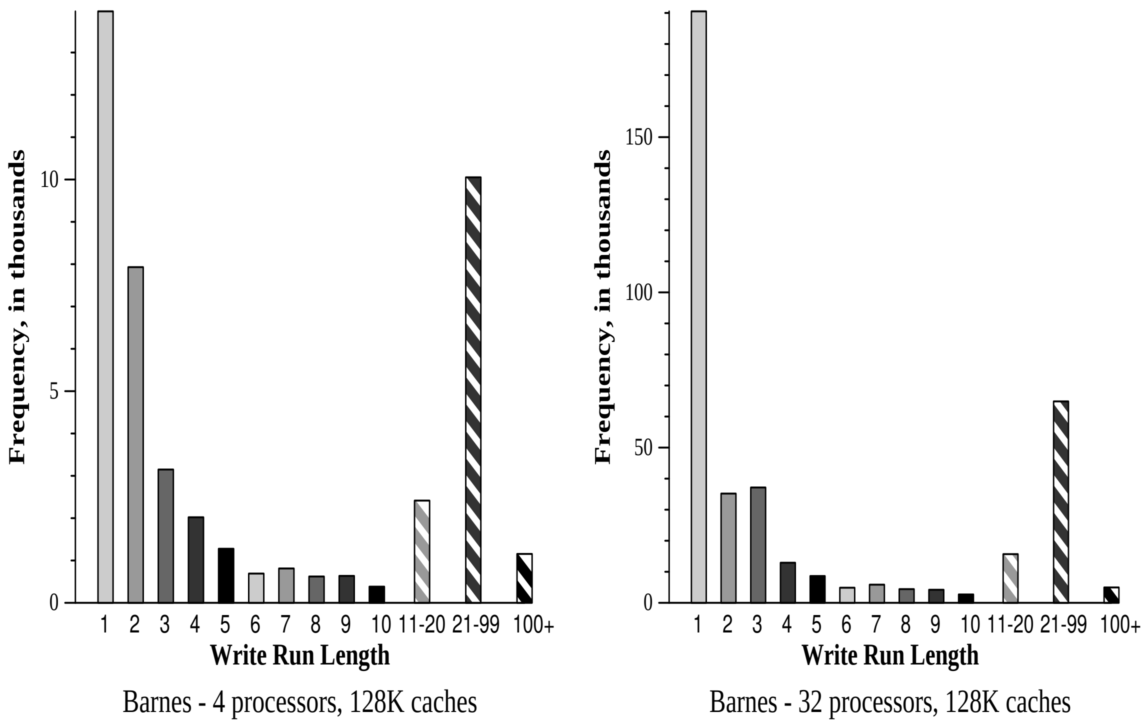


Figure 4.2: Write run lengths for Barnes

Gauss

The most noticeable feature of the Gauss data is the large spike at a write length of 8 (Figure 4.3). The spike is expected due to the access patterns in the program. As each processor modifies its rows, it writes every element to the right of the current column. Since every 64 byte cache block can contain 8 double precision floating point values, most write runs are 8 long. As discussed in Section 2.3, the working set for Gauss fits in less than 512K; however, even with 32 128K caches, conflict misses still cause blocks containing some matrix elements to be evicted on each iteration. Thus there are not many write runs of length greater than 8.

In the absence of a sharing line, a WU protocol would perform worse than WI on Gauss, since the vast majority of writes exceed the breakeven point (invalidate ratio) of 5. As we shall see, however, the presence of the sharing line enabled WU to perform as well as WI.

Badgauss

The Badgauss benchmark was run with larger caches (512K) and greater associativity (32) than the Gauss benchmark. Since conflict and capacity misses were all but eliminated, the distribution of write run lengths shifted from Gauss's many runs of length 8 to runs of length greater than 100 (Figure 4.4). In addition, since (by design) the algorithm makes the entire array actively shared, under WU the sharing line will be activated on every write to the array. Thus under WU all writes to blocks containing array elements will incur a write update transaction. This should cause a large drop in performance versus WI, especially when large number of processors are used. The adaptive algorithms should do well on Badgauss, since the long write runs will cause blocks to be invalidated right away.

Cholesky

Like Gauss, Cholesky also has a spike at write runs of length 8 (Figures 4.5 and 4.6). However, unlike Gauss, Cholesky also has many write runs of length other than 8. When 4 processors were used with 32K caches, Cholesky had a wide distribution of write runs, with a large number of runs in the 11–20 and 21–99 range. When 128K caches were used, the length of most write runs (except those of length 100 or

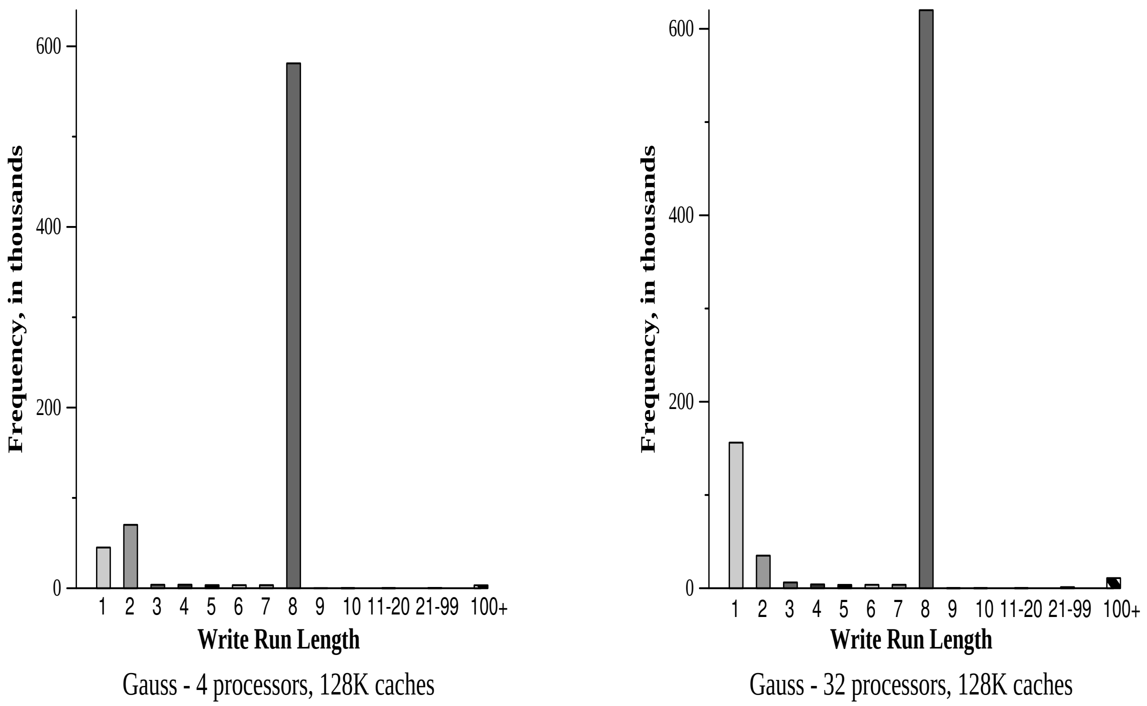


Figure 4.3: Write run lengths for Gauss

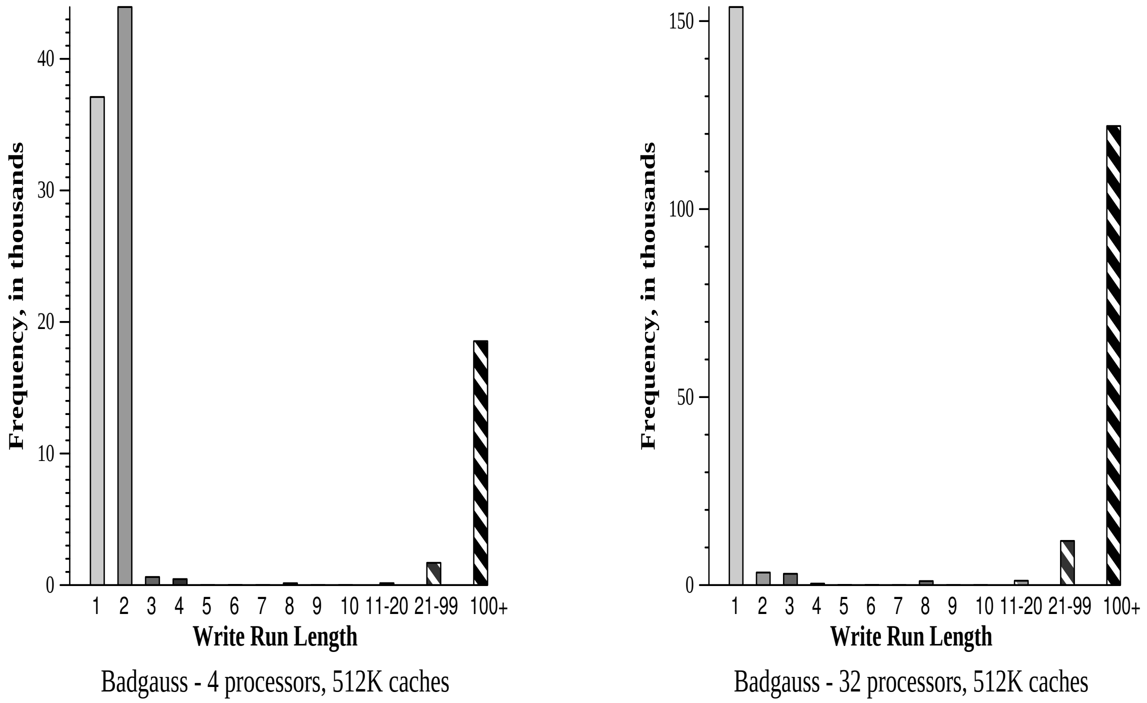


Figure 4.4: Write run lengths for Badgauss

greater) increased. When 32 processors were used the number of runs of length 1 and 8 dramatically increased, though there were still a non-trivial number of longer write runs. Unlike the 4 processor case, there were few differences between the 32K and 128K cache runs.

Cholesky's long write runs would tend to favor an invalidate protocol over an update protocol. However, since the WU protocol we chose does not send updates for blocks that are not currently shared, it is difficult to tell from just the write run data if the WU protocol will be able to do as well as the WI protocol.

MP3D

MP3D exhibits a great deal of true sharing in one of its two main data structures, therefore it is not surprising that it has a large number of short write runs, especially of length 1 (Figure 4.7). When 4 processors were used, there were also a respectable number of runs (800,000) of length 11–20, and 146,000 runs of length between 21 and 99. When the number of processors was increased to 32, write runs in general decreased in length. The number of runs of length 1 increased by nearly 50%, while the number of runs of length 11–20 decreased to 235,000 and the number of runs of length 21-99 decreased to 29,000. The write run lengths decreased because the additional processors shared data to a higher degree.

Because MP3D has a large number of short write runs, update-based protocols should perform better than invalidate protocols, especially when larger numbers of processors are used. However, MP3D does have a significant number of longer write runs, which can cause an update protocol to send out a large number of updates if the blocks being written are present in two or more caches.

Pverify

Because of large amounts of both false and true sharing, the vast majority of write runs in Pverify are very short — 1 or 2 writes before the block is referenced by another processor. Such write patterns heavily favor update-based protocols. As Figure 4.8 shows, increasing the number of processors to 32 greatly increased the number of write runs of length one, while changing the cache size had little effect on write run distributions for either 4 or 32 processors.

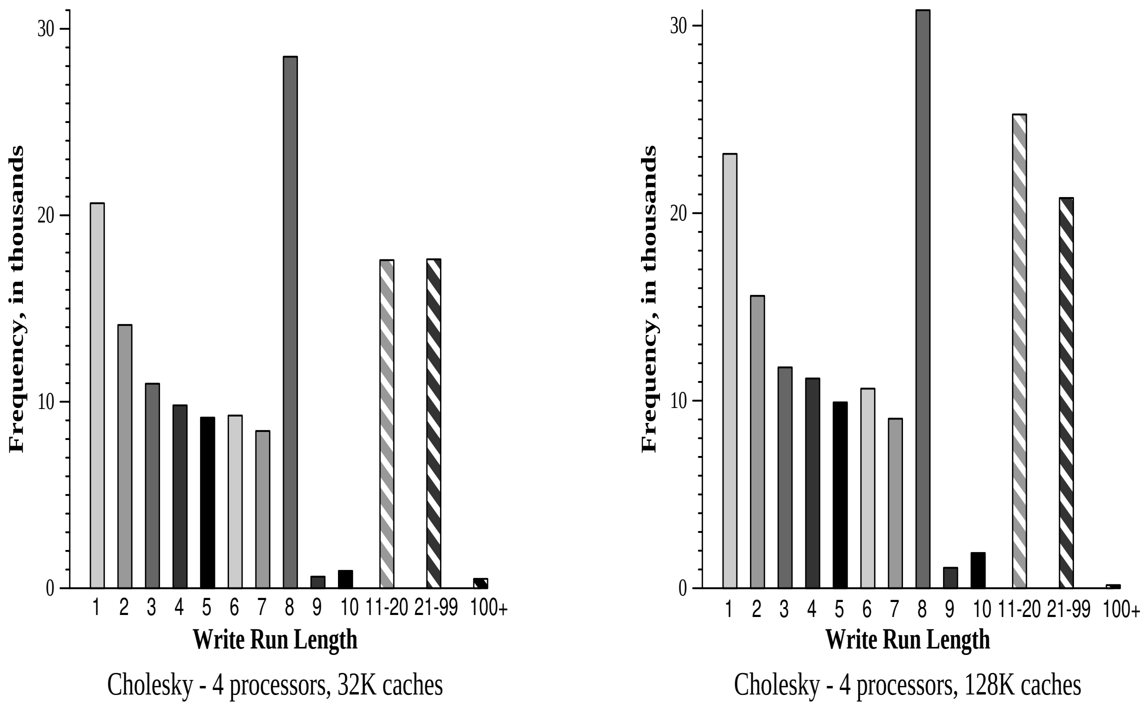


Figure 4.5: Write run lengths for Cholesky, 4 processors

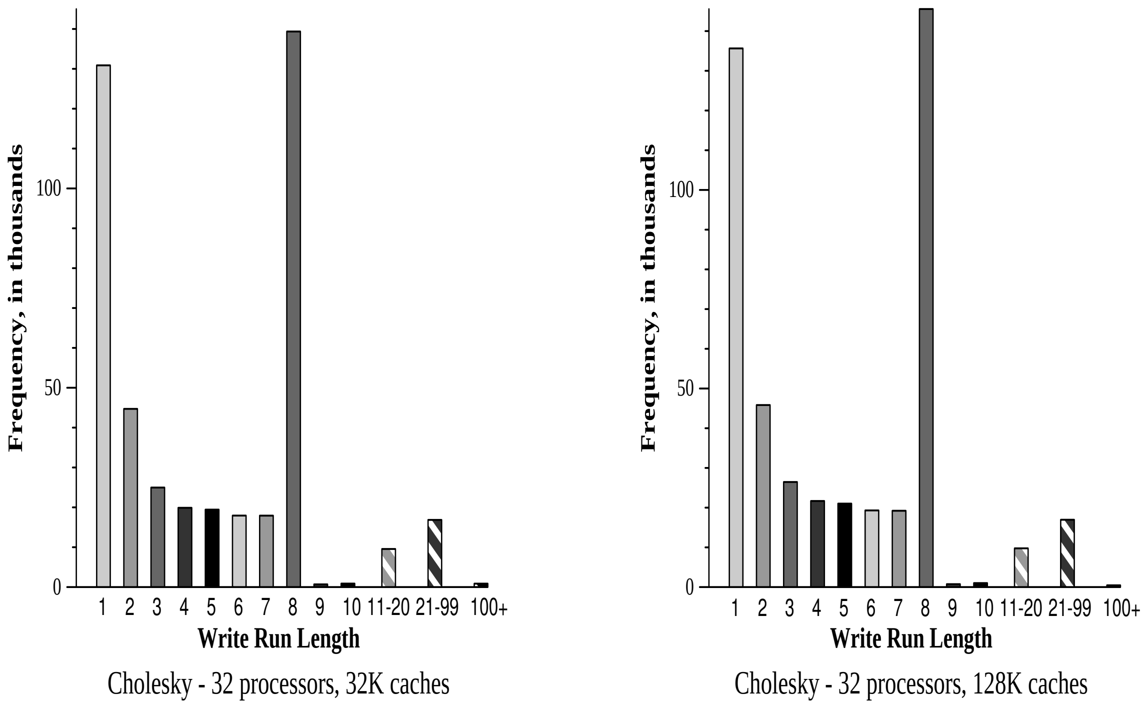


Figure 4.6: Write run lengths for Cholesky, 32 processors

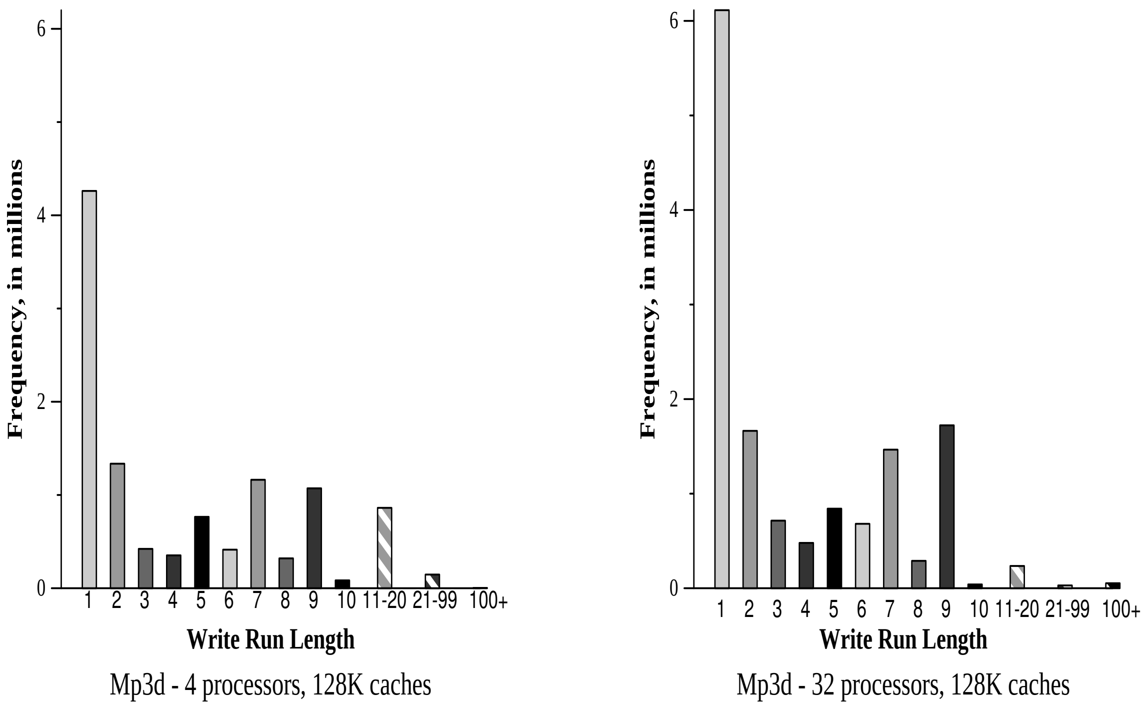


Figure 4.7: Write run lengths for MP3D

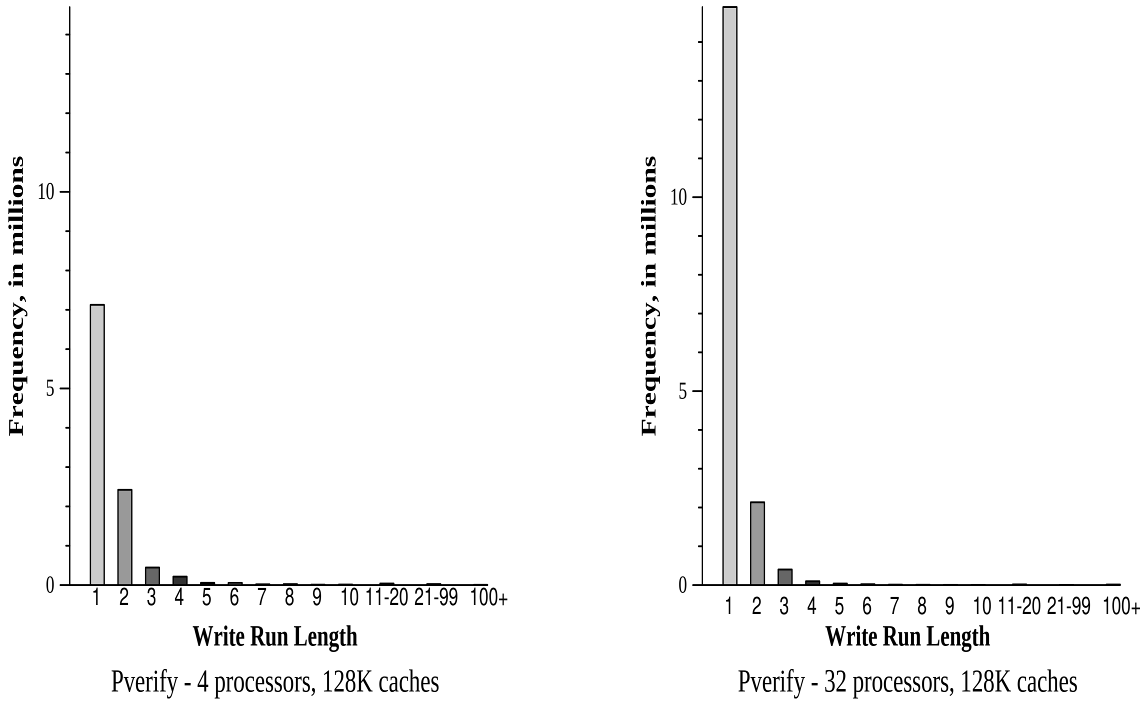


Figure 4.8: Write run lengths for Pverify

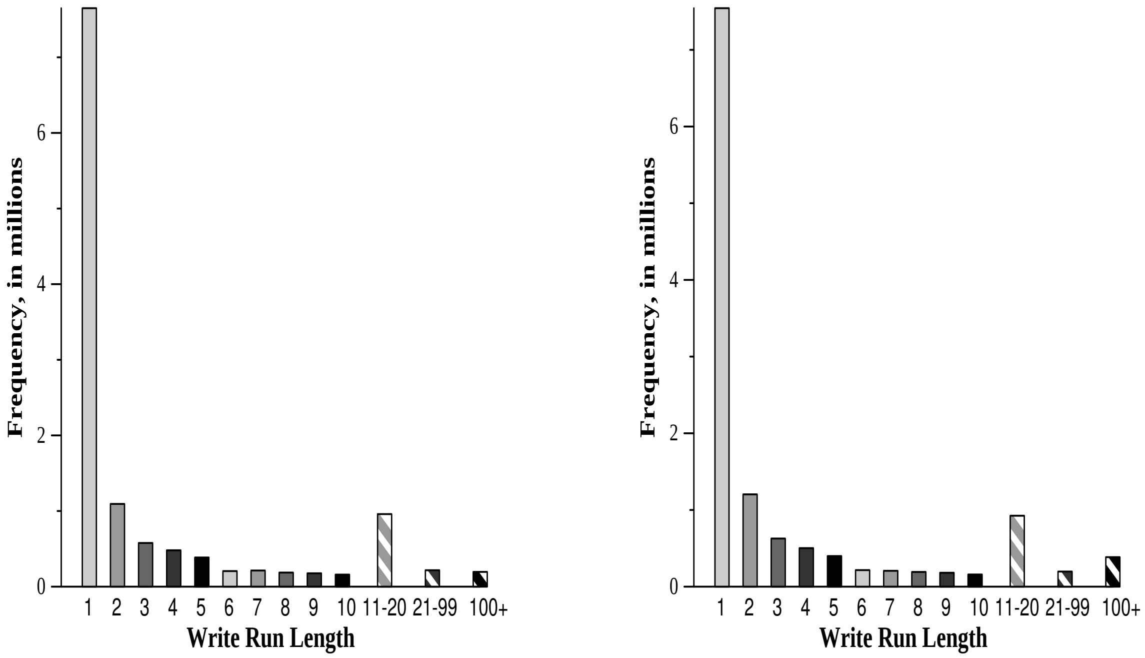
Topopt

Topopt, like Pverify, has a significant amount of true sharing, and false sharing when large block sizes are used. Thus, it is not surprising that when using a large or small number of processors with large or small caches, most write runs are of length 1. However, there are differences between the write run distribution for 4 processors (Figure 4.9) versus 32 processors (Figure 4.10). When 4 processors are used, there are a significant number of run of length greater than 10. There are even a non-trivial number of runs of length greater than 100, especially when 128K caches are used. In contrast, when 32 processors are used, the number of write runs of length 1 jumped dramatically, while the number of long write runs dropped. Still, even with 32K caches and 32 processors, Topopt had over 200,000 write runs of length greater than 10 — 100,000 of them of length greater than 100. When 128K caches are used, over half a million runs were longer than 10, and over 250,000 were of length 100 or greater. Despite these long write runs, we would expect that because of the large number of write runs of length 1, an invalidate protocol would not perform well in the absence of snarfing. An update protocol that did not use a sharing line would lose performance on the substantial number of long write runs in the program.

4.5 Results summary

In this section, we summarize the results that we present in more detail in a later section. We also present graphs detailing the performance of the four protocols using snarfing, 128K caches and 32 processors (Figure 4.11). For comparison purposes, we also ran the Snoopy Reading protocol, which is a less expensive adaptive protocol. For our Snoopy Reading runs, a block was dropped from the cache if 3 updates were received without an intervening access by the processor.

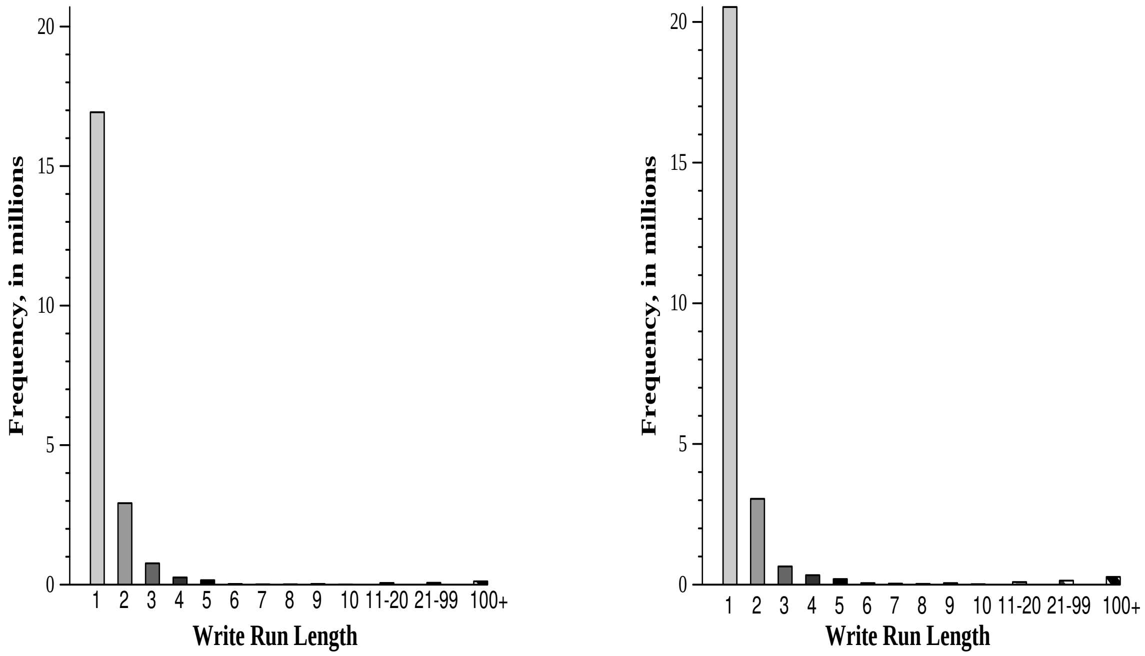
The adaptive protocols, especially Random Walk, performed well compared to WI and WU. Considering first the results with snarfing, both adaptive protocols performed better than either WI or WU on MP3D and Topopt, while Random Walk performed better than the other three protocols on Gauss. On Cholesky and Barnes the adaptive protocols performed as well as the better of the two standard protocols. And on the Badgauss program, the adaptive protocols avoided the poor behavior of WU, and performed slightly better than WI. Only on Pverify did both of the



Topopt - 4 processors, 32K caches

Topopt - 4 processors, 128K caches

Figure 4.9: Write run lengths for Topopt, 4 processors



Topopt - 32 processors, 32K caches

Topopt - 32 processors, 128K caches

Figure 4.10: Write run lengths for Topopt, 32 processors

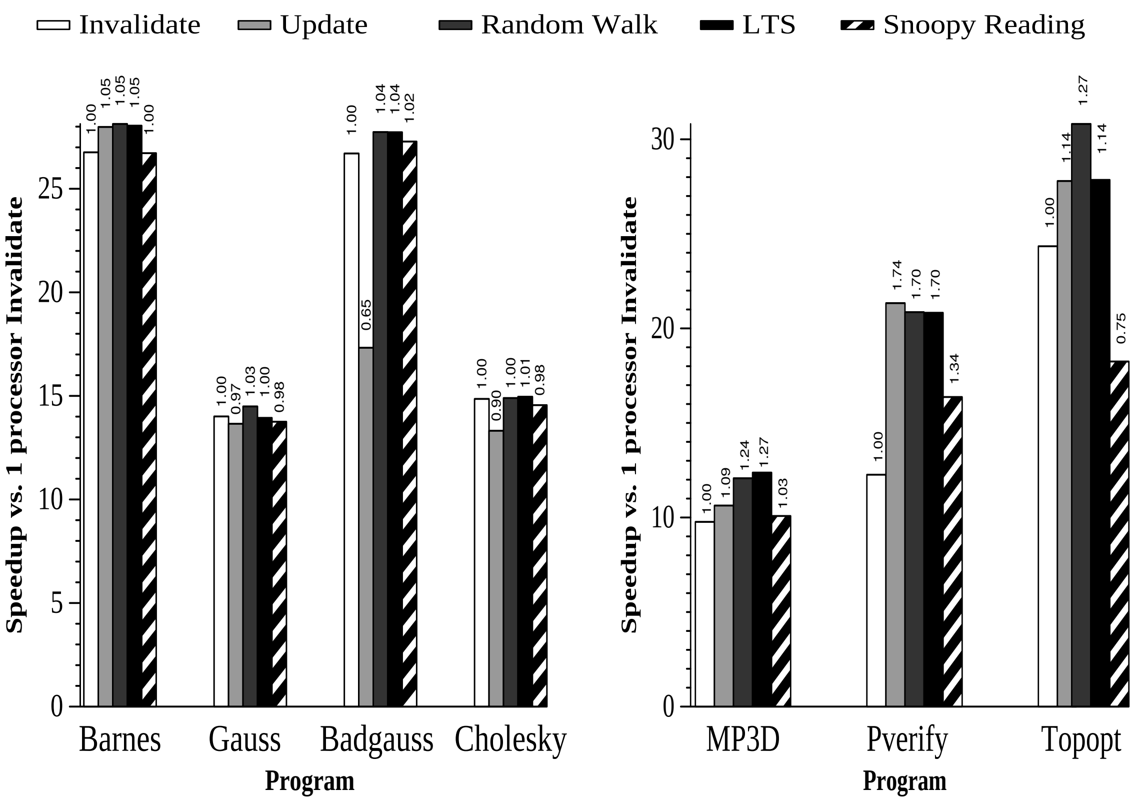


Figure 4.11: Results Summary: 128K Caches, 32 Processors, Snarfing

adaptive protocols perform slightly below that of the best standard protocol (WU). The Snoopy Reading protocol did not do nearly as well as the dynamic adaptive protocols. For those programs where WI works better than WU (Barnes, Gauss, Badgauss, and Cholesky), it generally did about as well as WI. For those applications that did significantly better using WU (Pverify, and Topopt), the Snoopy Reading protocol did relatively poorly. On MP3D, it did slightly better than WI, while the dynamic protocols performed 25% better. It did not come near the performance of the adaptive protocols (or WU) on Pverify, and actually performed worse than all protocols on Topopt.

Without snarfing, Random Walk still performed as well or better than the standard protocols on nearly every benchmark and configuration, while LTS often did not perform nearly as well. The reason why LTS often did not work as well as RW is that it often decided to switch to using invalidate on a block sooner than RW. Thus it was on those applications where WU worked better than WI that LTS often did not perform as well as RW. Random Walk outperformed all others on Topopt, and did as well (or a bit better than) as the better of WI or WU on Barnes, Gauss, Cholesky, and MP3D. Only on some configurations of Pverify did RW not do as well as WU.

On the surface, it may be surprising that WU did well on so many of the selected benchmarks. One reason is that the performance of WU degrades less than WI in the presense of false sharing, since only those portions of the block which are written are transmitted over the bus, rather than the entire block which is the case with WI. In addition, our simulation assumptions favored WU over WI, since we assumed both a large block size and a fast bus relative to the speed of the processors. Since WU usually required more bus transactions than WI, increasing the cost of initiating a bus transaction would affect WU more than WI. In addition, in many of the benchmarks the application's working set did not fit in cache, which caused blocks that were not being actively used in a given cache to be replaced. This tended to increase the number of blocks that are found in only one cache, i.e. blocks that may be written without an update transaction. In systems with large (second-level) caches, replacement activity may be minimal or non-existent. In this case, a block which is written exclusively by a single processor, but was referenced at one time by another processor (perhaps in the algorithm's initialization phase) would still have updates sent for every write. Note that the Badgauss benchmark was constructed to exhibit this kind of behavior. Applications that exhibit this sort of behavior perform poorly under WU, but the adaptive protocols handle the situation well by recognizing the long write run(s) and invalidating the block immediately.

One remaining issue is whether the cost of the adaptive schemes is justified by the increase in performance gained by using them. Given that LTS has a higher hardware cost than RW, and in most cases did not perform better than RW, it is hard to justify using LTS over RW. In our test programs, RW did not significantly outperform the better of WU or WI, except for certain instances of MP3D and Topopt. In most of the other cases, RW did as well or slightly better than the best choice of WI or WU for a particular application. In the final analysis, one needs to weigh the benefit of using an adaptive protocol and obtaining performance equal to the better of WI and WU versus the additional complexity and hardware needed by the adaptive protocols.

4.6 Performance with Snarfing

Badgauss

In this section, we show the results for our artificial benchmark on which WU performs poorly. We used 512K caches so that the entire array would remain cached in all processors, forcing WU to do an update transaction on every write. As expected, WU did poorly on Badgauss, though the effect of the many update transactions did not affect performance much until 32 processors were used (Figure 4.12). The reason for this is that a relatively high amount of computation is done for every store in the program. Thus it is only when the cost of an update became high (due to contention) that the additional bus load caused by the large number of updates became a factor. WU used 40–60 times more bus transactions than invalidate, and transferred 8–13 times the data (Figure 4.13). The important point about these results is that the adaptive hybrid strategies avoided the poor behavior of WU by immediately invalidating when writing to blocks that make up the array. The adaptive strategies invalidated immediately because the initial invalidate threshold was 0. In addition, write runs were (much) longer than the invalidate ratio, thus invalidate threshold did not (in general) increase above 0.

Barnes

Because of the low degree of sharing in Barnes, varying the coherence protocol had a relatively small impact on performance. Since the speedup was close to linear in all cases, the differences between the four protocols were necessarily minimal. Using the update protocol or the two dynamic schemes with 32K caches increased performance over invalidate by 1% using 16 processors and by 5% using 32 processors (cf. Figure 4.14). The improvements were the same for the 128K cache configuration.

The differences between the four protocols is more pronounced when other performance metrics are considered. The invalidate protocol had much higher miss rates, both for reads and writes, than the other three protocols (see Figure B.1). For all configurations, the WU protocol had the lowest miss rates, followed by the Random Walk and the LTS protocols. The reductions in miss rates were substantial; for example, for the 16 processor 32K cache configuration, using WU cut the read miss rate by 92%, while the LTS protocol cut the miss rate by 78%. In addition, the

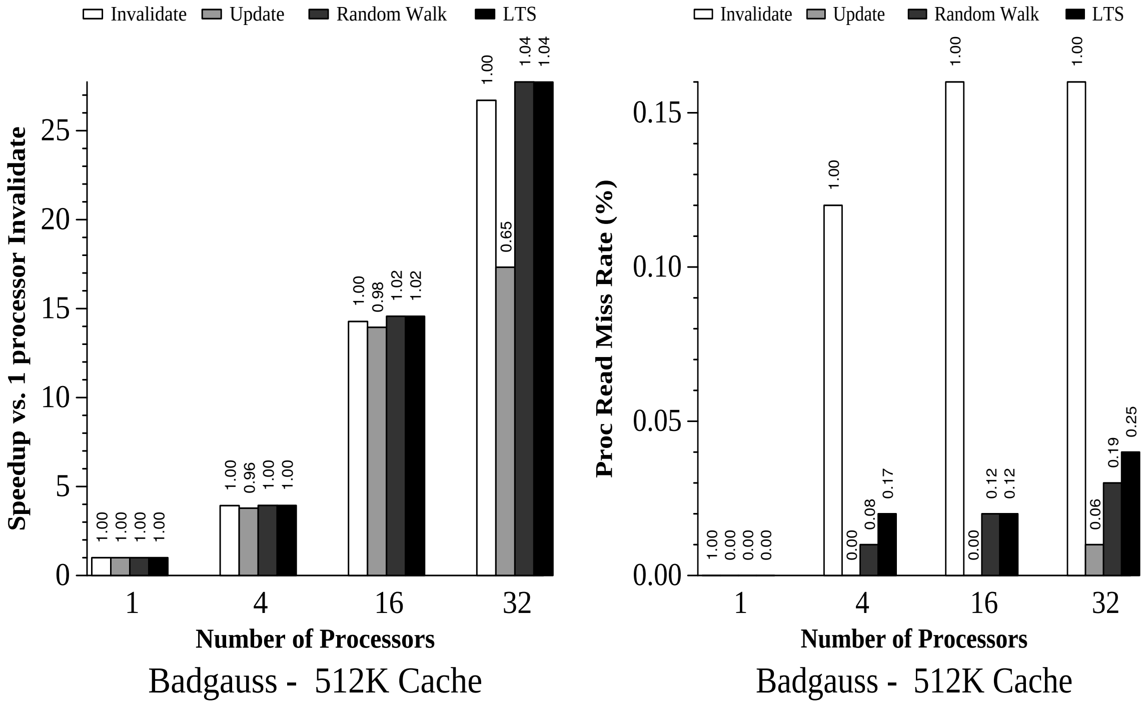


Figure 4.12: Speedups(L) and Read Misses(R) for Badgauss

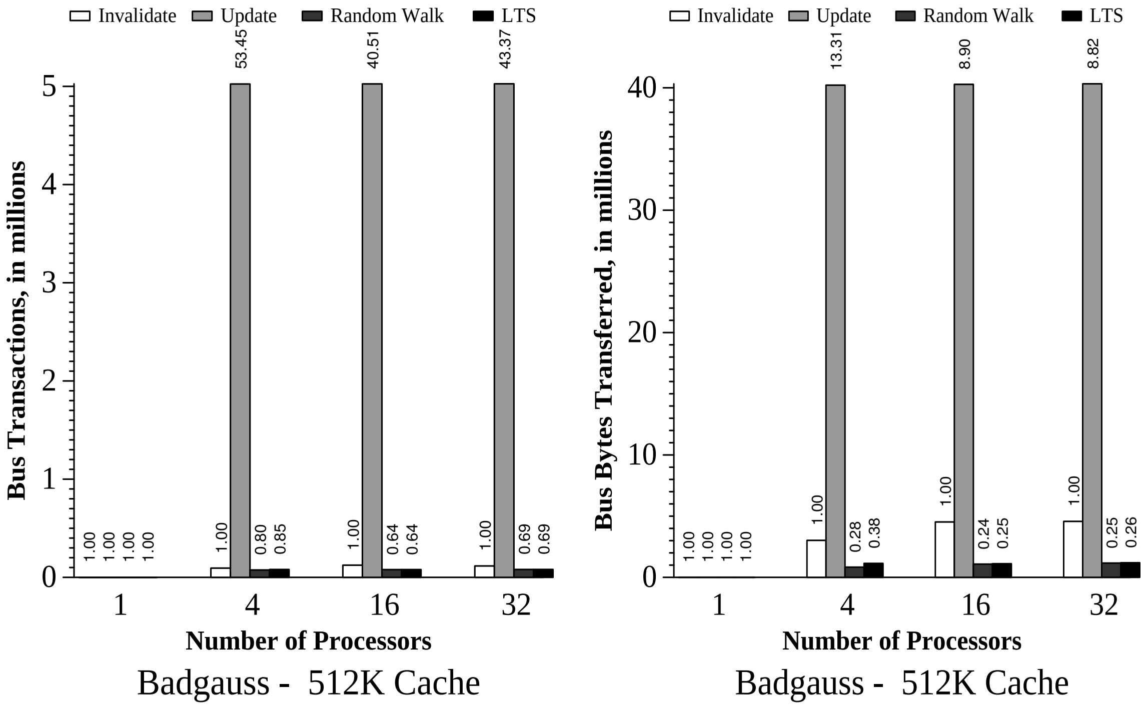


Figure 4.13: Bus transactions(L) and Bytes Transferred(R) for Badgauss

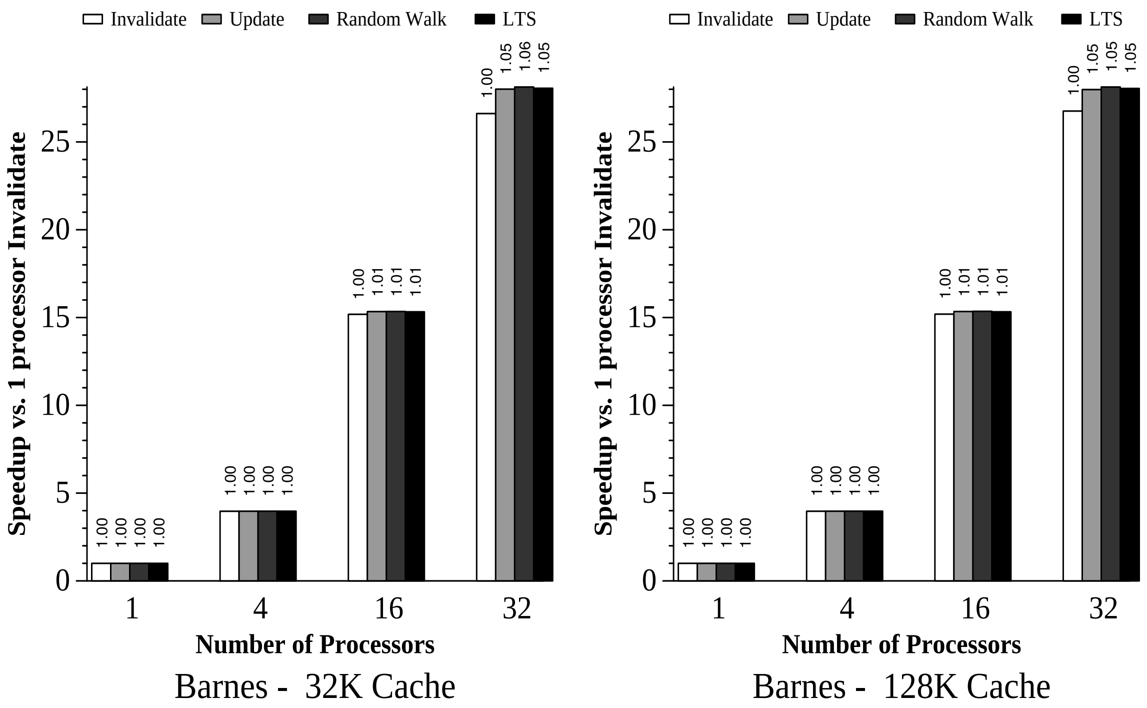


Figure 4.14: Speedups for Barnes (with snarfing)

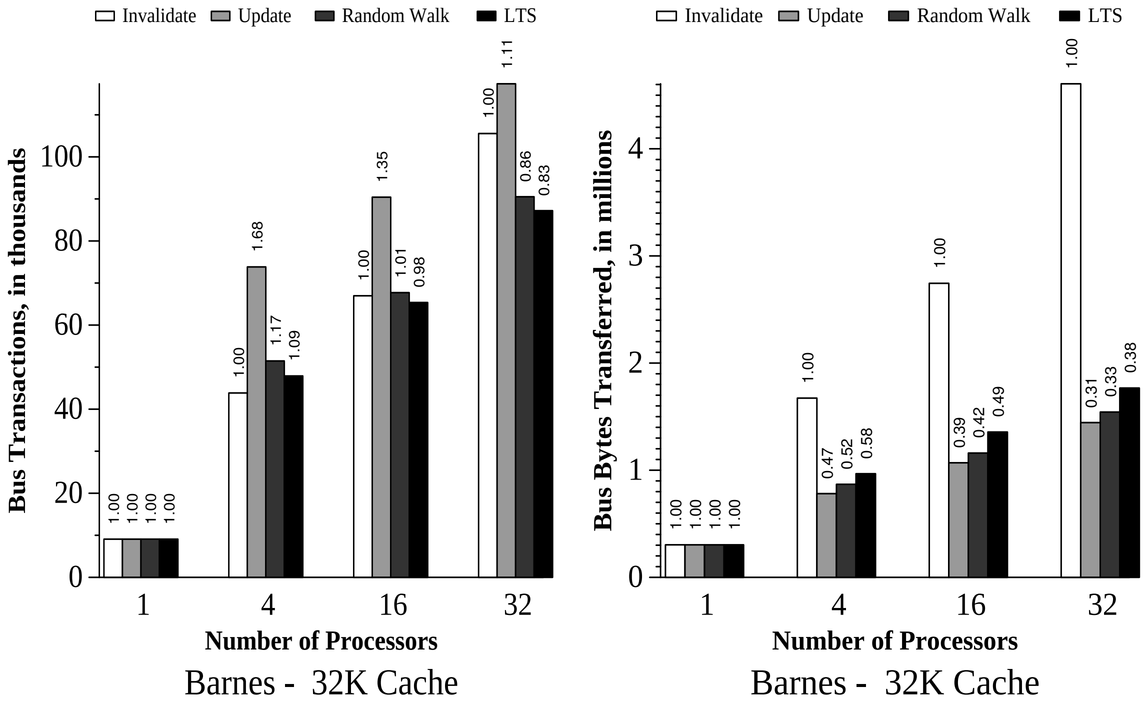


Figure 4.15: Bus transactions (L) and Bytes Transferred for Barnes (with snarfing)

non-invalidate protocols reduced write miss rates by a comparable amount.

Although using an update protocol did reduce miss rates, it often caused an increase in the number of bus transactions due to additional update transactions. This was the case for the WU protocol; for all configurations, it required more bus transactions than the WI protocol (Figure 4.15). The increase was greatest when 4 processors were used — 68% when 32K caches were used, and 80% with 128K caches. When 32 processors were used, the increase was much less — 11% and 17% for 32K and 128K caches respectively. In contrast with the WU protocol, the two hybrid protocols didn't suffer a large increase in the number of transactions. For 4 processors, RW used about 20% more transactions, while LTS used about 9% additional transactions. When 32 processors were used, both adaptive protocols used *fewer* transactions even than WI. The decrease ranged from 11% for RW and 128K caches to 17% for LTS and 32K caches. Because the dynamic adaptive strategies were able to adjust their coherence strategy on a block-by-block basis, they were able to obtain most of the reduction in miss rates obtained by the WU protocol without a large increase in the number of bus transactions.

While the three non-WI protocols sometimes increased the number of bus transactions, in all cases they significantly reduced the number of bytes transferred over WI. The greatest reductions occurred when using WU, but the two adaptive strategies were not far behind. Reductions ranged from 42% for the 4 processor, 32K cache LTS run to 72% for WU when using 128K caches and 32 processors. It is these reductions, along with the reductions in miss rate that are responsible for the performance increases of the three non-WI protocols. The large decreases in miss rates and bytes transferred did not lead to large increases in performance because bus contention did not hinder program performance even when the WI protocol was used; even in the worst WI configuration, bus utilization did not exceed 15%. Despite the lack of large performance increases for the adaptive schemes, both reduced read and write miss rates and the number of bytes transferred without the large increases in the number of bus transactions that occurred with the WU protocol.

Gauss

Overall, all four protocols (WI, WU, RW, and LTS) performed roughly the same on the Gauss benchmark. The poor overall speedup of Gauss on 32 processors was due to bus saturation, chiefly caused by reads and writebacks due to conflict misses. Increasing the bandwidth to memory by doubling the bus width would improve the performance of all schemes (cf. 3.

While nearly all write runs were long enough to favor WI over WU, WU avoided doing an update for every write to the matrix. Because rows are statically allocated to processors, a given row in the matrix was usually present only in the cache of the processor that was modifying it. Because the sharing line was not raised on writes to row elements, the WU protocol avoided sending out an update for every matrix element written.

For the 32K cache case, both WU and RW performed about 3% better than WI for 16 and 32 processors (Figure 4.16). The chief reason for this is the slightly lower read miss rate (by 5%–10%) for WU and RW (Figure B.3). Both the array of flags used to synchronize successive iterations of the solver loop, as well as accesses to the b vector benefited from updating, since both are extensively write shared. The use of updating led to small decreases in both the number of bus transactions and the number of bytes transferred (Figure 4.17). Last Three Samples performed about the same as WI, though it did manage very small reductions (1–3%) in the number of bus transactions and the number of bytes transferred, and somewhat larger reductions in the read miss rate (5–7%) in the miss rate.

When larger caches were used (128K), WU performed slightly worse (2%) than WI, while RW performed slightly better (3%) than WI (Figure 4.16). Though WU and RW both had somewhat lower miss rates (by 8% to 15%), the gains caused by lower miss rates of the WU protocol did not offset its larger number of transactions versus the WI protocol, as seen in Figure B.4. The adaptive protocols, on the other hand, had 1%–2% fewer transactions than even the WI protocol, except for LTS when using 32 processors. Last Three Samples again performed about the same as WI overall, but had slightly lower read miss rates (5–8%) than WI as well as small reductions in bus transactions (0–2%) and bytes transferred (2–3%).

Cholesky

Because of the number of long write runs, the WI protocol performed better than WU on Cholesky (Figure 4.18). The two adaptive protocols performed about the same as WI, varying in performance no more than 1% for all configurations. WU performed

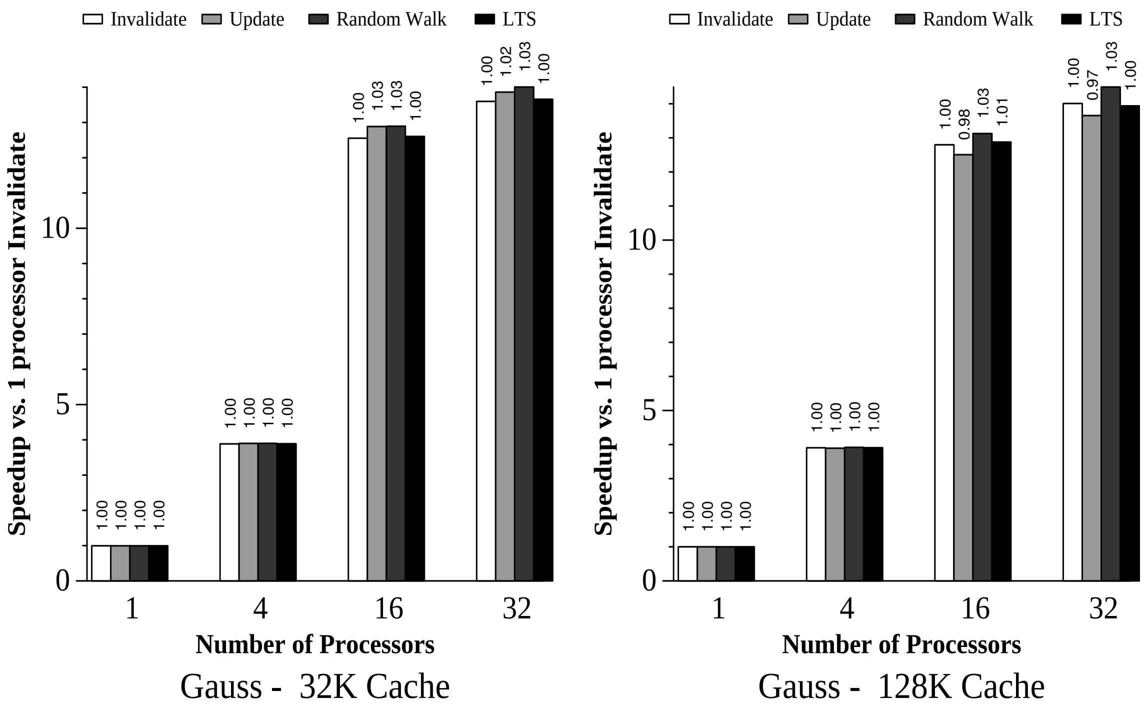


Figure 4.16: Speedups for Gauss (with snarfing)

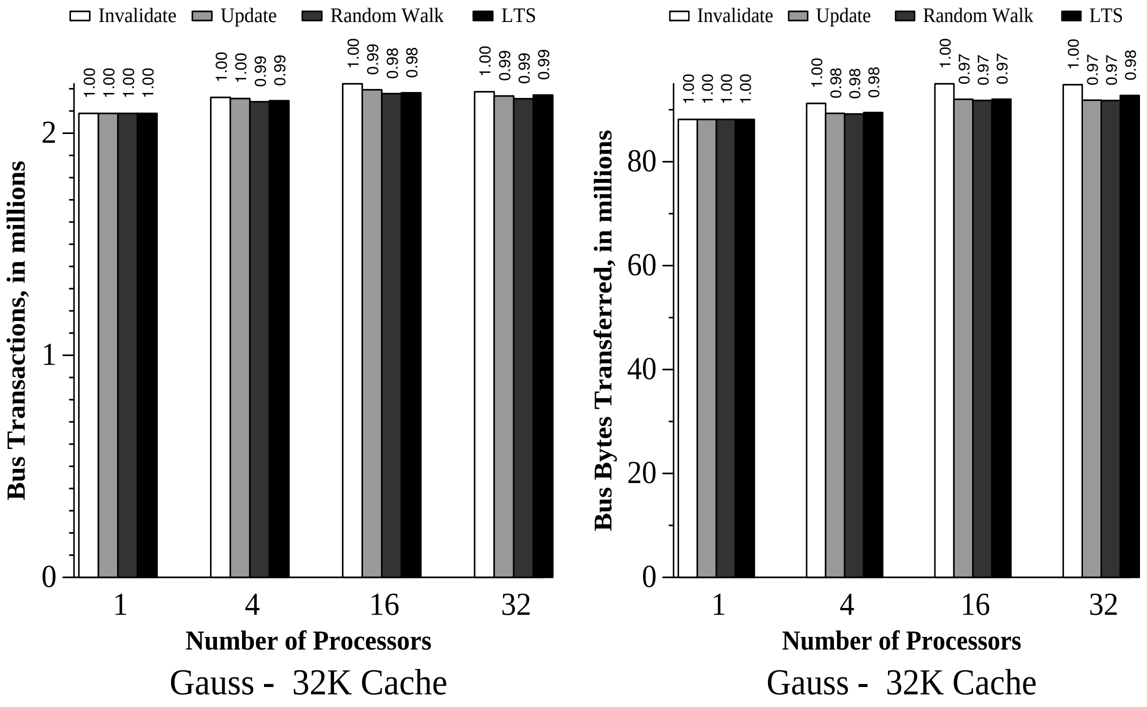


Figure 4.17: Bus transactions and Bytes Transferred for Gauss (with snarfing)

the worst of the 4 protocols. For both cache sizes, WU performed 5% worse using 16 processors, and about 10% worse using 32 processors. These performance decrease occurred despite the reductions in both read and write miss rates under WU. WU reduced read misses for the 32K configuration by about 12% for 4 and 16 processors, and 50% for 32 processors (Figure B.5). The reductions were even greater when 128K caches were used. The adaptive protocols also reduced the read and write miss rates over WI — though in most cases not as much as WU. The exceptions were for 32K caches and both 16 and 32 processors, where RW did better than WU.

The reason for the performance decline under WU can be seen in the graphs of the number of bus transactions and the number of bytes transferred (Figure 4.19). The WU protocol consistently used a great deal of additional bus transactions as compared to the WI protocol. When 32K caches were used, the increase ranged from 1.7 times as many transactions for 4 processors to 2.9 times for 32 processors. The increase was even worse for 128K caches, where the increase ranged from 3.2 to 3.6 times that of WI (Figure B.6). The number of transactions for WU did not increase by very much when going from 32K to 128K caches, but the number of transactions for WI decreased when the cache size was increased, leading to the greater disparity in the number of bus transactions. The increase in the number of bytes transferred was not as severe — it ranged from 11% to 33% for 32K caches and from 34% to 45% for 128K. The additional load placed on the bus by the many update transactions lead to the decreased performance of WU when compared to WI. The performance impact of using WU versus WI would have been even greater if the overall bus utilization of the application had been greater.

Using the adaptive protocols also increased bus transactions for the adaptive protocols, but not to the degree that WU did. The increase in bus transactions ranged from 5–24% for 32K caches, and from 14% to 33% for 128K caches. The increase of RW was in general slightly higher than that of LTS. Except for the 4 processor, 128K cache case, the adaptive protocols reduced the number of bytes transferred below even that of WI. The decrease ranged from 1%–5% for 4 and 16 processors to about 10% for 32 processors.

Overall, the adaptive protocols did fairly well on Cholesky. They performed about as well as the better of the two standard protocols (WI), while decreasing the miss rate without a huge increase in the number of bus transactions, and in most cases

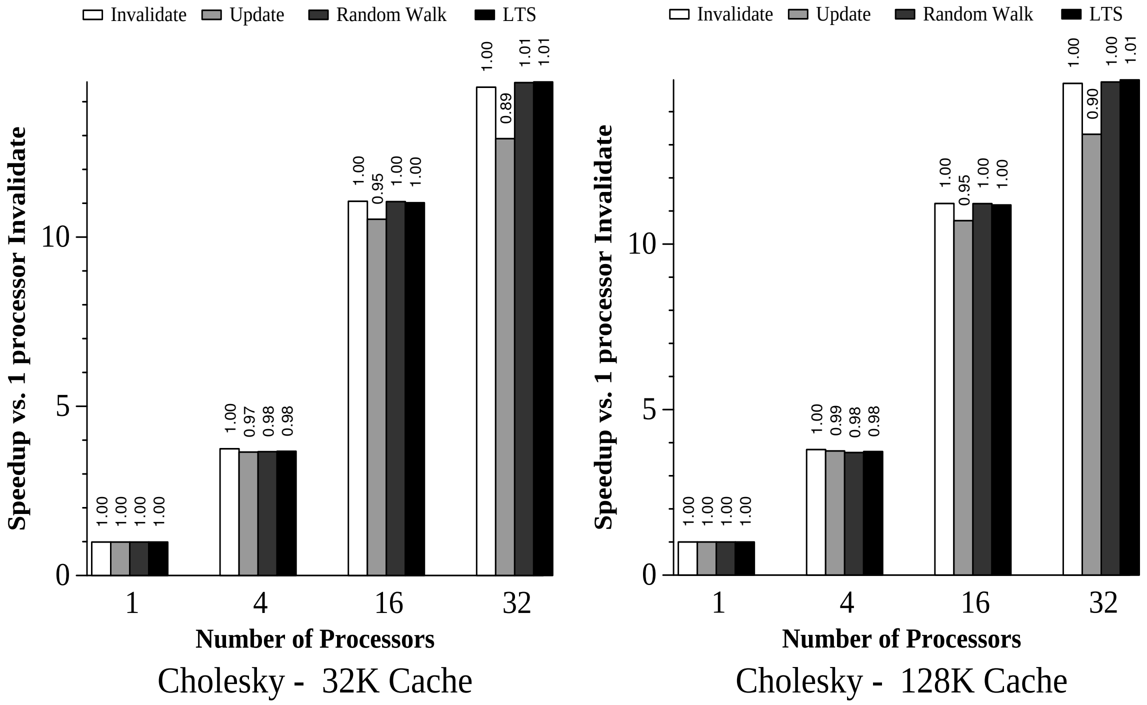


Figure 4.18: Speedups for Cholesky (with snarfing)

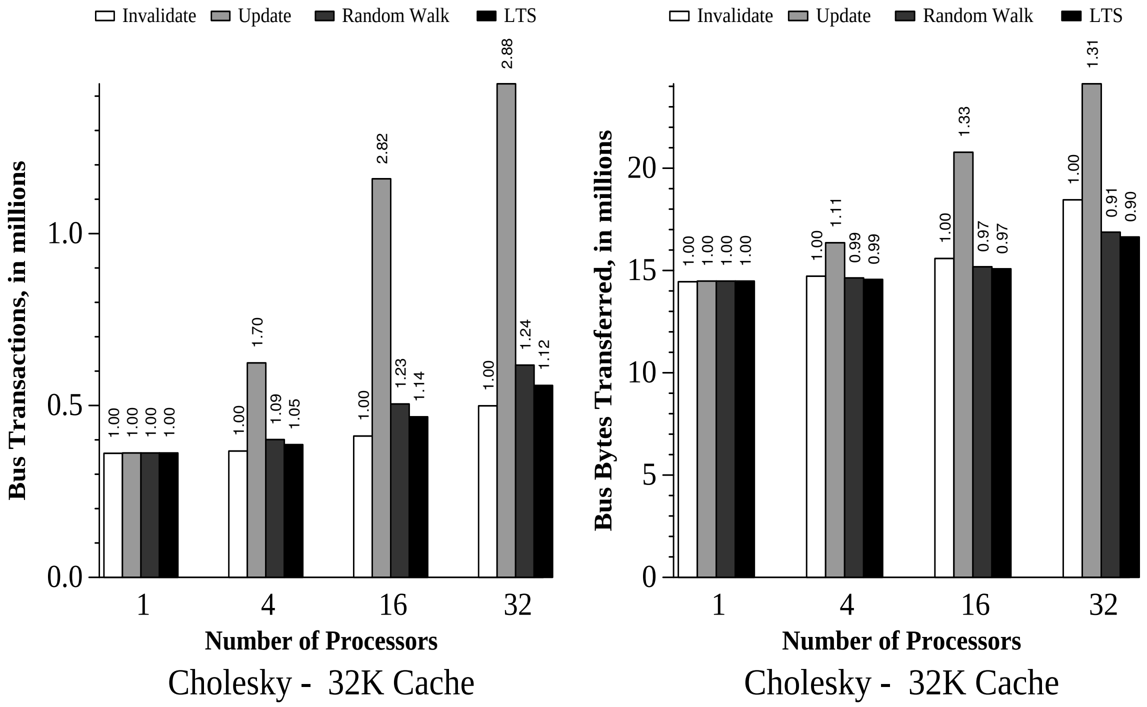


Figure 4.19: Bus transactions and Bytes Transferred for Cholesky (with snarfing)

transferring less data than the WI protocol.

MP3D

Because of the large amount of sharing in MP3D, it had poor speedup on all four protocols. When using 32 processors with 128K caches, speedup was less than 13 even with the best choice of protocol. Still, there were performance differences between the 4 protocols for both sizes of processor cache tested.

When 32K caches were used, both of the adaptive protocols performed better than either WI or WU (Figure 4.20). Last Three Samples performed the best using 16 processors with an 18% performance increase over WI. It also outperformed the other 3 protocols when using 32 processors with an 20% improvement over WI. Random Walk was just behind LTS in performance with improvements of 14% and 16% for 16 and 32 processors respectively. These improvements are due to the reduction in bus use by the adaptive protocols. Though the WU protocol did not perform as well as the adaptive protocols, it had the lowest read (cf. Figure B.7) and write miss rates of any of the 4 protocols. While the adaptive protocols reduced the miss rate by about 31–38% for 4, 16 and 32 processors, the WU protocol reduced the miss rate by about 57%. The problem with WU was that it used a great deal more bus transactions than either WI or the adaptive protocols, which drove up bus traffic (Figure 4.21). Using Write Update increased the number of transactions over WI by 82% for 4 processors to 91% for 32 processors. The increase in transactions was far smaller for the adaptive protocols. While WU had far more bus transactions than the other 3 protocols, it transferred fewer bytes than the other protocols (Figure 4.21). The difference between WU and the adaptive protocols was small (3 percentage points) when 4 processors were used, but grew larger as more processors were used. Despite the fact that WU transferred fewer bytes than the adaptive protocols, the large number of bus transactions it required lowered its performance below that of the adaptive protocols.

Like the case when using 32K processors, the adaptive protocols outperformed both WI and WU when using 128K caches (Figure 4.20). Last Three Samples again outperformed all others with a 21% improvement over WI using 16 processors and a 27% improvement when using 32 processors. Random Walk was only slightly slower than LTS, with WU behind with a 10% performance increase over WI at 16 and 32 processors. Write Update decreased the read miss rate over WI even more with

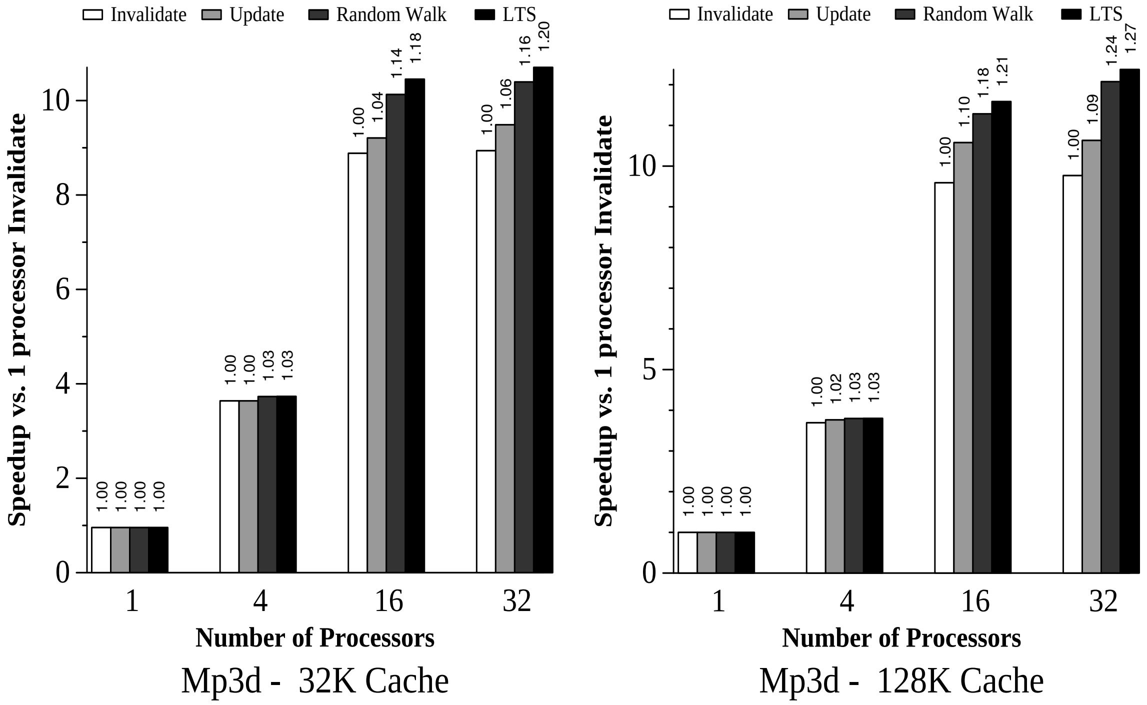


Figure 4.20: Speedups for MP3D (with snarfing)

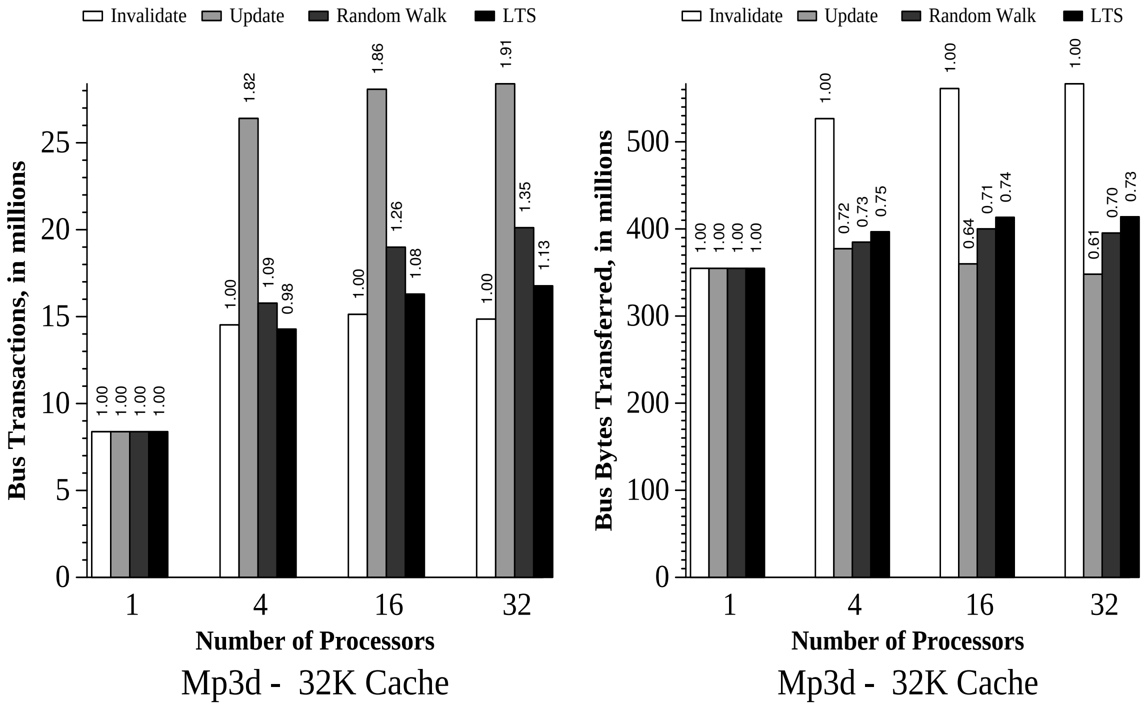


Figure 4.21: Bus transactions and Bytes transferred for MP3D (with snarfing)

128K caches, eliminating nearly 80% of the WI misses (Figure B.7). The adaptive protocols eliminated about 40% of misses, yet they did so without the dramatic rise in bus transactions that occurred when using WU. Write Update used about twice the bus transactions that WI did, while the adaptive protocols used about 10–38% more bus transactions (Figure B.8). Write Update did transfer about half the bytes that WI did, but the reductions achieved by the adaptive protocols (29–38%) were not far behind.

Both adaptive protocols did well on MP3D. They selectively used an update policy where needed, reducing the miss rate and the number of bytes transferred without drastically increasing the number of bus transactions.

Pverify

As discussed in Section 2.3, Pverify has a great deal of true and false sharing, which led to a large number of very short write runs. Therefore, we would expect that an update strategy would work better than invalidate.

When 32K caches were used, much of the traffic on the bus was due to reads and writebacks due to capacity and conflict misses. Because of this, using WU or an adaptive scheme over WI did not dramatically increase the poor speedup of the application (Figure 4.22). For 16 processors, using RW increased performance by 19%, with WU and RW close behind at 18% and 17% respectively. For 32 processors, WU was best with a 19% improvement, while RW and LTS both performed about 15% better. The improvements were entirely due to the reduction in bus traffic, since the bus was completely saturated when 16 or 32 processors are used. The three non-invalidate schemes reduced the read miss rate by about 20% for all configurations (Figure B.9). The write miss rate was cut by all three by more than 80%. The reduction in misses helped to reduce the number of bus transactions by 7–10% and the number of bytes transferred by 12–19% (Figure 4.23).

The differences between using WI and WU or an adaptive strategy were much larger when 128K caches are used. Write Update, RW, and LTS increased performance by 20% when 16 processors were used. For 32 processors, using WU increased performance by 74%, with RW and LTS behind at 70%. Read misses were cut by about 60% for 4 and 16 processors using WU and RW, and slightly less for LTS (Figure B.9). All three protocols cut misses by about 50% when 32 processors were

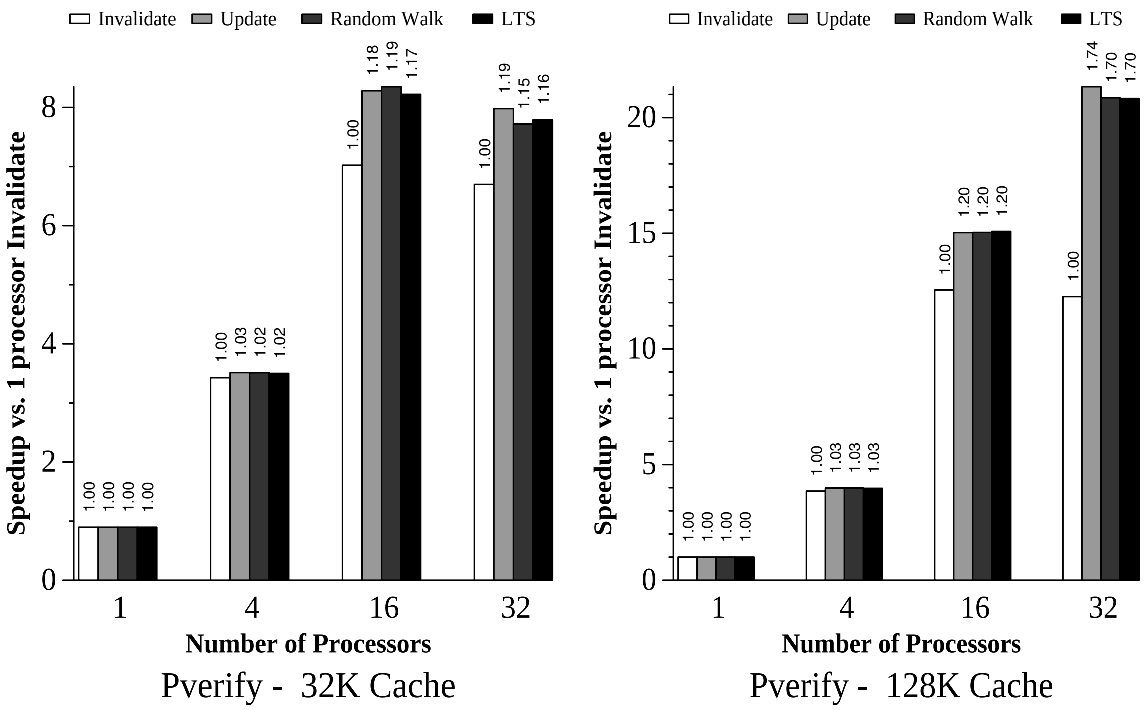


Figure 4.22: Speedups for Pverify (with snarfing)

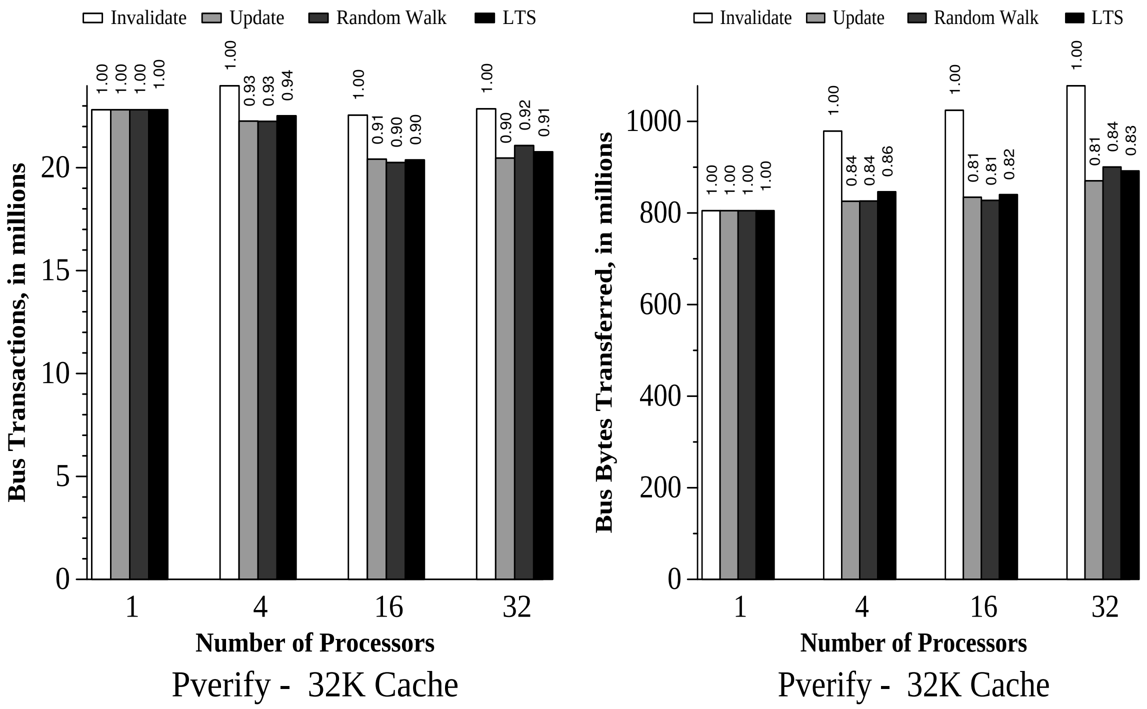


Figure 4.23: Bus transactions and Bytes Transferred for Pverify (with snarfing)

used. Write misses were also reduced by about 88% for LTS, and by 95% or more for RW and WU. Because of the reduced number of misses, bus transactions were cut by about 20% for all configurations. The reduction in number of bus transactions, plus the fact that updates only transferred 8 bytes, while re-reads due to invalidates transferred 64 bytes (most of which were usually not used in Pverify) resulted in a drop in the number of bytes transferred by 40–50% for all three non-invalidate protocols over WI.

These results are not surprising, given the poor behavior of Pverify when using large block sizes [AB95]. While false sharing hurts performance under both WI and WU, it much worse under WI, since the entire cache block is "ping-ponging" between the false sharing caches, rather than just the values actually being written. The adaptive protocols performed quite well, nearly matching WU's performance. Because nearly all of Pverify's write runs were short, the adaptive protocols used the update strategy nearly all the time.

Topopt

As discussed earlier, Topopt exhibits many short write runs due to true and false sharing. It also does a great deal of barrier synchronization, which adversely affects performance, especially when read snarfing is not used. Thus it is not surprising that the non-invalidate strategies performed well. However, despite the many short write runs, WI with snarfing also did relatively well on Topopt, because as discussed above, Topopt also has a significant number of longer write runs.

When 32K processor caches were used, WI did well, achieving a speedup of about 19 when using 32 processors (Figure 4.24). WU did better than WI, performing 13% better than WI at 16 processors, and 10% better at 32 processors. The two adaptive schemes also did well using 16 processors. Random Walk improved over WI by 14%, while LTS's improvement was 11%. The speedups were similar at 32 processors, where Random Walk improved speedup by 12%, with the other two close behind at 10%. The three non-invalidate schemes cut the number of read misses by about 45% for 4 processors to over 84% for 32 processors (see Figure B.11). Write misses were reduced by about 40% for all configurations. One difference between WU and the adaptive strategies can be seen in Figure 4.25. While WU used more transactions than WI for 4 and 16 processors, the two adaptive schemes used fewer transactions

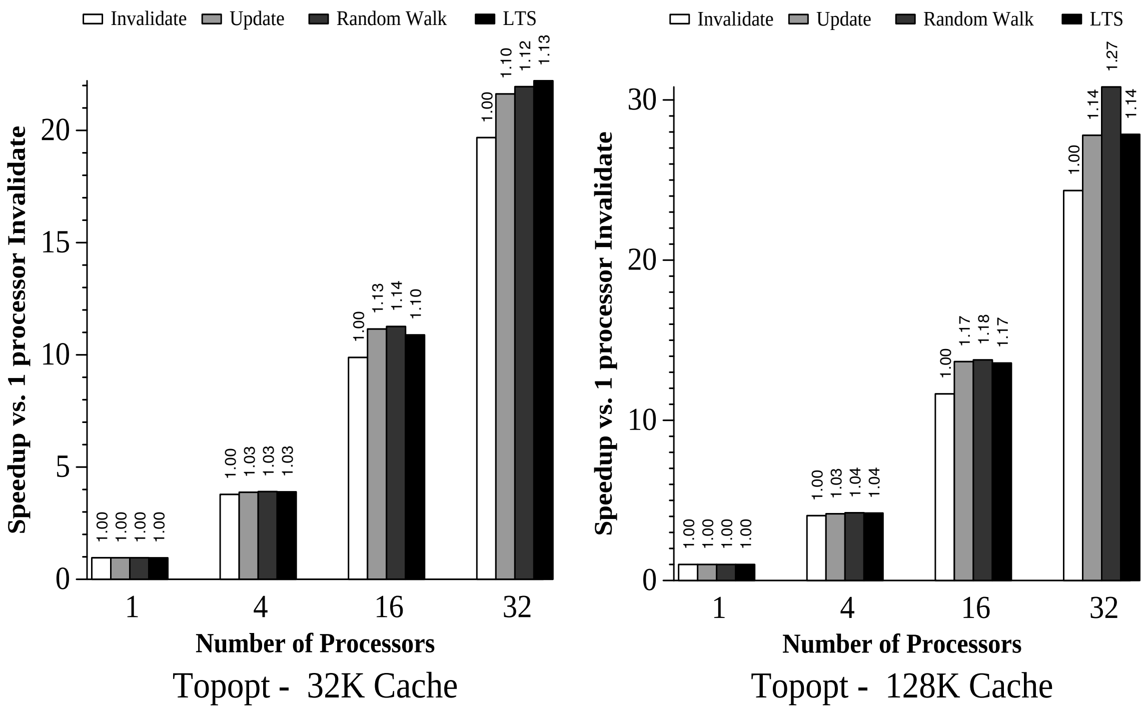


Figure 4.24: Speedups for Topopt (with snarfing)

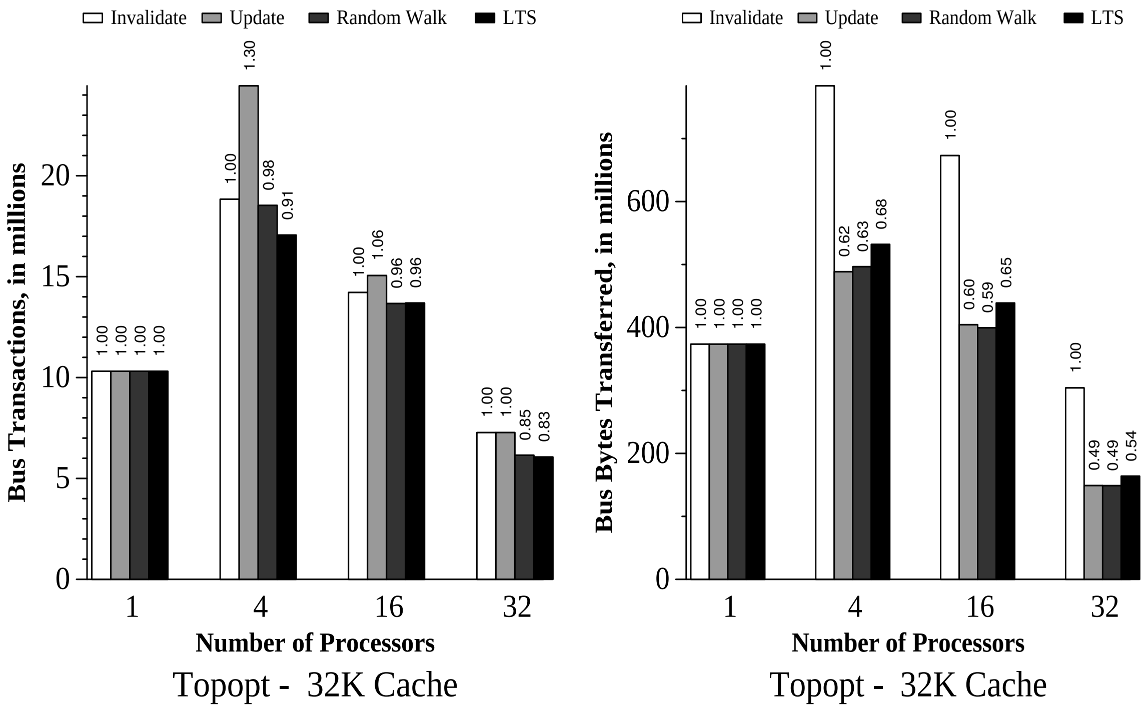


Figure 4.25: Bus transactions and Bytes Transferred for Topopt (with snarfing)

than WI for 4, 16, and 32 processors. The three non-invalidate schemes reduced the number of bytes transferred over WI by 15–20%.

The adaptive protocols performed even better when the large cache size was used. Using 16 processors, Random Walk performed 18% better than WI, and both LTS and WU performed 17% better than WI. Random Walk performed better than either WU or WI when 32 processors were used. Random Walk improved speedup over WI by 27%, while LTS and WU had a 14% improvement. Write Update had the lowest read miss rate of any of the four protocols — it was over 95% less than that of WI for the three multiprocessor runs. Random Walk matched WU's read miss performance at 16 and 32 processors. LTS's miss rate was about 7 times higher than that of WU, but still was much reduced over that of WI. The difference between WU and the adaptive protocols is evident when one looks at the number of bus transactions (Figure B.12). While WU used more bus transactions than WI for 4 and 16 processors, the adaptive protocols always used fewer transactions than WI. The non-invalidate protocols cut the number of bytes transferred by about 70–85% over WI, with the biggest reductions achieved by RW and WU.

The two adaptive protocols performed well on Topopt. Random Walk performed better than the two standard protocols on overall speedup. It achieved the miss reductions of WU without increasing the number of bus transactions as WU did. Though LTS did not perform as well as RW, it did do as well as WU using 32K caches, and slightly better when using 128K caches.

4.7 Results without snarfing

We now turn to the results of our simulation runs without snarfing. In the following sections we compare the results of our simulations without snarfing to the results with snarfing given above. Graphs will be shown only for those cases where the benchmark's performance differed significantly from when snarfing was not used.

Badgauss

As will be discussed shortly, turning off snarfing had little effect on the performance of Gauss. Since Badgauss was derived from Gauss, we expect that the lack of snarfing would not affect Badgauss's results.

Barnes

Turning read snarfing off did not affect overall performance of any protocol when 16 or fewer processors were used, but did have a significant negative effect on the performance of WI when 32 processors were used (Figure 4.26). The performance of RW and WU were not affected by the lack of snarfing; however LTS's performance was affected, since it used invalidation more often than either RW or LTS. Like the case with snarfing, WU had the lowest read miss rate of all four protocols (Figure B.13). The absence of snarfing caused the miss rates for the adaptive protocols to increase, especially at 16 and 32 processors. As seen in Figure 4.27, the increased number of misses caused the adaptive protocols to use more bus transactions than WU, except when 4 processors were used. And like the case with read snarfing, WU transferred the fewest number of bytes among the four protocols.

Because WU had a lower miss rate, and fewer bus transactions and bytes transferred than the other the adaptive protocols, one might expect that its overall performance would be better than that of the adaptive protocols. However, since bus utilization was low for the non-invalidate protocols, the increased bus traffic of the adaptive protocols did not have a large effect on performance.

Gauss

Removing snarfing had little effect on the overall execution time of any of the four protocols. This is not surprising, since there is little active sharing in the application, thus there were few opportunities for snarfing to make a difference. The number of bus transactions and bytes transferred did increase slightly for 16 and 32 processors for the invalidate protocol versus the snarfing case, but it was not enough to affect the results to a large degree.

Cholesky

Turning snarfing off had little effect on the overall performance of any of the four protocols. The performance of WU did increase relative to the other 3 protocols to the point where it performed only 2–5% worse than WI, rather than 5–10% (Figure 4.28). Read miss rates were not markedly affected by the lack of snarfing, but the use of a higher invalidate ratio with the adaptive protocols caused them to use the update

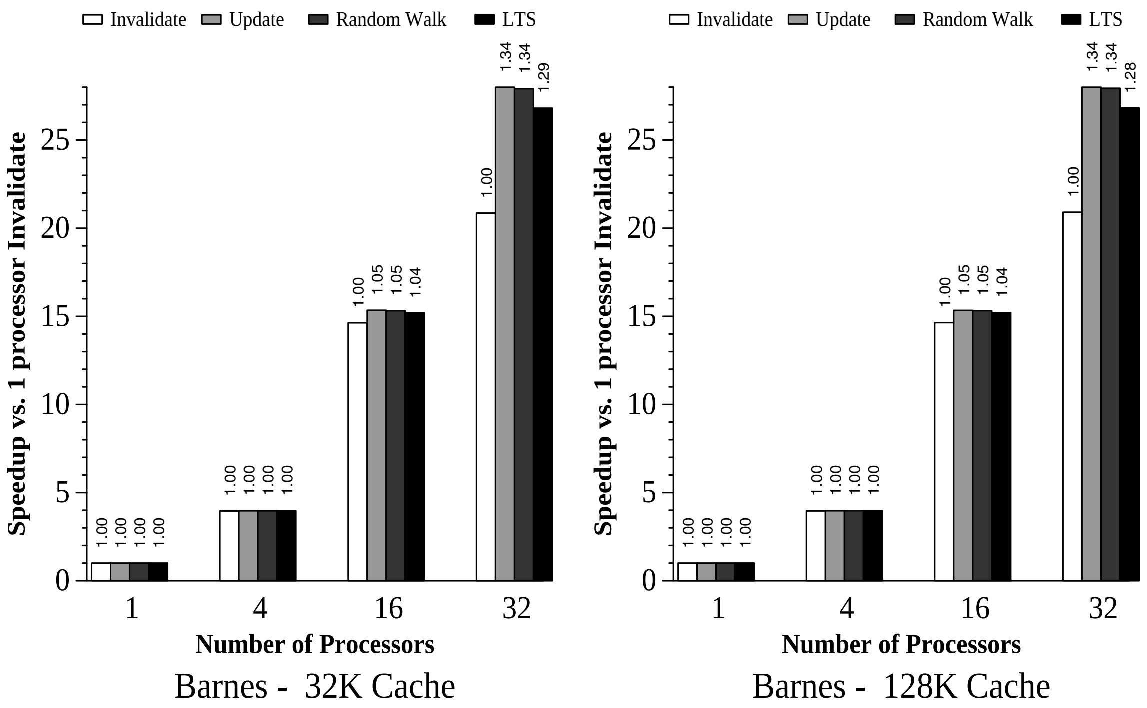


Figure 4.26: Speedups for Barnes (without snarfing)

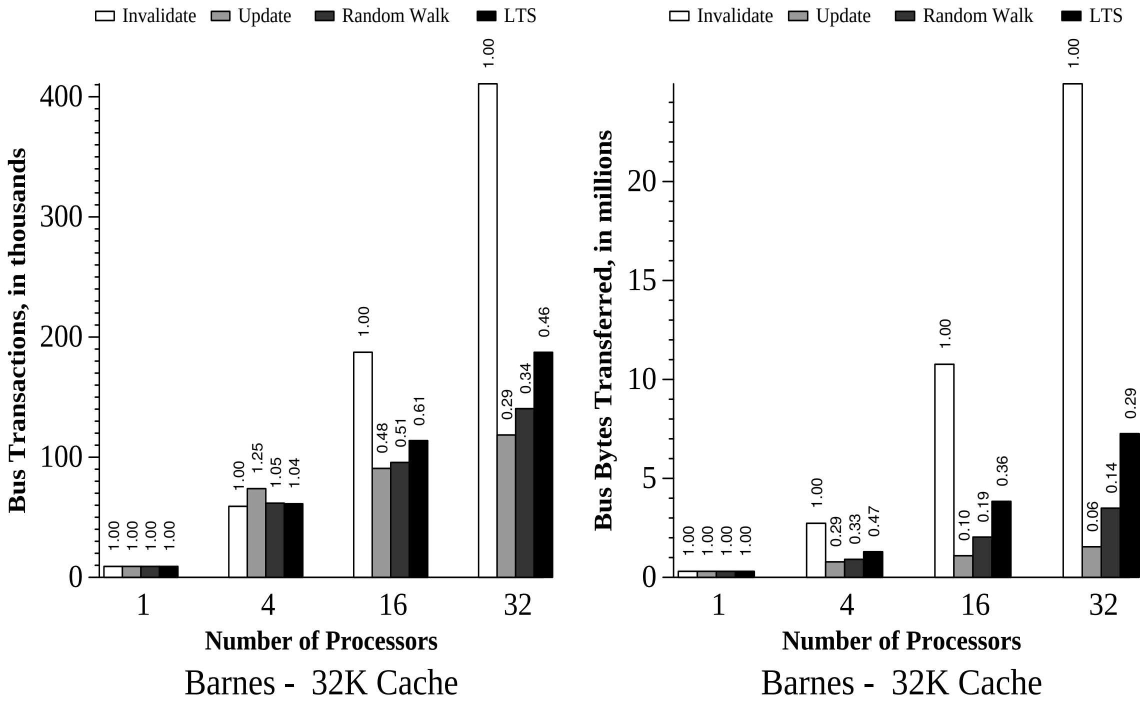


Figure 4.27: Bus transactions and Bytes transferred for Barnes (without snarfing)

strategy more often than when snarfing was turned on. This caused the number of bus transactions for the adaptive protocols to rise sharply in comparison to when snarfing was used (Figures 4.29 and B.16), though WU still had more transactions than the other three protocols. While the adaptive protocols transferred fewer bytes than WI or WU when snarfing was on, without snarfing the adaptive protocols transferred more bytes than WI (but fewer than WU) when 4 or 16 processors were used. Again, this was a consequence of using a larger value for the invalidate ratio, which tended to favor the use of the update strategy over the invalidate strategy.

MP3D

Turning off snarfing adversely affected all 4 protocols[2]. The lack of snarfing affected the performance (relative to WI) of both WU and LTS — it increased the relative performance of WU to the point that it performed almost as well as RW, while it decreased the relative performance of LTS to about 5–10% above that of WI (Figure 4.30).

Because an invalidate ratio of 18 was used with the adaptive protocols, updating was used more often in the adaptive protocols than when snarfing was turned on. The higher ratio meant that Random Walk switched to invalidate very rarely, which means that its performance was about the same as WU, especially when 32K caches were used. With 128K caches, RW outperformed WU by 2 percentage points when using 16 processors, and 7 points with 32 processors. Random Walk and WU had similar read miss rates and numbers of bytes transferred. They differed in the number of bus transactions when 128K caches were used — RW had about 70% more transactions than WI, while WU had about 80% more transactions. These additional transactions caused the slight loss in performance of WU relative to RW.

In contrast with RW, LTS used invalidation quite often, which led to a relative decrease in performance. LTS's miss rate was higher than either WU or RW, and approached that of WI at 32 processors (Figure B.17). LTS used about the same number of bus transactions as RW (Figure 4.31), but transferred more bytes than

[2] Snarfing works by supplying data to caches that have invalid blocks. Though the write update protocol does not create invalid blocks through invalidate transactions, such blocks are created when processors issue simultaneous bus requests. In the case of simultaneous read requests, snarfing supplies the block to all requesting processors other than the one whose request succeeded.

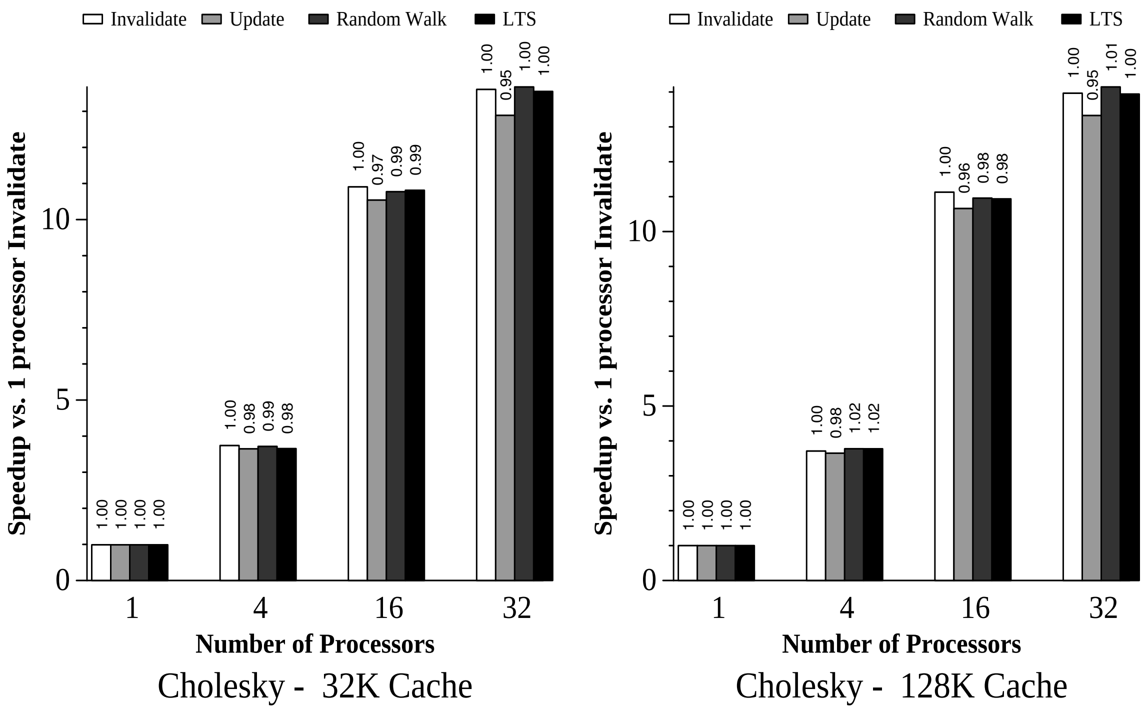


Figure 4.28: Speedups for Cholesky (without snarfing)

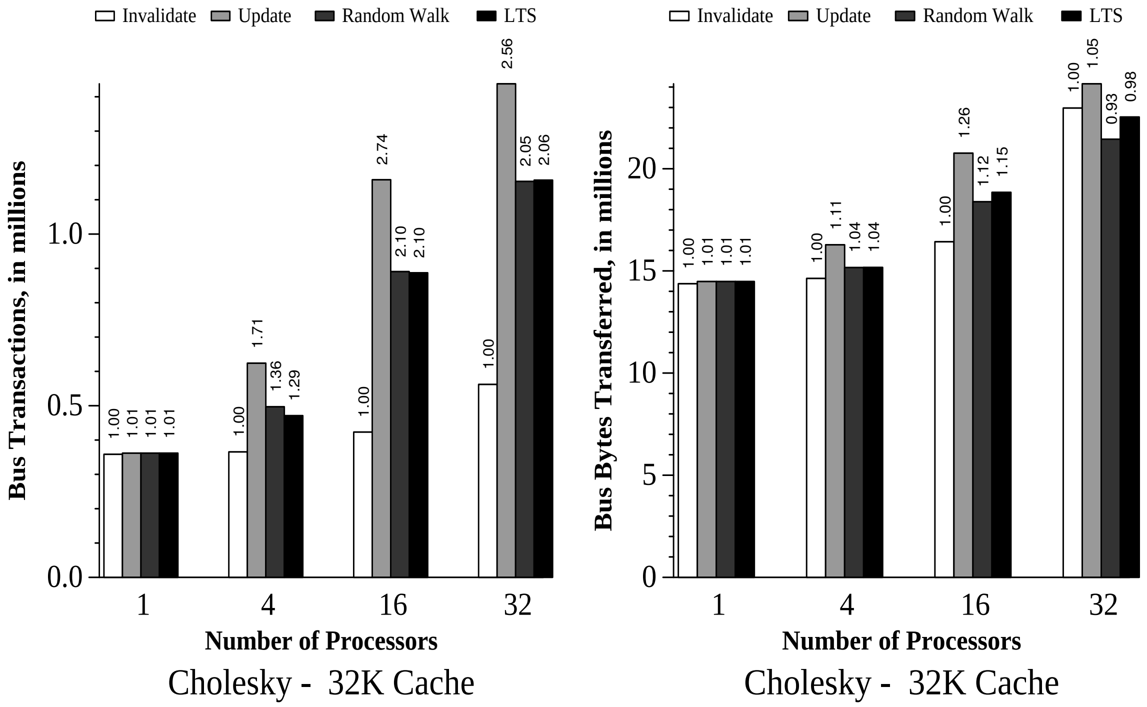


Figure 4.29: Bus transactions and Bytes transferred for Cholesky (without snarfing)

either RW or LTS. The additional data that was transferred directly led to the slowdown of the program under LTS.

In summary, RW performed the best of the four protocols, both with and without snarfing. With snarfing, RW performed nearly as well as LTS, and better than WU and WI. Without snarfing, RW equaled or exceeded the performance of all other protocols on all configurations.

Pverify

As expected, turning read snarfing off had a significant adverse effect on WI because of Pverify's large amount of false sharing. Performance for WI without snarfing dropped 18% for 32 processors and 32K caches; for 128K caches, the performance drop was 37% (cf. Figure 4.32). The lack of snarfing also caused increases in the read and write miss rate for WI, as well as increases in the number of bus transactions and bytes transferred (Figure 4.33). The percentage increases were worse when 128K caches were used, since a greater portion of cache misses and bus traffic are due to coherence actions when larger caches are used. In contrast, the lack of snarfing had little effect on overall performance of WU and RW, while it did have a small negative effect on LTS. The biggest performance drop (26%) for LTS happened when using 32 processors and 128K caches. The performance of LTS dropped more than RW because it tended to switch to invalidate more often than the Random Walk protocol. The only category in which there was a large difference when snarfing was turned off for the adaptive protocols was in the write miss rate; fortunately, only 2% of all data accesses were shared writes.

Topopt

Turning snarfing off had a severe performance impact on WI (Figure 4.34). Where WI achieved speedups of 20 or larger using 32 processors with 32K caches and snarfing turned on, without snarfing Topopt under WI managed a speedup of only 7. Of the two adaptive protocols, Random Walk was not affected by the lack of snarfing, but it did have a negative effect on LTS when 16 or 32 processors were used. An anomalous result occurred with WU in that Topopt without snarfing ran faster using both 16 and 32 processors and 128K caches than the version with snarfing. This despite the fact that latencies for read and update operations were lower using snarfing. It is

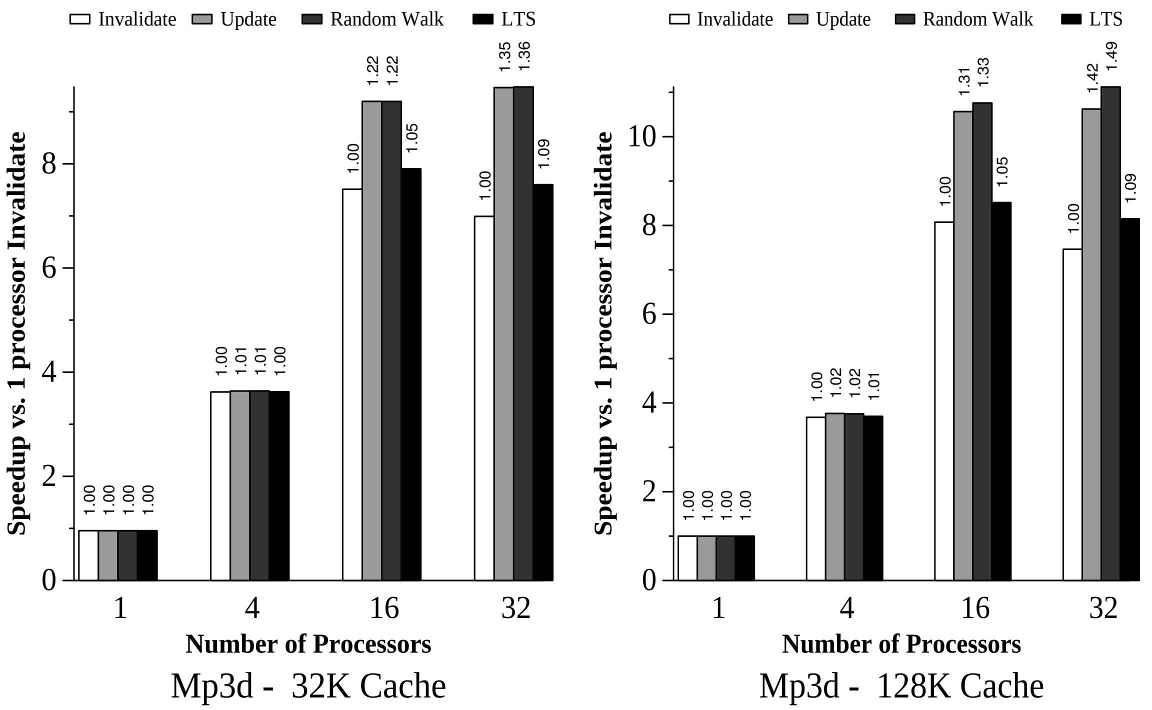


Figure 4.30: Speedups for MP3D (without snarfing)

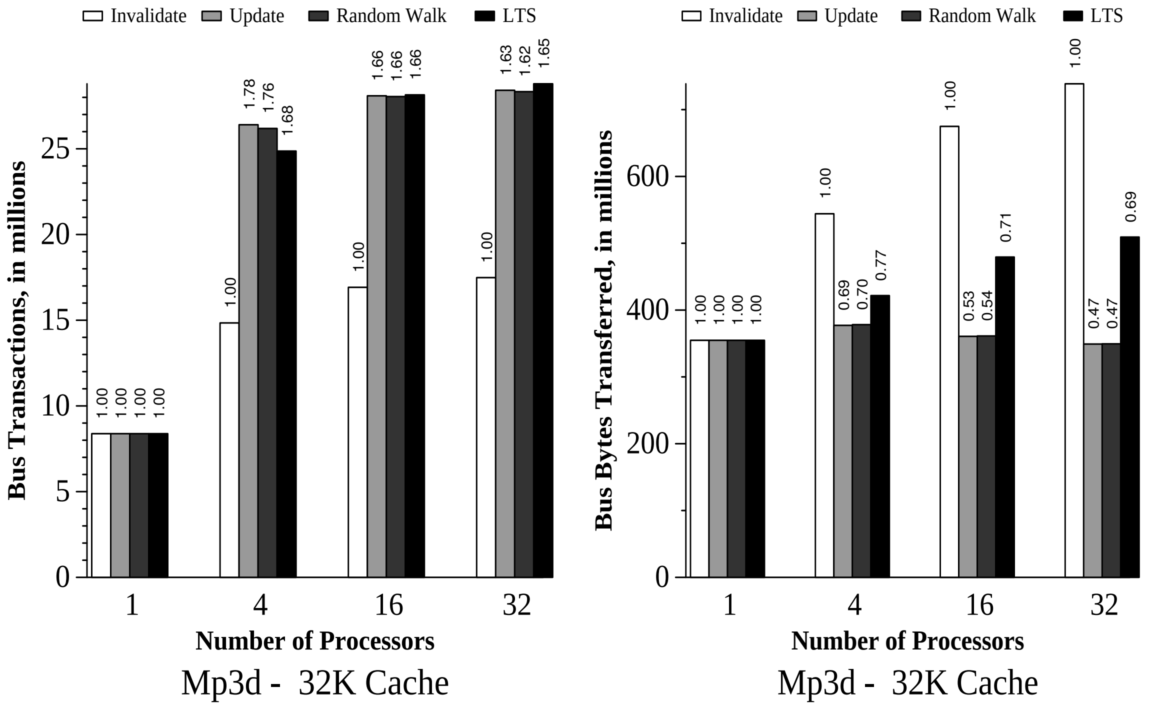


Figure 4.31: Bus transactions and Bytes transferred for MP3D (without snarfing)

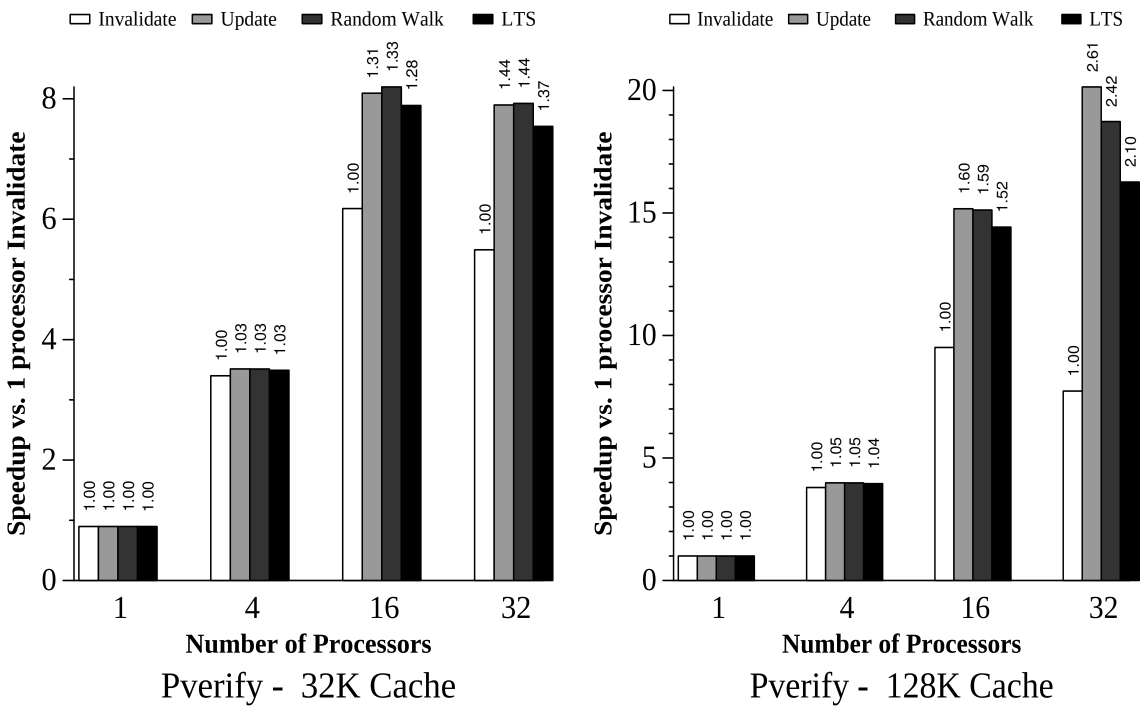


Figure 4.32: Speedups for Pverify (without snarfing)

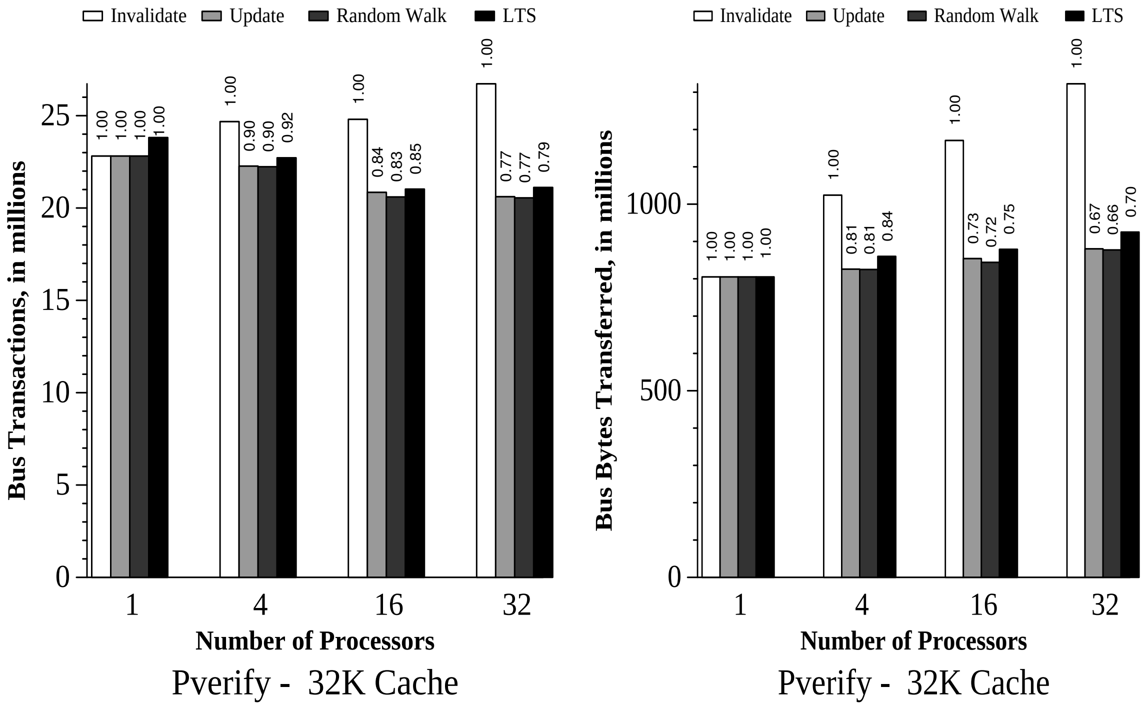


Figure 4.33: Bus transactions and Bytes transferred for Pverify (without snarfing)

likely that turning off snarfing decreased contention for the lock guarding the barrier code, which led to increased performance.

The behavior of WI without snarfing showed some surprising results in the areas of read miss rates and bus traffic metrics. Turning snarfing off actually decreased both the read and write miss rate, while it increased the number of bus transactions and the number of bytes transferred. The increases were relatively small for 4 processors, but were very large when 32 processors were used. The main reason this occurred is that there are five barriers in the inner loop of the simulated annealing code. Since barriers were implemented using a simple algorithm and test and test and set (TTS) locks, there was a great deal of contention for the lock guarding the critical section of the barrier code. This contention increased bus traffic, which made lock releases take longer than if read snarfing were used. Since locks were effectively held longer when snarfing was not used, processors spun longer on the value of the lock variable, which increased the number of cache hits. In addition, because bus latencies without snarfing were much higher because of additional reread traffic, the time interval between successive memory operations by a given processor increased. This decreased the degree of active sharing in the program; in fact, the number of invalidations dropped by 40% and the number of read misses dropped by 16% when snarfing was turned off. However, these drops were small in comparison to the large increase in the number of bus read transactions.

In contrast with WI, turning off snarfing had a small (negative) effect on both RW and WU with regard to the read miss rate, the number of bus transactions, and the number of bytes transferred. The change had more of an effect on the LTS protocol, since it tended to choose invalidate more often than RW did. Turning off snarfing caused an 11% drop in LTS's performance for 16 processors and both cache sizes, and for 32 processors and 128K caches. The largest performance drop (24%) happened with 32 processors and 32K caches. For this configuration, the read miss rate increased by 17%, bus transactions increased by 45%, and the number of bytes transferred increased by 120%. The bus contention caused by the increased traffic led to the speedup drop experienced by LTS.

While LTS did not perform without snarfing as it did with snarfing, Random Walk's performance was barely affected by the lack of snarfing. It still equaled or exceeded the performance of WU (and WI) on all configurations.

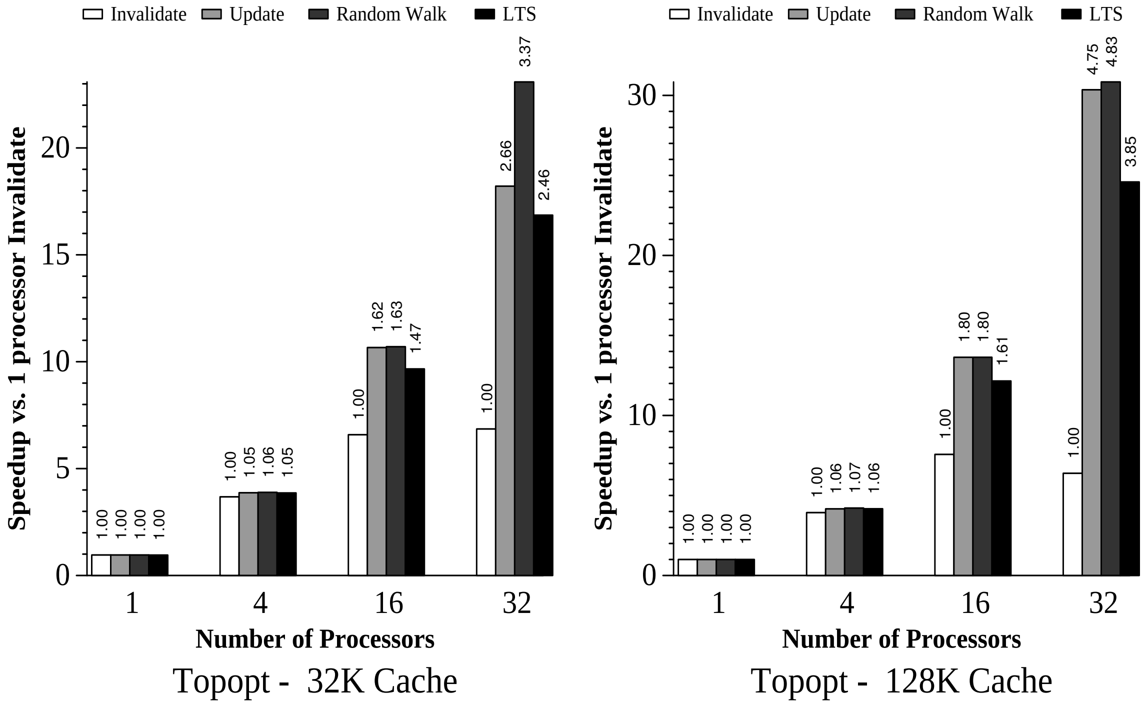


Figure 4.34: Speedups for Topopt (without snarfing)

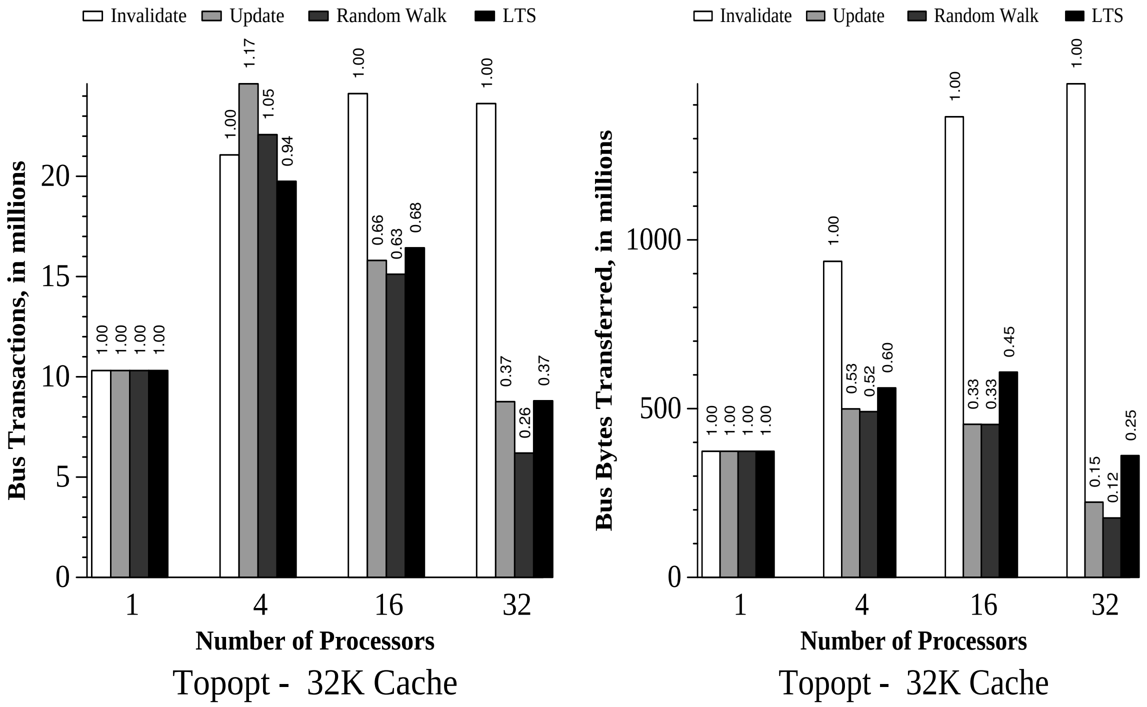


Figure 4.35: Bus transactions and Bytes transferred for Topopt (without snarfing)

Part II

Hierarchical Bus Improvements

Chapter 5

THE HIERARCHICAL PROTOCOL

5.1 Introduction

Though single-bus multiprocessors have many advantages, it is difficult to effectively connect a large number of processors to a single bus because the limited bus bandwidth is quickly exhausted. One solution is to use a *hierarchical architecture*, which scales to more processors than a single bus system. The term "hierarchical" is used because these systems are somewhat analogous to the memory hierarchy in uniprocessor systems: different levels of storage exist in the system, each with its own performance characteristics. Multiprocessor hierarchical designs usually resemble trees, with the processors at the leaves. Processors are grouped into clusters (or hierarchies of clusters), each with its own interconnect, so the architecture can exploit communication locality by limiting data exchanges to the smallest cluster that contains both communicating entities. This significantly reduces memory latency compared to single bus systems, which increases application performance. It also reduces broadcast traffic on the interconnect, which allows the memory system to accommodate additional processors. An additional feature of some hierarchical systems is that data may be moved or replicated in the system so that it is close to the processors that are using it.

The hierarchical architecture classification is a very broad one, and encompasses many different types of systems. Table 5.1 lists some hierarchical systems and their important features. In the next few paragraphs, we examine those features and their tradeoffs.

One of the most important design features of a hierarchical system is the type of interconnect used. Choosing the interconnect not only determines the latency and bandwidth available to processors (and clusters), but also influences what, if any coherence protocol will be used. Among the choices for interconnect are:

Busses An interconnect commonly found in uniprocessors, busses can also be used in multiprocessors. Busses are relatively simple structures that have low latency when lightly loaded. By their nature, busses provide low-cost broadcasting capabilities. The major drawback of busses is limited bandwidth — busses normally allow only one module to transmit at a time, and bus length limits how fast the bus can cycle. Chapter 2 of Robert Bedichek's thesis [Bed94] has a good discussion of the trade-offs in designing high-performance busses. Some high performance busses used in multiprocessors include the XDBus [SFG+93] (used in the SPARCcenter 2000) and the POWERpath-2 (used in the SGI Challenge).

Rings Rings address the limited bandwidth problem of busses by using only point-to-point links between caches/clusters. Because there is only one sender and one receiver on a given link, and the distance between them is (presumably) short, links can be clocked at higher speeds than busses. Depending on its size, a ring may have multiple transactions in flight at any one time, which increases its maximum bandwidth (though this complicates coherence enforcement).

Rings also provide low-cost broadcast, since caches or clusters can easily observe information as it passes by on the ring. One down side of rings is that latency increases linearly with the physical size of the ring, as well with as the number of elements attached to it.

Meshes In a two-dimensional mesh, raw interconnect bandwidth is further increased over that of a ring by providing each processing element with 4 point-to-point links to other elements. Like a ring, multiple transactions may be in progress at the same time on the mesh. In addition, there are multiple paths in a mesh between a sender and receiver. The existence of redundant paths means that interconnect latency can remain low even when there is other traffic on the mesh.

A major difference between meshes and rings or busses is that broadcasting information to all elements is an expensive operation. Because of this, snoopy cache protocols are not used in mesh architectures.

Crossbars Crossbars are another high-bandwidth interconnect in which each processor or cluster has its own dedicated connection, as do the memory modules.

Switches at the intersections control which processor is connected to which memory module at any given time. Crossbars offer high bandwidth, but can be costly, since the overall number of switching elements grows as the square of the number of elements connected.

The mechanism used to enforce cache coherence is another important feature of a multiprocessor. Ensuring coherence in hardware can be a very difficult problem, so much so that some machines rely on software to provide coherence. Here are some hardware methods for providing cache coherence:

Snoopy In a snoopy cache coherence protocol (as described in Part I of this thesis), each cache is responsible for keeping itself coherent by observing memory transactions on the interconnect. Because each cache which may be affected by a memory transaction must observe the transaction, snoopy protocols are usually implemented on systems which have inexpensive broadcast capabilities. Snoopy protocols are usually cheaper and less complicated than other coherence strategies.

Directory Directory-based coherence protocols record which caches share a given block in a directory entry, which is associated with the block in main memory. When a write is done to a block, the directory is consulted, and update or invalidate messages are sent to the appropriate caches. The simplest directory scheme is the full-map directory [CF78], in which each directory entry consists of one bit for each element which might cache the block, plus one additional bit to indicate if the block has been modified with respect to memory. The DASH system uses a full-map directory. To save directory bits, limited directory protocols can be used. An example of such a protocol is the LimitLESS protocol [CKA91], which uses a small number of hardware pointers for each memory block. If the number of pointers is insufficient to maintain the block's sharing list, then a full-map directory protocol is emulated in software to keep the block coherent.

SCI The Scalable Coherence Interface includes not only a cache-coherence protocol, but also specifies a standard for physical and logical connections [Gus92]. The SCI cache coherence protocol consists of a doubly-linked sharing list of caches which have the block. With each cache block is stored pointers to the previous and next cache on the sharing list. The pointer to the head of the sharing list (if any) is stored with the memory block. The advantage of SCI over snoopy protocols is that it doesn't need a cheap broadcast mechanism. The SCI protocol also scales relatively well with an increase in the number of caches or the amount of memory in a system. The amount of storage required is two pointers for every cache block, and a pointer and some state bits for every memory block. The drawbacks are that invalidations or updates require time proportional to the number of caches sharing the block, and that the protocol itself is fairly complicated, since it entails distributed modifications to a distributed data structure without the use of a global lock.

Another design feature is that of a shared, or cluster cache. Such a cache can serve two purposes. First, if the cluster cache maintains *inclusion* [BW92] with regard to the caches below it, then the cluster cache can filter requests from the upper level(s) of the system to the lower (cluster level), since the cluster cache contains directory information for all blocks cached below it. Second, if the cluster cache contains data blocks, it can satisfy requests which miss in the cluster's caches, reducing both overall memory latency and traffic on the upper interconnect.

The last feature we will discuss is that of main memory organization. The alternatives are:

Uniform Memory Access (UMA) Main memory is at the top of the hierarchy, or root of the tree.

Non-Uniform Memory Access (NUMA) Main memory is divided into sections, usually with one section per processor or cluster. An interesting problem which arises from this choice is the placement of data structures in memory so that data is located close to the processor(s) that are most heavily using it.

Cache-Only Memory Architecture (COMA) A COMA machine has no main memory; instead, large Main Caches (called Attraction Memories in [HLH92]) take the role of main memory. Because blocks are stored only in Attraction Memories, the data associated with a given address has no fixed home location.

Since the current location of a given data block is not made known to all processors all of the time, efficient means of searching for a block are necessary. To do this effectively, the interconnect in a COMA should support efficient broadcast, allowing all caches on a given level to be examined in parallel. Both COMA machines in Table 5.1 support efficient broadcast. COMA machines also require more tag storage bits, since each block in the large Attraction Memory requires one.

Table 5.1: Hierarchical Systems

Feature	Architecture						
	Wilson	DDM	KSR	Stanford DASH	Galactica Net	Convex Exemplar	Hector
Cluster Interconnect	Bus	Bus	Ring	Bus	Bus	5X5 Crossbar	Bus
Top Interconnect	Tree of Busses	Tree of Busses	Tree of Rings	Mesh	Mesh	Four Rings	Two Level Ring
Cluster Cache Coherence	Snoopy	Snoopy	Snoopy	Snoopy	Snoopy	Directory	Snoopy
Top Cache Coherence	Snoopy	Snoopy	Snoopy	Directory	Hybrid	SCI	None
Cluster Cache?	Yes	Dir.†	Dir.†	Yes	Yes‡	Yes	No
Inclusion?	Yes	Yes	Yes	No	N/A	Yes	N/A
Memory Organization	UMA	COMA	COMA	NUMA	NUMA	NUMA	NUMA
Reference	[Wil87]	[HLH92]	[FIR93]	[LLJ+92]	[WTP+92]	[Con94]	[VSLW91]

†Dir.: Only directory entries are cached, not data.
‡Each cluster shares a portion of main memory, which can be used as a cache of other clusters' pages.

An advantage of hierarchical architectures over others is that in the future, the first (bottom) level may be placed on a single integrated circuit chip. This will become technically feasible as circuit density increases. It is less likely that the additional space will be used for caches, since increasing the size of a cache usually slows down access to it. A study done by Nayfeh and Olukotun [NO94] concluded that with current technology, putting two processors per chip leads to better performance than one processor per chip with a larger cache. They also found that using multi-chip modules to increase the number of processors per cluster leads to further performance increases.

5.2 Our research

In this second part of our thesis, we examine enhancements to a hierarchal bus architecture. In this chapter, we present an invalidation-base protocol for a tree hierarchy of busses. Later chapters will present simulation results for the base protocol described in this chapter, as well as enhanced versions of the base protocol.

Because processors can't observe all transactions on all busses, a hierarchical bus coherence protocol is significantly more complicated than a single bus protocol. After describing the simulated architecture (Section 5.3), we present an overview of the protocol in Section 5.4. We show that there is a need for transitional cache states that are not necessary in single bus architectures. In the final section, we illustrate the actions of the protocol.

5.3 The architecture

The protocol is designed to run on an architecture that consists of a hierarchy of busses as shown in Figure 5.1. Figure 5.1 depicts a machine with 3 levels of hierarchy; the protocol that we developed works on a machine that may have from two to an arbitrarily high number of levels. At the bottom of the hierarchy (level 0) are processors along with their caches. The processors are grouped into clusters of some number of processors per cluster, typically two to eight. Each processor cache has a dual directory which permits simultaneous processor access and cache coherence checks initiated by bus requests.

The intra-cluster processors are connected to one another via a conventional bus, which we call the level 1 bus. Also connected to the intra-cluster bus is an intra-cluster cache/bus connector. This component both connects each level 1 bus to the corresponding level 2 bus, and also caches data for the processors in the cluster below it. To reduce bus traffic, all caches in the system are write back caches. All non-processor caches in the system satisfy the *inclusion* property, that is, each cache at a given level contains a superset of the contents of the caches below it in the hierarchy. The primary reason for doing this is to limit unnecessary propagation of bus traffic from upper (i.e. towards the root of the tree) levels of the hierarchy to lower (cluster) levels [BW88]. Because all caches obey inclusion, an upper level cache will not forward requests on its upper bus to its lower bus if the block in question is not present in its cache, because it is known not to be present in the caches below it. Note that inclusion is guaranteed not through associativity, but through the actions of the protocol, as in [Wil87]. When an upper cache replaces a block in its cache, it sends a message down that forces all caches below it in the hierarchy to invalidate the block.

Each bus connector/cache (hereafter referred at as a *bus cache*) has a dual directory so that operations on both busses it is connected to can proceed in parallel. Each bus cache also has a number of bus request buffers that hold both bus requests that have been enqueued waiting for access to either bus, as well as information about read/write/swap operations in progress that are being acted on in other parts of the system.

In a three level system, the bus caches for the processor clusters are themselves divided into groups called *super clusters*. Each group's cache clusters are connected to each other via a bus. In addition, there is a super cluster bus cache which connects the super cluster to the rest of the system. In a three level system, the super cluster bus caches are connected via a bus to each other and to main memory; however, if the system has more than three levels, the super cluster caches are further grouped into super clusters via a bus and an upper level cache.

At the top of the hierarchy is memory. Like in single bus snoopy protocols, blocks found in memory have no state. Requests for blocks that are not found in any cache progress up the hierarchy until they reach the top, where they are satisfied by main memory.

In the remainder of this chapter, caches will be referred to using the notation Cache(S, L, N) where:

S is the supercluster number,

L is the level number, and

N is the number of the cache on that level.

In addition, cache states will appear in capital letters, (such as VALID EXCLUSIVE) while bus commands will appear in italics (such as *READ EXCLUSIVE*).

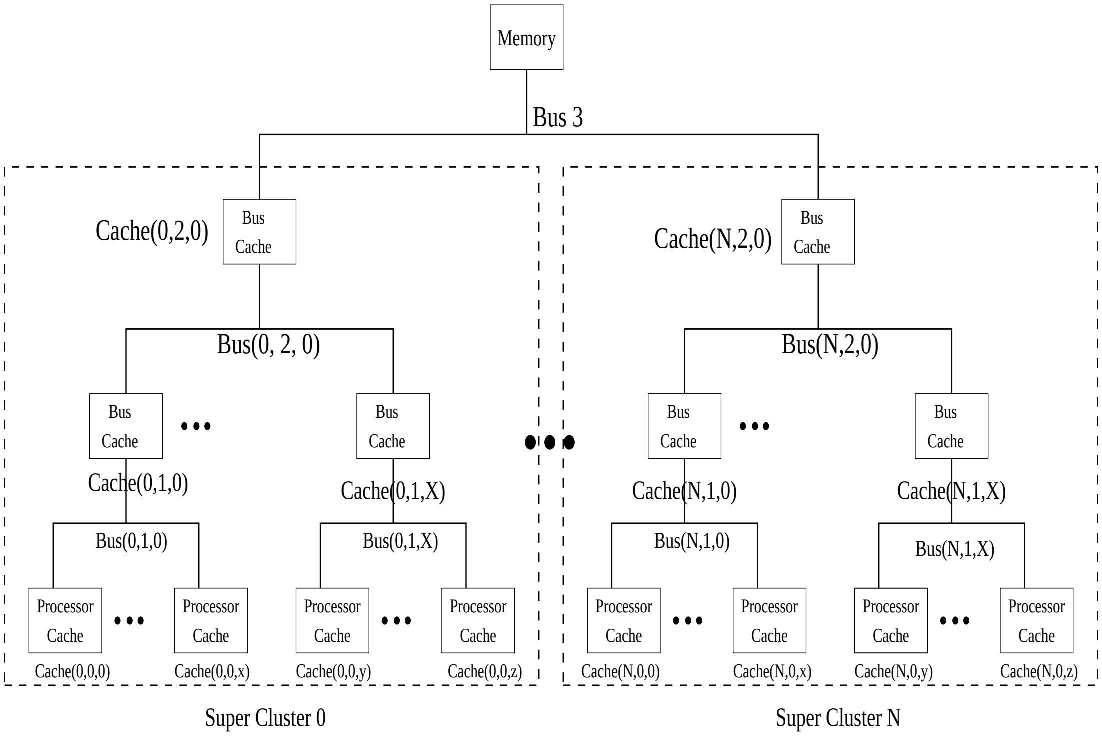


Figure 5.1: Machine Architecture

5.4 Overview of the protocol

The protocol is designed to minimize response time and bus traffic, especially at the higher levels of the hierarchy. To do this, requests are satisfied as close as possible to the requesting processor. The protocol will attempt to satisfy a request using caches at the same level as that as that of the requester; the request is forwarded to higher levels only when necessary. In some cases, a request may need to travel up the hierarchy, then back down to another branch of the tree; the reply retraces the same path back to the request originator. In addition, blocks are written back only to the next higher level of the hierarchy. Only when a cache at the top the hierarchy writes a block back does the block migrate to memory.

5.4.1 Protocol states

Processor level

The processor level protocol most closely resembles the Illinois protocol [PP84]. The non-transitional states are as follows:

INVALID The block is not present in the cache.

VALID EXCLUSIVE The block is present in no other processor cache except this one, and the data has not been written by the processor.

READ SHARED The data in the block is up-to-date, and is possibly shared by other processor caches. The data may be dirty with respect to main memory, but it is not the responsibility of this cache to write it back.

DIRTY The block is present in no other processor cache except this one, and this cache has modified the data with respect to memory or other non-processor caches.

There are also transitional states which will be detailed later.

Non-processor cache states

The non-processor caches have the following non-transitional states:

INVALID The block is not present in this cache or any cache below it in the hierarchy.

READ SHARED The block is present in the cache in a clean state. The block may exist in a clean state in caches below this one in the hierarchy. The block may exist elsewhere in the system in a clean state.

VALID EXCLUSIVE The block is present in the cache, but is not guaranteed to be up-to-date. The block may exist below this cache in a dirty or clean state.

DIRTY SHARED The block is present in the cache and is dirty with respect to memory. This cache is responsible for writing the value back to the next higher level in the hierarchy. In addition, the block may exist lower in the hierarchy in READ SHARED state only.

DIRTY OWNED The block is present in the cache and is dirty with respect to memory. The block does not exist lower in the hierarchy.

5.4.2 Bus request types

Having different states for non-processor caches is usual for hierarchical architectures [Wil87], [YTB92]. Depending on cache state and processor actions, the following bus requests can be present (see Appendix C for a complete specification of state transitions and bus request types).

READ A request to load a cache block.

READ DOWN A request to load a cache block that is traveling down one part of the hierarchy after traveling up another part of the hierarchy.

READ EXCLUSIVE A request to load a cache block so that the originating processor may write the block.

READ EXCLUSIVE DOWN Similar to a *READ EXCLUSIVE* request except that it is traveling down the hierarchy.

WRITE BACK PURGE The block is being written back to the next higher level in the hierarchy due to replacement in the lower level cache.

INVALIDATE A request to invalidate other copies of a given block.

INVALIDATE DOWN A request to invalidate other copies of a given block. The request is traveling down the hierarchy, unlike an *INVALIDATE* request which travels up the hierarchy.

SWAP INVALIDATE Similar to an *INVALIDATE* request except that the block is swapped rather than written in the originating cache.

SWAP READ EXCLUSIVE Similar to a *READ EXCLUSIVE* request except that the block is swapped rather than written in the originating cache.

PURGE A request to invalidate the block in question in all caches at this level of the hierarchy and below. No response is needed by the originating cache.

PURGE REPLY Like a *PURGE* request, except that the originating cache waits for some sort of reply from a cache below it.

5.4.3 Processor cache actions

Hits in processor cache

Because the emulated processor has a load/store based instruction set, all traffic to and from memory is generated by explicit instructions. The three types of instructions are LOAD, which loads data from memory; STORE, which writes data to memory; and SWAP, which atomically exchanges the value in a register with a word in memory. A processor cache access will hit and require no further action on the part of the memory system under the following circumstances. If the block is DIRTY or VALID EXCLUSIVE, all accesses (LOAD, STORE, and SWAP) hit. If the block is VALID EXCLUSIVE and the request is not a LOAD, the block status is changed to DIRTY. In addition, a LOAD request to READ SHARED block hits. All other cases are considered processor cache misses.

Misses in processor cache

When a request misses in a processor cache, the processor is forced to wait until the request is satisfied. This is not a requirement of the protocol; it simply made the simulator easier to implement. The processor's cache allocates a block for the data (if necessary) and enqueues the appropriate bus request. There are several miss cases:

- A LOAD request to a block in the INVALID state generates a *READ* request.

- A STORE request to a block in the INVALID state generates a *READ EXCLUSIVE* request.

- A STORE request to a block in the READ SHARED state generates an *INVALIDATE* request.
- A SWAP request to a block in the INVALID state generates a *SWAP READ EXCLUSIVE* request.
- A SWAP request to a block in the READ SHARED state generates a *SWAP INVALIDATE* request.

Because conflicting requests can be generated "at the same time" by different processors, and because there is no implicit serialization device such as the bus in a single bus system, it is possible that a request generated by a miss will fail (cf. Section 5.4.5). If the request succeeds, the processor is restarted; otherwise the processor will wait some amount of time after the failed request has been returned to it, then retry the request. Ideally, the processor cache would wait just long enough for the condition that caused the request to be retried to resolve itself; in practice, it is impossible to know how long to wait. We used a fixed retry waiting time of 15 cycles. We also evaluated an Ethernet-style backoff scheme, in which the retry waiting time is doubled when a retried request is retried again. Note that a *SWAP READ EXCLUSIVE* (respectively *SWAP INVALIDATE*) request is handled exactly the same way as a *READ EXCLUSIVE* (respectively *INVALIDATE*) by the memory system; the only difference is how the originating cache handles the request when it returns.

5.4.4 Bus cache actions

Generally, a hit in a bus cache means that the request will not need to proceed further up (or down) the hierarchy. A bus cache in a READ SHARED state can satisfy a *READ* request, while a bus cache in either the DIRTY OWNED or DIRTY SHARED state can satisfy *READ* or *READ EXCLUSIVE* requests. Because of the large number of possible state/request combinations, bus cache actions will not be further detailed here; consult Appendix C for more information.

5.4.5 Need for transitional states

In addition to the above states, the protocol uses transition states to inhibit additional requests for a block when the state of the block is in flux. This helps to greatly reduce the complexity of the state space of a given block. For example, suppose a processor wishes to write to a given block which is not in its cache (call it cache **X**). The block exists in a cache (call it cache **Y**) in another cluster in state DIRTY. To obtain the block, cache **X** must send a request which propagates up and then back down the hierarchy to cache **Y**. Assuming the request succeeds, the up-to-date value of the cache block won't be found in any cache until the reply has reached cache **X**. Because of this, all other requests for this block must be cancelled and retried until the block is in a stable state in cache **X**. This is done by allocating a block (if necessary) on each cache on the path from **X** to **Y** and assigning a transitional state to that block. Then, when under the normal cache snoop mechanism a cache detects a request on the bus(es) it is attached to that matches a block in a transition state, it signals that the request should be cancelled. The request is then sent back to the originator, which will try the request again after waiting a period of time.

Though the current protocol and simulator use three transition states, a hardware implementation would require only one transition state. Three states were used in the simulator for ease of design and debugging.

Resolving conflicting cache accesses

Occasionally, a request from a cache's upper bus and another request from its lower bus will attempt to access the same cache block on the same cycle. When this occurs, the request from the lower bus will be delayed until the request from the upper bus has been processed. The reason that the request on the upper cache is given preference is that it is likely that the upper request has traveled further than the lower request, hence making it wait for the lower request will likely force the upper request (rather than the lower) to be retried. This is a more expensive operation than retrying the lower request.

In a similar vein, two (or more) conflicting block requests may exist waiting for access to the same bus at the same time. If all the conflicting requests are in lower cache bus buffers, then the first request to obtain the bus has priority over all other requests. However, any upper cache bus request has priority over any lower cache bus request, for the same reason given above that an upper bus transaction has priority over a lower bus transaction if they access the same block on the same cycle.

5.4.6 The transitional states

There are three transitional states:

TRANS INVALID The block is in transition to the INVALID state.

TRANS SHARED The block is in transition to the READ SHARED state.

TRANS EXCLUSIVE The block is in transition to DIRTY, for processor caches; or in transition to the VALID EXCLUSIVE state for bus caches.

5.4.7 Replacing a block

In a processor cache

If a processor needs to replace a block not in the DIRTY state, it simply overwrites the current block with no further action needed. If the block is in the DIRTY state, the processor schedules a *WRITE BACK PURGE* request. When the request is serviced, the block in the second level cache changes its status from VALID EXCLUSIVE to DIRTY OWNED, since no other cache in the cluster has the block.

In a non-processor cache

Replacing a block in a non-processor cache is trickier because of the need to maintain inclusion. Thus a non-processor cache must ensure that all caches below it invalidate the block in question. The simplest case is if the status of the block in the replacing cache is DIRTY OWNED; the cache simply writes the block to the next higher level in the hierarchy. No further action is needed, since the block doesn't exist below this cache. Other cases are more complicated, and require a request to be sent down the hierarchy, and in some cases a reply needs to be sent back to the originator. For more detail, see Section 5.5.2 below.

5.5 Illustrations of the protocol in action

In order to illustrate some (subtle) points in the protocol, we “trace” the protocol through a write miss as well as a write back example. In the figures that accompany the examples, the state in a given cache is for a given block **A**. Block **A** will often be referred to as “the block”. An INVALID state can also mean that the block is not present in the cache.

5.5.1 Write request example

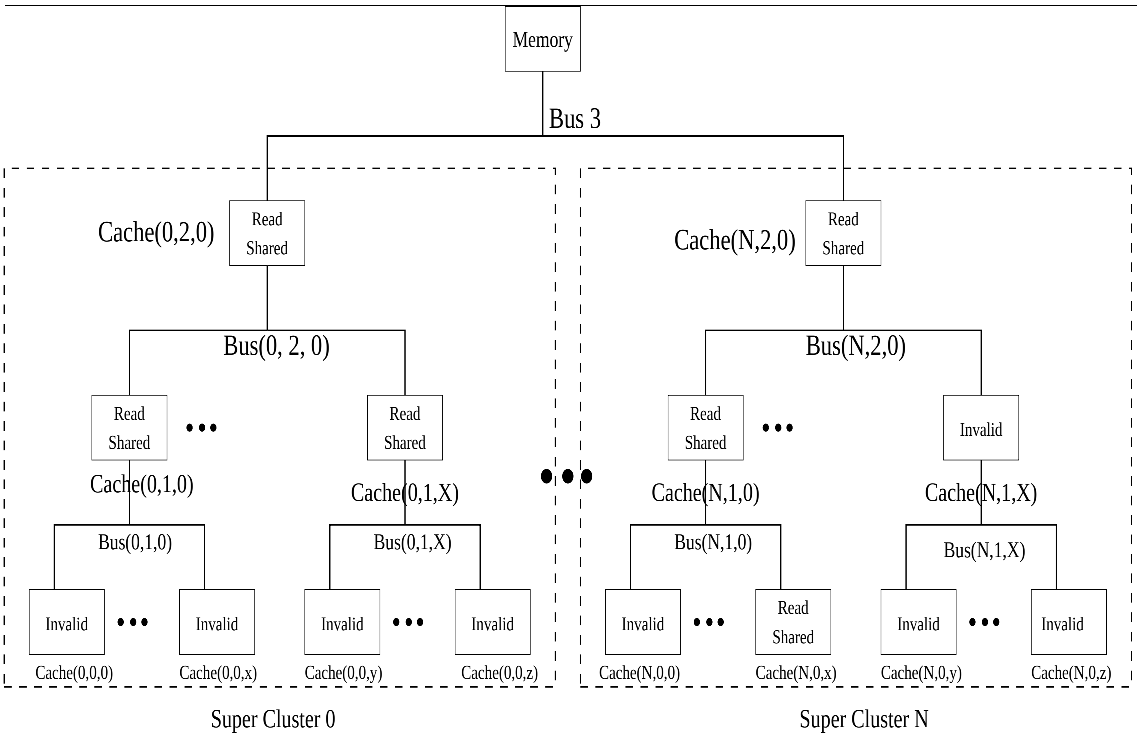


Figure 5.2: Initial State for Write Example

Block A is in the READ SHARED state in caches (0,1,0), (0,1,X) and (N,1,0), and well as the two super cluster caches (0,2,0 and N,2,0).

Assume we have the situation shown in Figure 5.2. Several caches have a copy of the block in the READ SHARED state (two caches at level 2, three at level 1 and one at level 0). Suppose now that the processor attached to cache(0,0,0) wishes to write the block. Since it is not present in its cache, it pre-allocates and pins a block. We pre-allocate the block to avoid some unpleasant situations that could occur if a successful request was returned to a cache that was unable to store the returned block because all blocks were pinned. Processor caches block on misses, so pinning blocks is not necessary; it is necessary in higher level caches however. The processor then enqueues a *READ EXCLUSIVE* request on bus(0,1,0). Once the request gains access to the bus, cache(0,1,0) sends the contents of the block to cache(0,0,0), but also signals that computation cannot proceed until the invalidation of the block in other caches has succeeded. Cache(0,1,0) also pins the block down so it won't be replaced, and enqueues an *INVALIDATE* request on bus(0,2,0). This action is necessary because this level 1 cache has an up-to-date copy of the block, and the only action it needs to perform is to gain exclusive access to the block on behalf of the cache beneath it.

Once the *INVALIDATE* request gets onto bus(0,2,0), the two following actions occur. First, all caches on level 1 in super cluster 0 check whether they have the block; each cache that does have the block changes its status to TRANS INVALID and enqueues an *INVALIDATE DOWN* request on the bus below it. Second, because the upper level cache(0,2,0) has the block in READ SHARED state, it must pass the *INVALIDATE* request up the hierarchy; i.e., repeat the action of cache(0,1,0). Thus it pins the block down to stop it from possibly being replaced, and enqueues a request on the level 3 bus. The system now looks like Figure 5.3.

Once the *INVALIDATE* request gets access to the level 3 bus, it causes all caches (such as cache(N,2,0)) that have the block in READ SHARED state to enqueue an *INVALIDATE DOWN* request on the bus below it, much as cache(0,1,X) did in response to the *INVALIDATE* request on bus(0,2,0). Because *INVALIDATE DOWN* requests are never canceled, they are guaranteed to succeed. The *INVALIDATE* request succeeds because it has reached the top of the hierarchy, which guarantees that the block will be invalidated in the entire system. As a result, a reply is enqueued on bus(0,2,0), and cache(0,2,0) changes its state to TRANS EXCLUSIVE. An *INVALIDATE* request also succeeds when the cache for the bus it is on has the block in either the VALID EXCLUSIVE, DIRTY OWNED, or DIRTY SHARED state. The system now looks like Figure 5.4.

The reply proceeds back down the hierarchy to cache(0,0,0), setting both

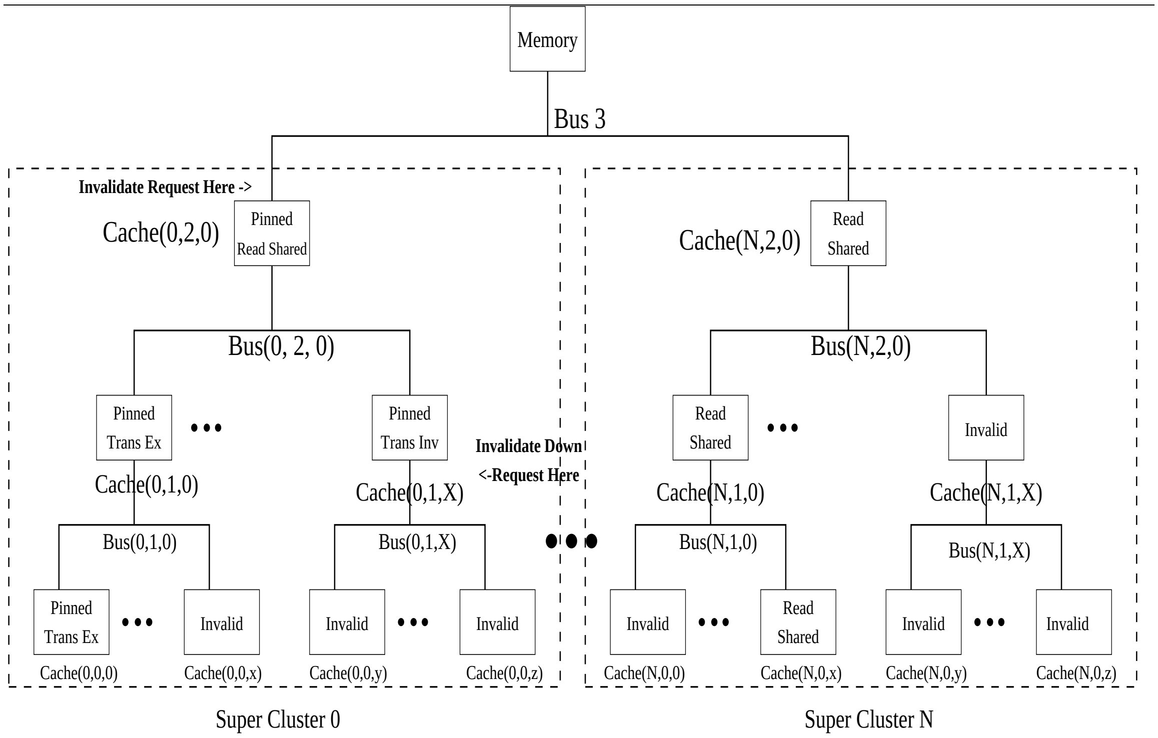


Figure 5.3: Write Request: Step One

The invalidate request has propagated to the top of the left supercluster. An Invalidate Down request has been sent to cluster (0,1,X). The block is in transition in the left supercluster, but is still READ SHARED in caches in the right super cluster.

cache(0,2,0) and cache(0,1,0) to VALID EXCLUSIVE. In the meantime, assuming the *INVALIDATE DOWN* requests on bus(0,1,X) and bus(N,2,0) have been processed, the system will look like Figure 5.5. Note that cache(N,0,x) still has the block in the READ SHARED state, pending the completion of the *INVALIDATE DOWN* request on bus(N,1,0). This is permissible since the system doesn't provide explicit synchronization on ordinary reads and writes. If the application requires explicit synchronization, it must use the SWAP instruction to construct critical sections. The final state of the system is shown in Figure 5.6. Note that the write may proceed when the state of cache(0,0,0) becomes DIRTY (Figure 5.5), which is before all actions involving the block become resolved. Note also that while the READ EXCLUSIVE emanating from cache (0,0,0) is in progress, no other READ

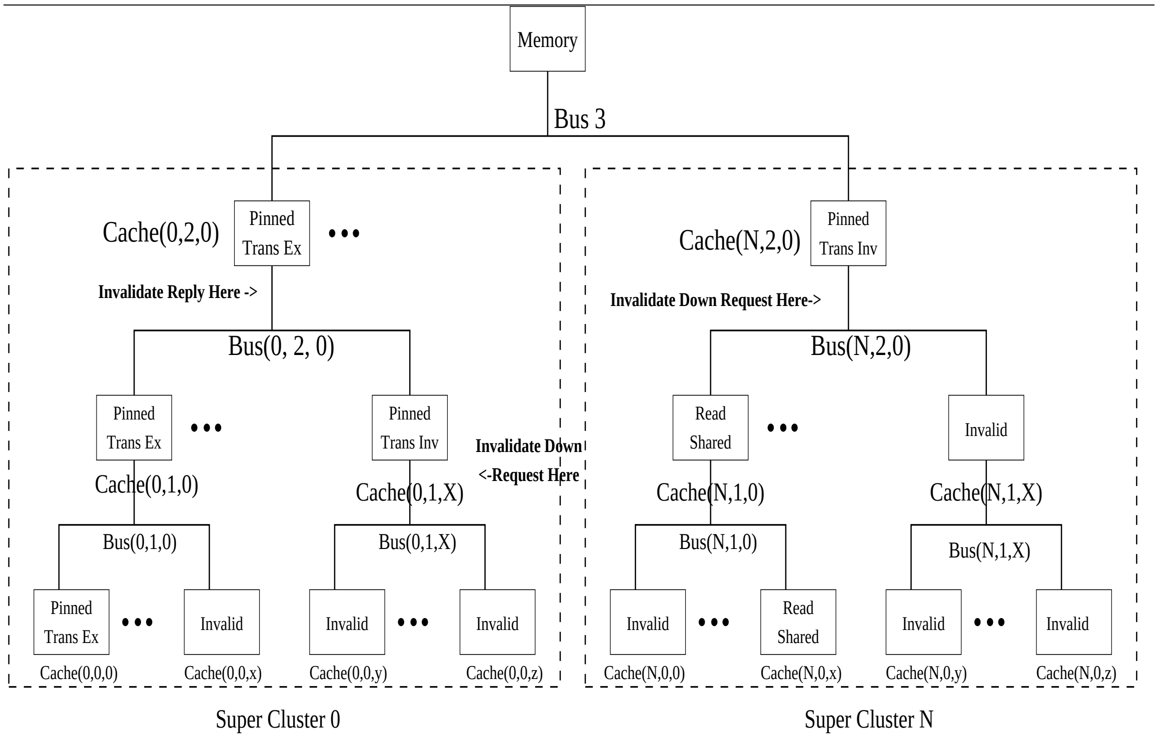


Figure 5.4: Write Request: Step Two

The Invalidate request propagated to the top bus in the system, and thus has succeeded. The successful reply is queued on Bus(0,2,0). In addition, an Invalid Down request is propagating down the right supercluster, where it will invalidate all instances of block A. The block is still in transition in the left supercluster.

EXCLUSIVE will succeed since some transitional state will be reached on the path to cache(0,0,0). Upon encountering the transitional state, a negative acknowledgment will be sent back to the source of the request. The only requests that may succeed while the READ EXCLUSIVE is in progress are READ requests that reach a cache still holding the block in a READ SHARED state, i.e., a cache that has not been reached yet by the waves of invalidations. This is consistent with the view that it is the programmer's responsibility to avoid data races. The workings of the system meet Scheurich and Dubois's three conditions that are sufficient for making a system sequentially consistent [SD87].

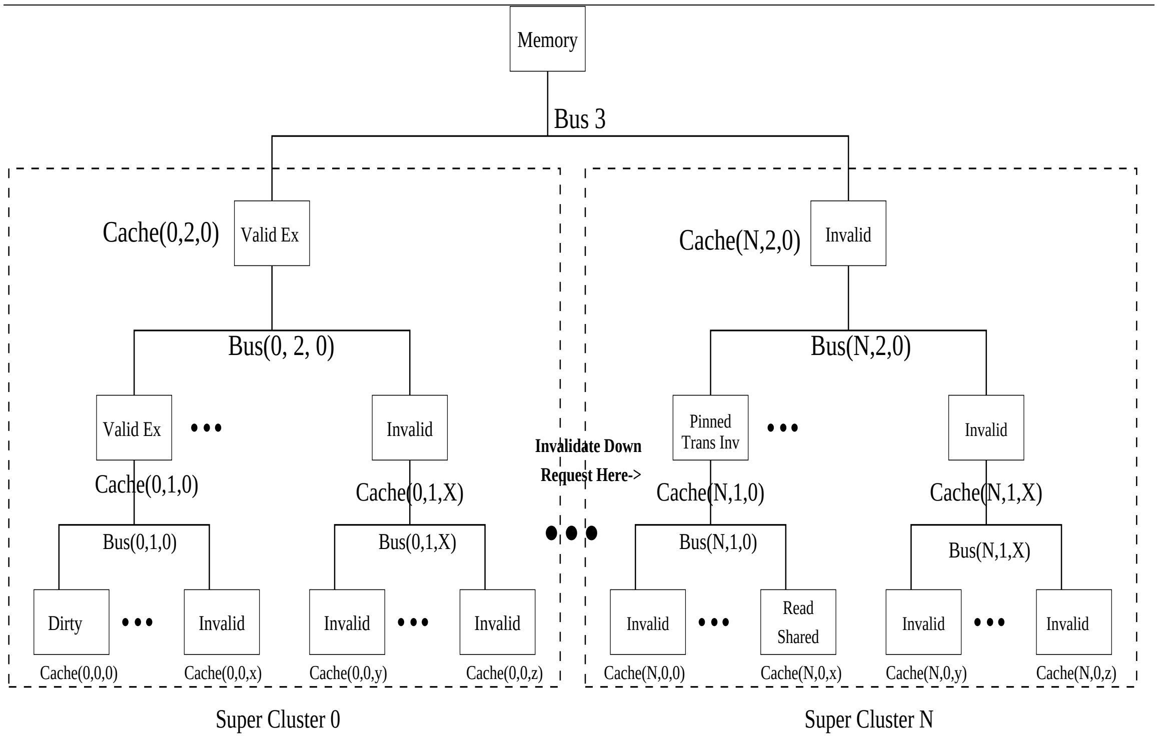


Figure 5.5: Write Request: Step Three

The Invalidate Reply has reached the originating processor, which has written the block (and is now DIRTY). The block is in non-transitional states in the left super-cluster, but is still in transition in cache (N,1,0). The block is still READ SHARED in cache(N,0,x).

5.5.2 Write back example

Replacing a block in a processor level cache is fairly simple (See Appendix C). However, replacing a block in a non-processor cache is a complicated issue, because of the need to maintain inclusion. Upon replacement, a non-processor cache must ensure that all caches below it invalidate the block in question. The process is complicated further by the possibility that the block exists in a DIRTY state below the replacing cache.

If the status of the block in the replacing cache is DIRTY OWNED, the cache simply writes the block to the next higher level in the hierarchy. No further action is

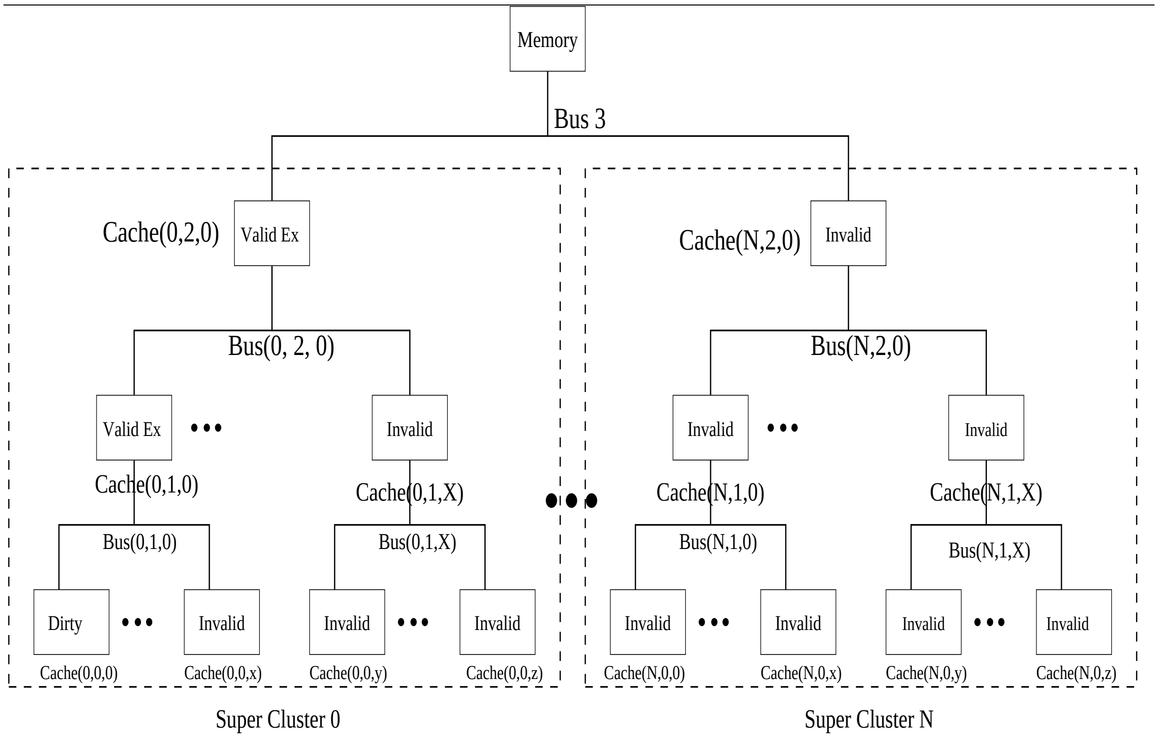


Figure 5.6: Write Request: Final Configuration

All transactions associated with the write request have completed. The block is Invalid in all processor caches except the writer's. The cluster and supercluster caches above the writing processor have the block in the VALID EXCLUSIVE state.

---

needed, since the block doesn't exist below this cache.

If the status is READ SHARED, a *PURGE* request is enqueued on the bus below this cache. The *PURGE* request propagates down the hierarchy (similar to an *INVALIDATE DOWN* request) invalidating the block wherever it appears. A *PURGE* request may not be canceled.

If the status is DIRTY SHARED, the action taken is a combination of what is done for the DIRTY OWNED and READ SHARED. A *PURGE* request is enqueued on the bus below the cache, and the block is written back to the next higher level in the hierarchy.

The case where the block is in state VALID EXCLUSIVE is more complicated than the previous cases because either the block doesn't exist below this cache, or the block exists in a clean or non-clean state, either in a processor cache or a lower level non-processor cache. Because the block may exist in a non-clean state, the replacing cache must delay its transition from VALID EXCLUSIVE to INVALID until it can be sure that either the block will eventually be INVALID in all caches below it, or that the non-clean block has made its way back up the hierarchy, past the replacing cache and has been written back to the next higher level. To do this, the replacing cache issues a *PURGE REPLY* request on the bus below it and changes its state to TRANS INVALID. The replacing cache delays its transition to INVALID until it sees either a reply to its *PURGE REPLY* on that bus, or the block itself being written back. Because the block might not exist below any given non-processor cache which has a block in VALID EXCLUSIVE mode, the replacing cache will watch the bus for one cycle after the *PURGE REPLY* it transmitted. If no further traffic is observed on that cycle, then the upper cache assumes that the block doesn't exist below it and forwards a reply back up the hierarchy if the upper cache didn't originate the *PURGE REPLY*. Thus, a *PURGE REPLY* proceeds down the hierarchy from the replacing cache (or its descendants) until one of three things happens:

- The block doesn't exist at all below a certain level. On this level, the upper cache is the lowest cache in the hierarchy to have the block. This cache returns a reply message up the hierarchy to the replacing cache.

- The block exists in one of the dirty states (DIRTY, DIRTY OWNED, or DIRTY SHARED) in a cache at some level lower in the hierarchy. In this case, the cache that has the block in the dirty state forms a reply to the *PURGE REPLY* that includes the contents of the block and indicates that the block needs to be written back.

- The request reaches the bottom level of the hierarchy. In this case, if the block doesn't exist in the DIRTY state in one of the processor caches, the low-level cluster bus cache returns a reply message.

Note that a non-processor cache must acknowledge a *PURGE REPLY* if it has the block, since lack of such reply indicates that the block is not cached below.

There is an additional wrinkle that can occur when a *PURGE REPLY* request is serviced. First, if one of the lower caches has the block in a transition state, the

PURGE REPLY will be canceled and returned to the originating cache. This can occur if a request has come from lower in the hierarchy, up to this level, then back down again. Since the true value of the block may be in motion, the *PURGE REPLY* is returned so that the complex case of canceling a successful request and reading the value from a bus request buffer doesn't occur.

Chapter 6

HIERARCHICAL SUBBLOCK PROTOCOL

In this chapter, we evaluate the effectiveness of the hierarchical architecture as described in the previous chapter. This evaluation is done under different protocols:

- The base protocol as described in Chapter 5.

- A variation on the subblock protocol described in Chapter 3

The motivation for the second set of experiments is that the subblock protocol was found to be successful when applied within a cluster. We wanted to investigate its effectiveness when applied in a hierarchical environment at both the intra and inter-cluster levels. In addition, we wished to investigate the effects of snarfing on the hierarchical architecture, since it is a useful technique for increasing performance, especially when false sharing is present.

This chapter is organized as follows. We describe the subblock protocol as applied to a two-level hierarchical structure in Section 6.1. Results of the simulations under the various protocols are given in Section 6.2. Section 6.3 summarizes our results and draws some conclusions.

6.1 Subblock protocol for two level architecture

When we expanded the single bus subblock protocol to a hierarchy of busses, we tried to make a minimal amount of changes to the single bus protocol in order for it to work at the cluster level in a hierarchical system. At the same time, we based the upper level subblock protocol on the "standard" hierarchical protocol developed in Chapter 5. One simplifying assumption we made is that, unlike the standard coherence protocol, the hierarchical subblock protocol was designed to work with two-level systems only. We also implemented snarfing (as outlined in Section 3.4 at both levels of the system.

6.1.1 Upper level protocol

As stated above, the upper level protocol closely resembles our standard hierarchical protocol. The major difference is the addition of sectors to the bus cache which record the state of the bus cache subblocks. Unlike the lower level protocol, the block state merely describes what type of subblocks may be found in the block. The block state makes it simpler to preferentially replace blocks that will not require a write back. Here are the three block states:

INVALID All subblocks are *Invalid*.

CLEAN SHARED All subblocks are either *Clean Shared* or *Invalid*. Such a block can be replaced without causing a write-back.

DIRTY SHARED Subblocks of any type may be found in a block which is in this state.

There are four subblock states. Given an unlimited number of state bits, we would have chosen to use 5 subblock states, corresponding to the 5 upper level states in the standard hierarchical protocol. Concerns about the number of state bits required forced us to discard a state. Of the five states, the choice came down to discarding either the *Dirty Owned* or *Dirty Shared* state. Discarding the *Dirty Owned* state would mean that dirty subblocks written back from the processor level to the bus cache level would have to be placed in the *Dirty Shared* state, which implies that other valid copies of the subblock may exist in the upper level of the system. Recall that having a *Dirty Owned* state means that if a dirty subblock is replaced in a processor cache, but is later written by a processor in the same cluster, then no upper bus transaction is necessary, which is not the case if the *Dirty Shared* state had to be used. Because upper bus bandwidth is usually the bottleneck in hierarchical bus systems, it should be saved in preference to saving lower bus bandwidth. Hence, we decided to discard the *Dirty Shared* subblock state.

Discarding the *Dirty Shared* subblock state required some changes to the lower level protocol. These will be detailed in the next section. Because a subblock in a bus cache cannot be both dirty (with respect to memory) and shared, dirty (and owned) subblocks are written back to memory when they are read by a processor in a foreign cluster.

Table 6.1: Allowable states for cluster subblock

Bus Cache Subblock States	Processor Cache Subblock Status			
	Invalid	*Clean Shared*	*Dirty Shared*	*Dirty*
Invalid	OK	Error	Error	Error
Clean Shared	OK	OK	Error	Error
Valid Exclusive	OK	OK	OK	OK
Dirty Owned	OK	Error	Error	Error

With this modification, the four upper level subblock states become:

Invalid The subblock is not valid.

Clean Shared The subblock is valid. It may be present in other bus caches and/or the cluster below

Valid Exclusive The subblock is valid, but the bus cache may not have an up-to-date copy of the data. A request must be sent to the cluster below, where the subblock might be present.

Dirty Owned The subblock is valid, and the bus cache has an up-to-date copy. The subblock is not valid anywhere else in the system.

A bus caches always maintains the inclusion property [BW88] with regard to the caches beneath it in the hierarchy. Table 6.1 shows the allowable cluster subblock states, given the bus subblock state.

Lower level protocol

The lower level protocol is very similar to that of the single bus subblock protocol as detailed in Chapter 3. Both the block and subblocks states are the same, as are most of the protocol actions. The significant difference between the two has to do with the fact that the bus caches in the hierarchical system do not have a *Dirty Shared* subblock state, as explained in the previous section. This difference manifests itself when a *Dirty* or *Dirty Shared* subblock is accessed by another processor located either in the same cluster as the modified subblock, or a remote cluster. We show how a few key situations are handled (for complete details of the protocol, see Appendix D).

- Remote read to a *Dirty* or *Dirty-Shared* subblock — The subblock is written back to memory as it goes over the top bus. Since all remote operations must go over the top bus, which is connected to memory, the write back does not cost extra bus cycles. (Figure 6.1)

- Remote write to a *Dirty* or *Dirty-Shared* subblock — The remote writer takes responsibility for writing the subblock back. (Figure 6.2)

- Write back of a *Dirty-Shared* subblock — Because it is possible that the subblock is shared by other caches in the cluster, the block cannot simply just be written into the bus cache in the state *Dirty Owned* state. Rather, other caches check to see if they posses a valid copy of the *block*; if so, then one cache (if need be) reads the *subblock* and sets its subblock state to *Dirty Shared*. (Figure 6.3)

- Read or write of a subblock *Dirty Owned* by the cluster's bus cache — Because we have no *Dirty Shared* bus subblock state, ownership of the subblock has to be transferred: either up, to memory; or down, to the processor's cache. Because we wished to conserve upper bus bandwidth, we chose to transfer ownership back down to the processor cache level. This is an obvious choice in the case where a subblock is being written, but not if the subblock is being read. If the subblock is being read, it goes into state *Dirty Shared*. (Figure 6.4)

6.2 Evaluation of the hierarchical architecture

We used the same basic methodology as for the single bus protocol in Chapter 3. The subblock size was 8 bytes, and the processor caches were 8K, 2 way set associative, while the bus caches were 4 way set associative. The subblock protocol used 64 byte blocks, hence there were 8 subblocks per block. We used both 32K and 128K bus caches. We calculated our speedups using a system with a single processor in a single cluster as the reference, using a 128K bus cache and an 8 byte block size. We simulated ten different combinations of processors and clusters, ranging in size

Legend:

DS = Dirty Shared DO = Dirty Owned

VE = Valid Exclusive Inv = Invalid

* = Contains up-to-date data in subblock

(Note: Shown are *subblock*, not block states)

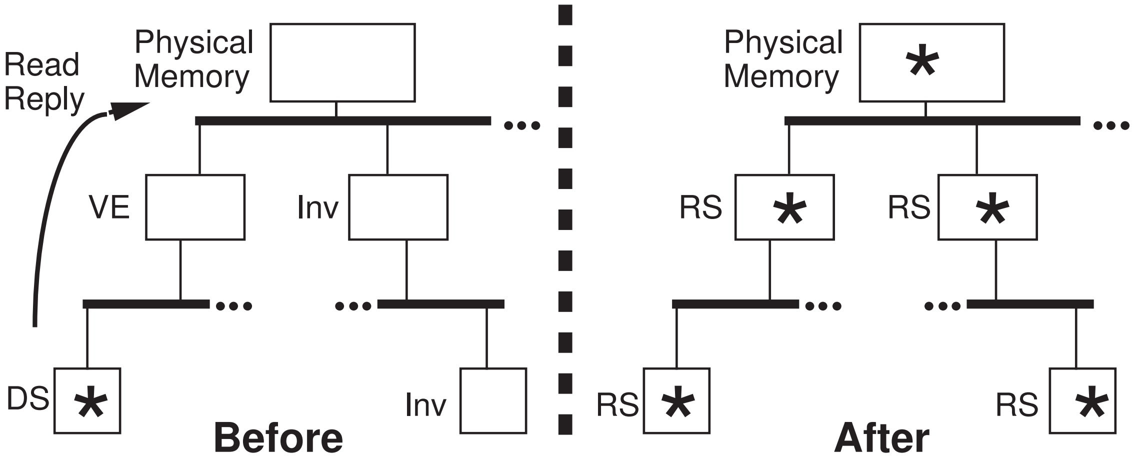


Figure 6.1: Protocol Example #1, Remote Read to Dirty-Shared Subblock

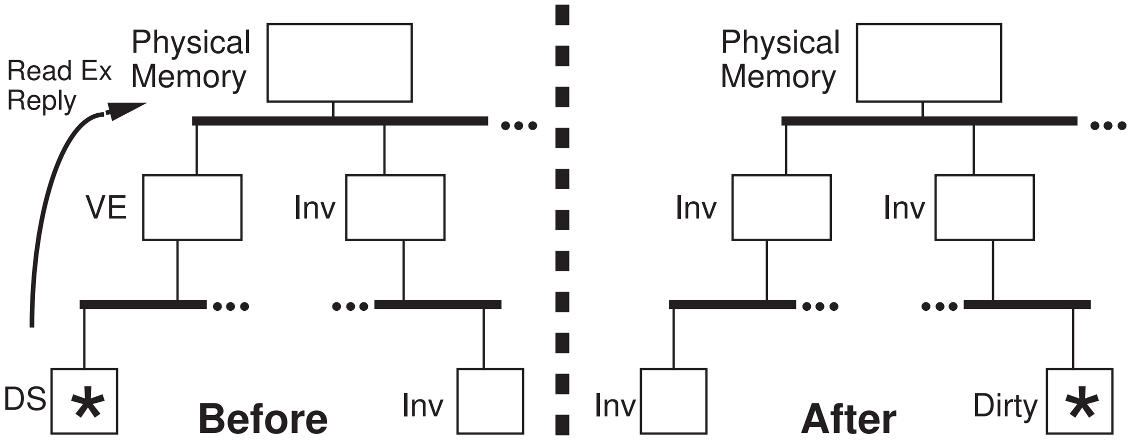


Figure 6.2: Protocol Example #2, Remote Write to Dirty Shared Subblock

Legend:

DS = Dirty Shared — DO = Dirty Owned

VE = Valid Exclusive — Inv = Invalid

* = Contains up-to-date data in subblock

(Note: Shown are *subblock,* not block states)

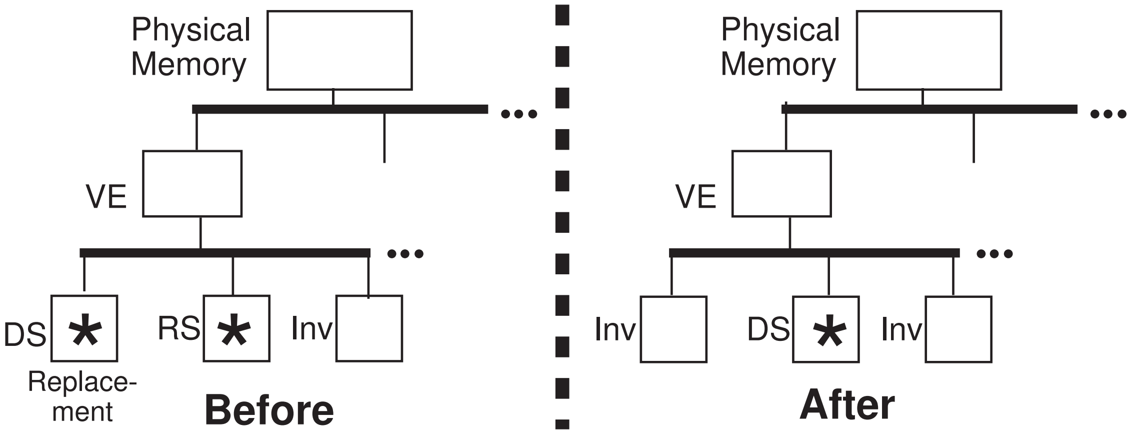


Figure 6.3: Protocol Example #3, Write Back of a Dirty Shared Subblock

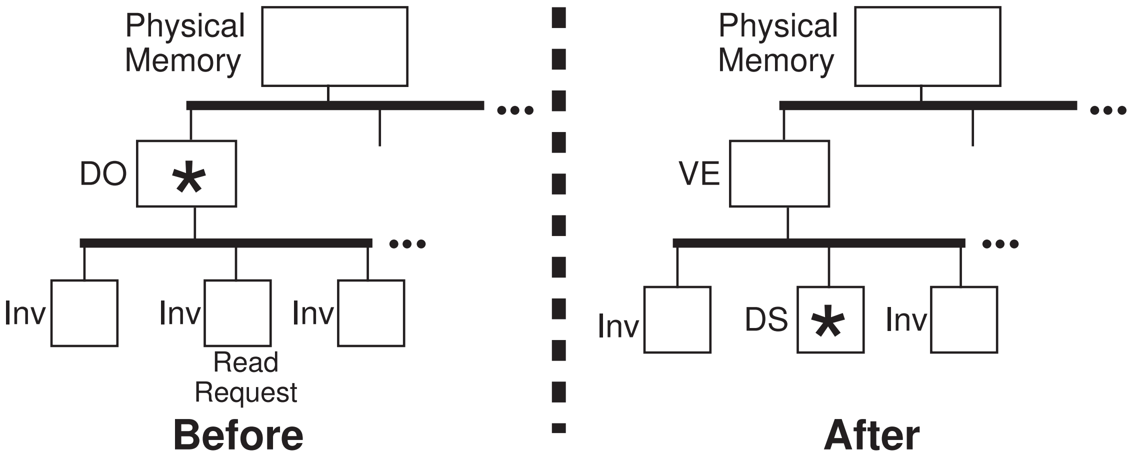


Figure 6.4: Protocol Example #4, Read of a Dirty Owned Subblock

(and cost) from the 2 clusters of 2 processors (2X2) configuration to the 8 clusters of 4 processors configuration. Two of the organizations (2X8 and 2X16) were very unbalanced because they have many processors per cluster and few (i.e. 2) clusters. Not only must the cluster bus support the traffic of 8 or even 16 processors, but also the processors will suffer more cache misses when the sum of the sizes of the processor caches approaches or exceeds (in the 32K bus case) the size of the bus cache, possibly causing many forced replacements as the bus cache maintains inclusion. Nevertheless, we still include those configurations in our results.

In the following sections, we give detailed results for the six benchmarks we studied. For each benchmark, we first report and discuss results when applying the standard hierarchical protocol of the previous chapter. We then consider the architecture with the subblock protocol and with snarfing. In the figures that follow, we present the ten configurations from left to right in order of increasing number of processors; and for a given number of processors, in increasing number of clusters. In the legend, the bars are labeled with the block size (for the Illinois protocol) and whether or not snarfing was used. For example, "8I No Snarfing" bar represents the results for the hierarchical protocol when using the small (8 byte) block size without snarfing. The numbers above each bar are the height of that bar normalized to the height of the "8I No Snarfing" bar.

Barnes

Hierarchical protocol

The Barnes-Hut application sped up well on the cluster architecture, achieving near-linear speedup up to 16 processors, and a speedup of about 27 using 32 processors. Because of Barnes' small data set, there was little difference between using the small or large bus cache. And since Barnes did comparatively few shared references, varying the block size had only a minor effect on overall performance (see Figure 6.5). Using the small block size did have a slight (up to 5 percentage points) advantage over using the large block size, particularly when 16 or 32 processors were used.

If we look at the number of upper bus transactions (Figure 6.6), we can see that there were comparatively few transactions on the upper bus. This is reflected in low upper bus utilization, which did not exceed 20% for any configuration (Figure 6.8). Cluster bus utilization was also 20% or below for all configurations, except the 2X16 cluster, where it was 30%. However, the low bus utilization figures can in some cases be misleading. When we examined the statistics for the number of times a bus transaction had to be retried, we found that the number of retried bus transactions was relatively large when a large number of processors were used. This was especially true in the 64 byte blocks case, and is the reason why the 64I protocol had more bus transactions than the 8I protocol for some configurations (2X16 (32K only); 6X4, 4X8, 8X4 (both 32K and 128K)). In contrast, 64I had fewer total cluster bus transactions than 8I, except for the 8X4 configuration, where it used 10% more transactions.

The 64I protocol was also at a disadvantage compared to the 8I protocol for the number of bytes transferred on the upper bus (Figure 6.7). For all configurations and both bus cache sizes, 64I transferred between 5 and 7 times the data that the 8I protocol did, which indicates that processors were not using most of the data in the 64 byte block being transmitted on the upper bus. The situation was not quite as bad for 64I on the lower level busses; there, 64I transferred about 3 times as much data as 8I did. False sharing was not responsible for the increased traffic, since using the large block size did not increase invalidation traffic on either the upper or cluster busses.

Though 64I did poorly when compared to 8I on bus metrics, it virtually equalled 8I performance when 16 or fewer processors were used, and suffered at most a 5% decline in performance when more processors were used. The reason why the negative effect was small to non-existent is that Barnes did relatively few shared references, hence memory system performance was not critical to good application speedup.

Subblocks and snarfing

Because of the last observation, running the subblock protocol had very little effect on program performance. In some cases, using subblock increased performance by 1–2%, while in others it had the same speedup as 8I. Only on the 8X4 configuration was there a tiny (1%) drop in performance. The subblock protocol had a larger impact on various bus traffic metrics. Except for the 2X2 and 2X4 organizations, the subblock protocol used fewer upper bus transactions than either the 64I or 8I protocols. The reduction below 8I's bus transactions ranged from about 10% to 30%. Subblocking transferred far fewer bytes on the upper bus than the 64I protocol, but transferred

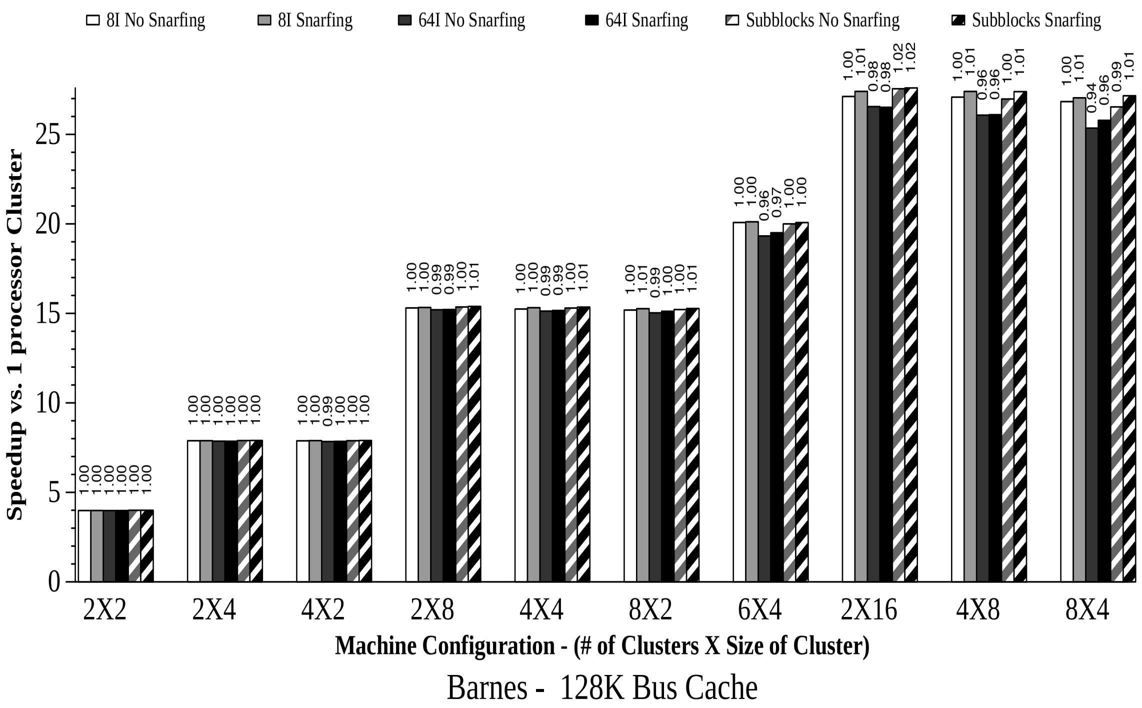


Figure 6.5: Barnes speedups, 128K bus caches

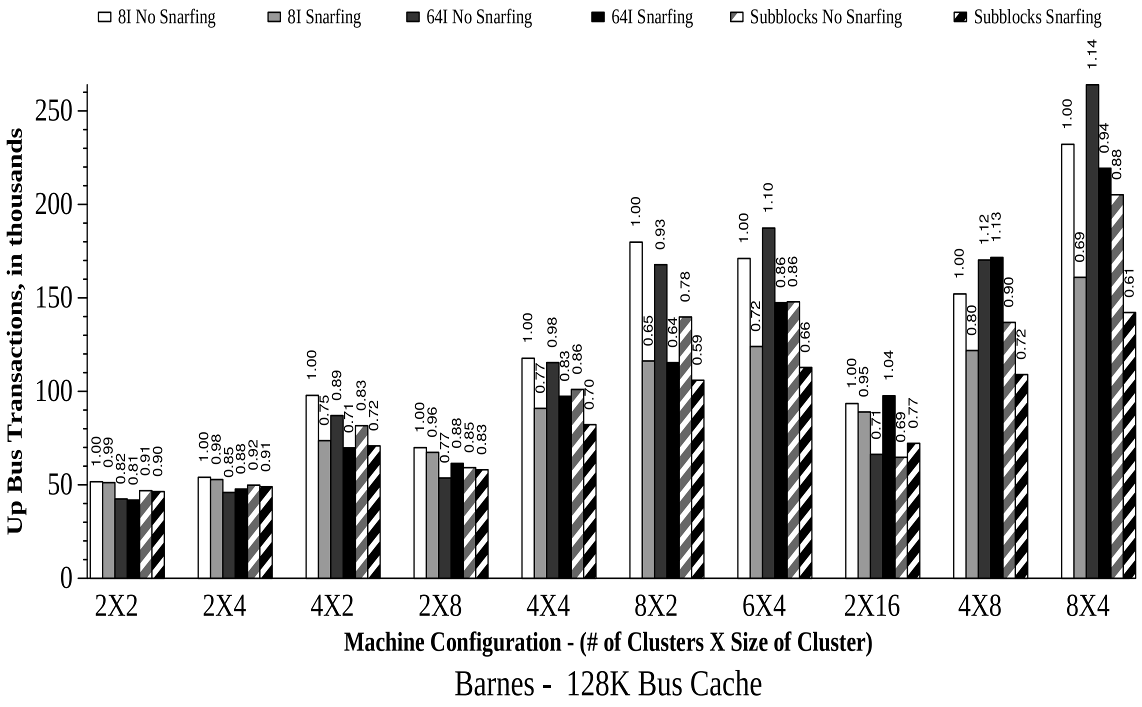


Figure 6.6: Upper Bus Transactions for Barnes

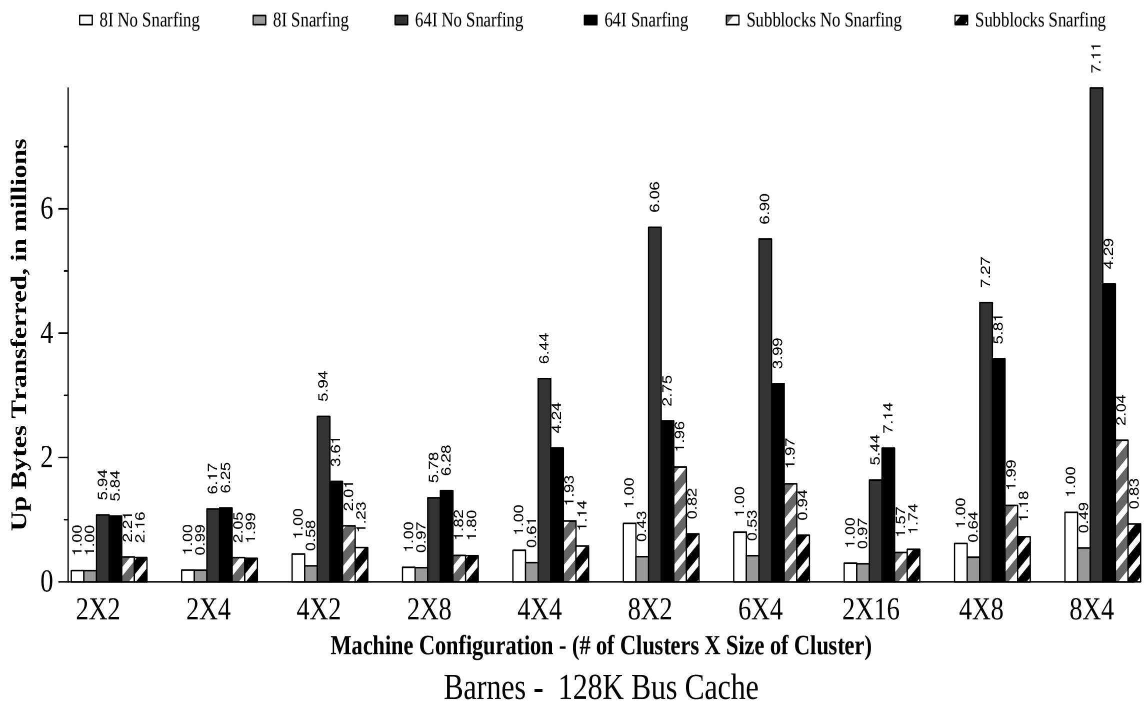


Figure 6.7: Upper Bus Bytes Transferred for Barnes

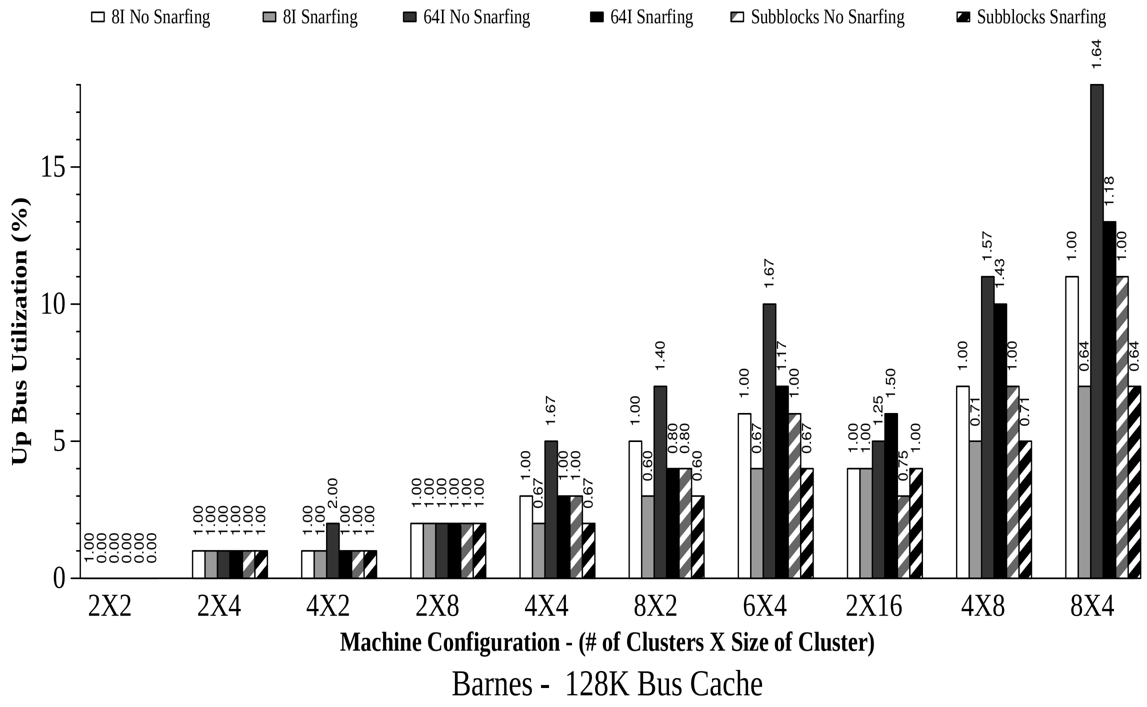


Figure 6.8: Upper Bus Utilization for Barnes

about twice the data that the 8I protocol did. The subblock protocol transferred more data than the 8I protocol because of its policy of transferring more than just the requested subblock. Subblocking also reduced the total number of cluster bus transactions below that of either 8I or 64I when 32 processors were used, and when 4 or more clusters were used with 16 or 24 processors.

Snarfing had a minor positive impact on program performance, but had a large effect on upper bus traffic, and a smaller effect on cluster bus traffic. Snarfing's effect on upper bus transactions was largest when there were more bus caches, which is to be expected since snarfing occurs more often when there are more caches connected to a bus. Snarfing aided all three protocols in reducing the number of bus transactions. Snarfing also substantially reduced the number of bytes transferred on the upper bus when 4 or more clusters were used. In certain cases when only 2 clusters were used, snarfing increased the number of bytes transferred. It is difficult to determine exactly why this occurred, but it appears that snarfing increased contention for certain blocks, especially blocks involved with synchronization operations. Snarfing also decreased the number of bus transactions and bytes transferred on cluster busses, but to a lesser degree than what happened on the upper bus.

In summary, both the subblock and snarfing techniques had substantial beneficial effects on upper bus traffic, and to a lesser extent, cluster bus traffic. Because Barnes made comparatively few shared memory references, these improvements did not decrease Barnes' execution time to a large degree.

Gauss

Hierarchical architecture

As we have seen in Chapter 3, Gauss performs best when using a large block size because of its spatial locality. This is quite evident from Figure 6.9. Though 8I's speedup didn't get much above 10 for any configuration, the 64I protocol achieved excellent speedup on nearly all configurations; in some cases, an apparent superlinear speedup was obtained because the additional processors and bus caches could hold more of the working set, and because the single processor cluster used for the comparison used an 8 byte line size. Not surprisingly, those configurations that had more clusters had better speedups than those with few clusters, though the differences were small except for the 2X16 configuration. Both the 2X8 and 2X16 organizations had very high lower bus utilization rates, which led to a drop-off in performance.

The reason for 64I's superior performance can be seen in Figure 6.10, the graph of the number of transactions on the upper bus. Since each transaction under 64I transferred eight times as much data as the 8I protocol, it used far fewer transactions to move data around. Using the larger block size reduced the number of transactions by 85–86% across all configurations and both sizes of bus cache. An interesting observation is that increasing the bus cache size from 32K to 128K did not markedly reduce the number of bus transactions for either block size. The lone exception to this was the 6X4 combination, which halved the bus transaction when the bus cache size was increased. The 6X4, 128K bus cache configuration had many fewer spills from it bus caches, and its miss rate was about 1/2 (or better) that of any other configuration. We are investigating why the 6X4 had fewer (presumably conflict) misses than the other organizations.

If we examine the graph for the number of bytes transferred on the upper bus (Figure 6.11), we see that using the large block size increased the number of bytes transferred by only 9–15% across all configurations, indicating that most of the data transferred in each 64 byte block was needed (sooner or later) by the processor, and that false sharing was not a problem. There was little variation in the amount of bytes transferred when 16 processors or less were used. Just as the 6X4 configuration had fewer upper bus transactions, it also transferred many fewer bytes when used with 128K bus caches. All three 32 processor organizations had fewer bus transactions than when fewer processors were used, due to lower miss rates, both at the processor and bus cache level.

Subblocks and snarfing

The subblock protocol performed quite well on Gauss. For all configurations and both sizes of bus cache, the subblock protocol equalled or slightly exceeded the already excellent performance of the large block Illinois protocol. The lone exception to this was the (unbalanced) 2X16, 128K bus cache organization where performance dropped by 1%. Like the case with single bus machines, the subblock protocol's policy of using large transfers worked well in applications (like Gauss) with good spatial locality. The number of upper bus transfers under the subblock protocol was nearly identical to the number used by the 64I protocol, and the number of upper bus bytes transferred was

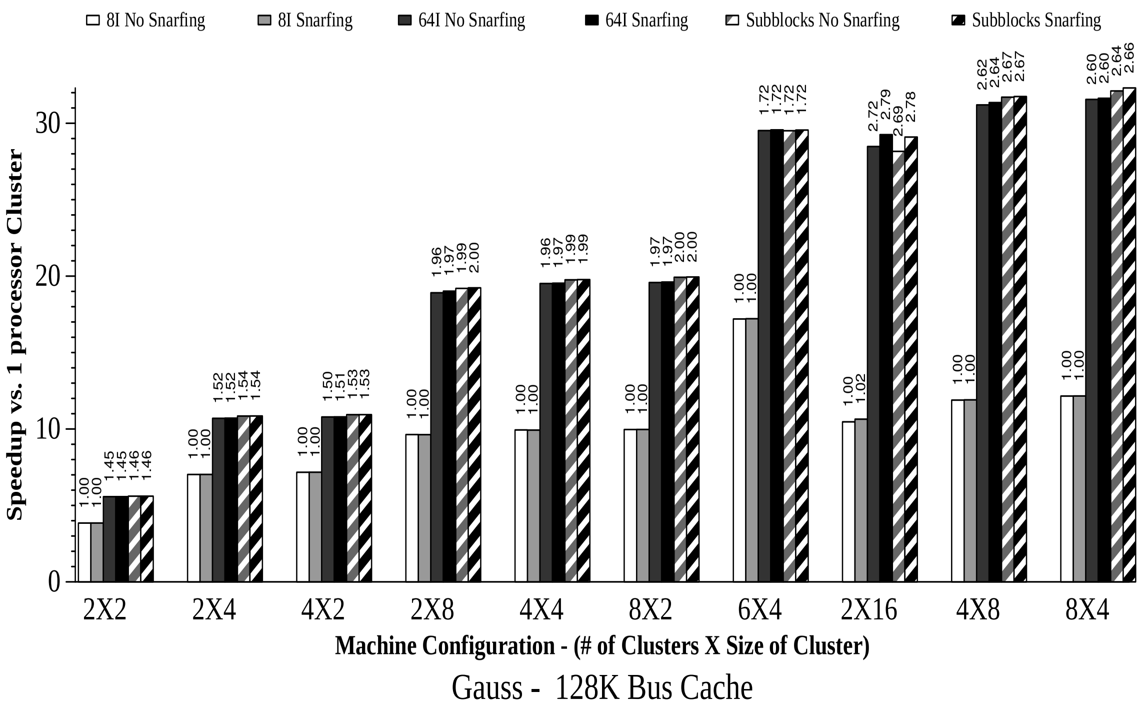


Figure 6.9: Gauss speedups

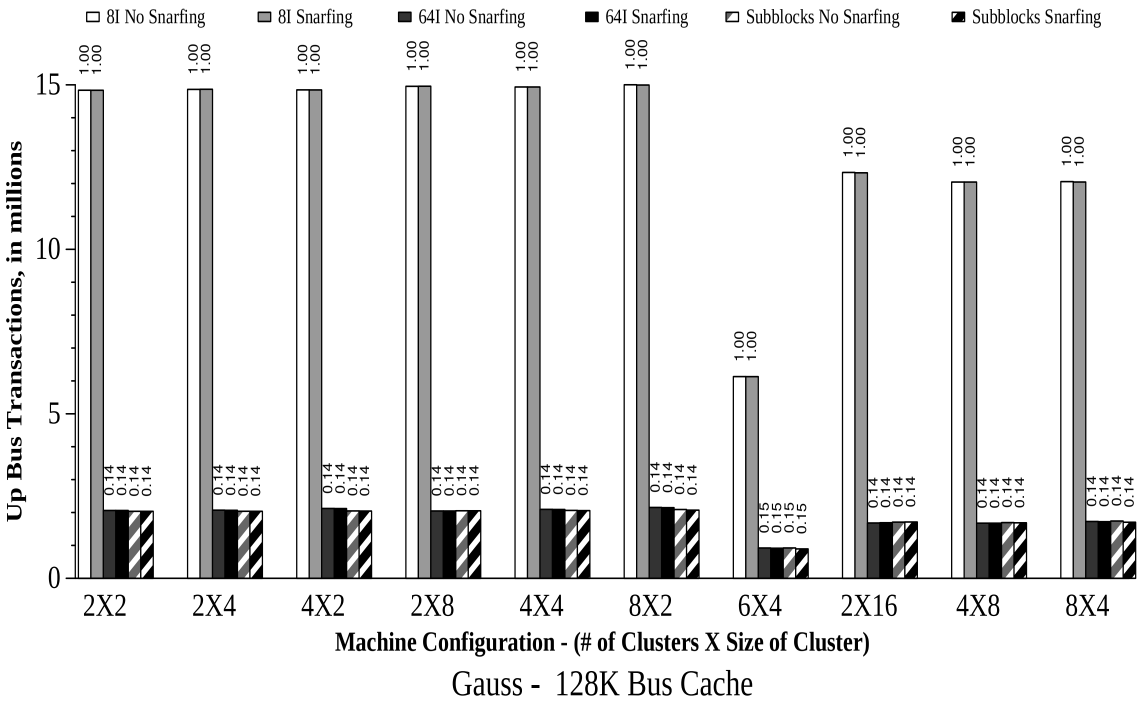


Figure 6.10: Upper Bus Transactions for Gauss

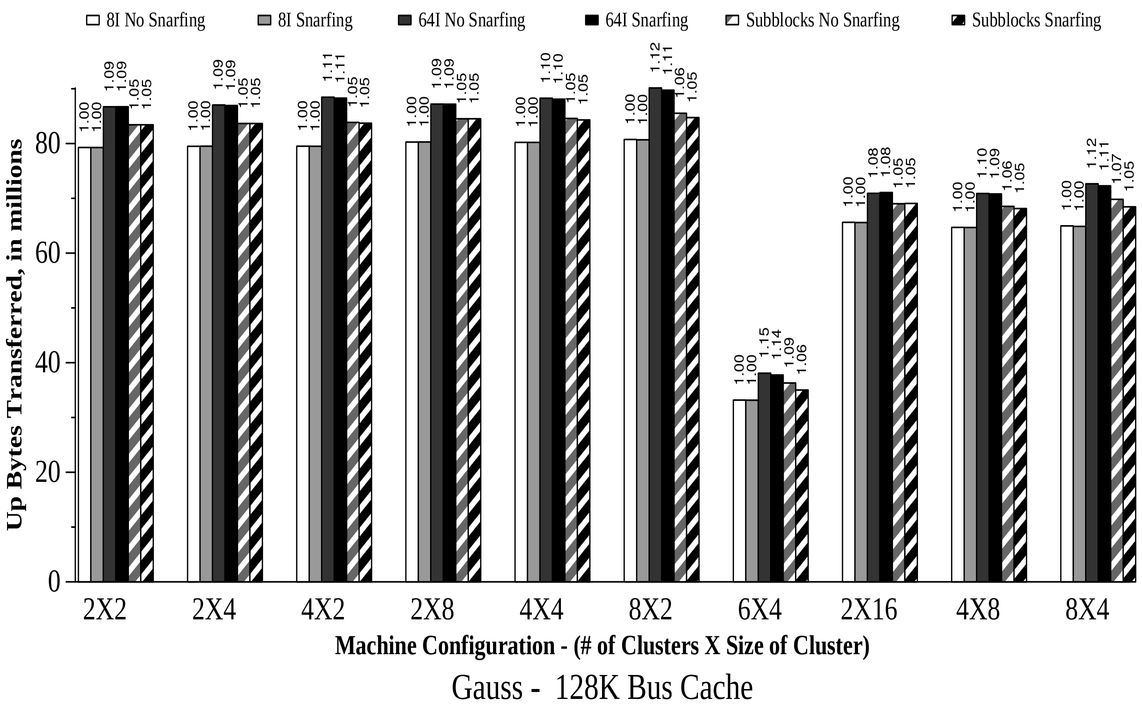


Figure 6.11: Upper Bus Bytes Transferred for Gauss

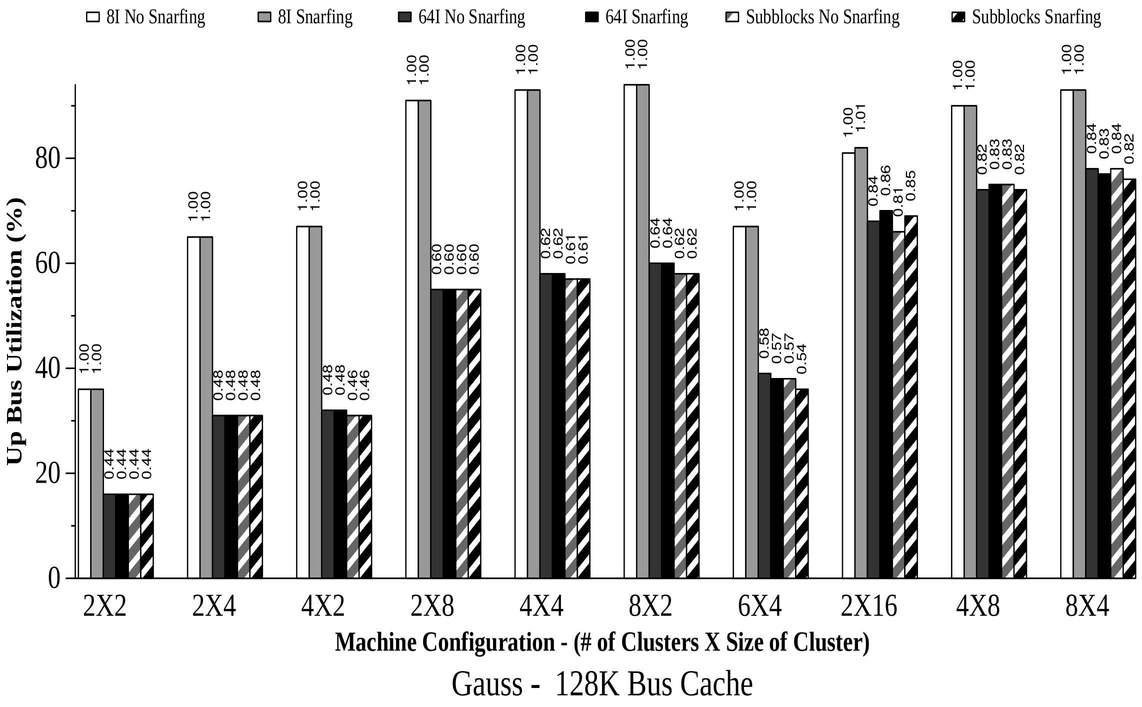


Figure 6.12: Upper Bus Utilization for Gauss

less than the 64I protocol (but still more than the 8I protocol). The subblock protocol also had lower miss rates in the bus cache than either the 8I or 64I protocol. This was especially true when using the large bus cache and larger number of processors. Unfortunately, this was offset by the subblock protocol's average processor cache miss rate, which was higher than that of the 64I protocol, but still much less than that of the 8I protocol. The higher miss rates are likely due to the subblock protocol not transferring all subblocks in a block in response to a read request because some of the non-requested subblocks were exclusive in the responder's cache.

Snarfing had little impact on Gauss's performance because there were few opportunities where it could have had a beneficial effect. Snarfing has the most benefit when many blocks are being shared and written by many processors. This does not occur in Gauss. Among all configurations, snarfing had the biggest impact on the 2X16 organization when 32K bus caches were used — it improved performance by 12% for 8I, 7% for 64I, and 9% for the subblock protocol. All other improvements were in the 1–2% range. Snarfing had no real effect on the number of upper bus transactions, and very small effect on the number of upper bus bytes transferred. Snarfing had more effect on the lower bus, where it cut the number of bytes transferred by 5–15%, with the greatest reductions occurring when larger numbers of processors were used.

Cholesky

Hierarchical architecture

Like Gauss, Cholesky performs best when using a large block size. Unlike Gauss, Cholesky did not achieve near linear speedup in our runs. The major reason for this is the size of the input matrix — it has limited available concurrency [SWG92]. Maximum speedup of about 17.5 was obtained using the 64 byte block size and 32 processors (in the 8X4 or 4X8 organizations) and 128K bus cache size. The advantage of using 64I over 8I was largest when the small bus cache size was used. The speedup increase ranged from 50–200% when 2 clusters were used, and from 23–74% for 4 or more clusters. The increases were much less dramatic when the 128K bus caches were used — the range was 12–30%, except for the 2X16 case which posted a 50% gain. Generally, the differences were largest when there were more bus transactions, either because of small bus caches or larger numbers of processors.

The improved performance of 64I over 8I was due to the reduced number of bus transactions needed by the 64I protocol. The 64I protocol reduced the number of transactions on the upper bus by 76–82% across all configurations and both bus cache sizes. Using the large block size reduced the total number of bus transactions on the lower busses as well, by 71–82%. These reductions occurred because Cholesky's algorithm usually accesses several contiguous elements of the arrays that represent the matrix; hence reading (or invalidating) 64 bytes at a time (rather than 8) saves many bus transactions. However, it was not the case that every time a processor read (or wrote) a datum that it needed all 64 bytes that came with the block. This can be seen in the graph for the number of bytes transferred (Figure 6.15). The 64I protocol transferred 48–70% more bytes than the 8I protocol when 128K bus caches were used. The figures are even higher (50–86%) when the small bus caches were used. The additional bytes transferred had a negative effect on the application's performance; however, the bus cycle savings due to 64I's fewer transactions outweighed the penalty due to the increased data transferred. Like Gauss, the increased data transfer was not due to false sharing, since 64I used 80% fewer invalidates than 8I did.

Subblocks and snarfing

The performance of the subblock protocol was not very good. In nearly all cases, the subblock protocol executed faster than the 8I protocol, but not as fast as the 64I protocol. This is not a surprise, given the subblock protocol's performance on Cholesky was closer to the 8I than the 64I protocol when using a single bus (see page 36). When small bus caches were used, the subblock protocol generally was closer in performance to the 64I protocol, while this was not the case when using the large bus cache size. The reason for this, and the reason why the subblock protocol didn't perform better involves the nature of Cholesky's data access pattern to the array of non-zero matrix elements. Each group of columns, or supernode, is likely to be modified by several processors during the course of the algorithm. Thus, when a processor modifies the elements of a supernode, it is likely that either the elements are dirty and exclusive in another cache, or have been replaced in the previous writer's cache. Under the subblock protocol, subblocks that are exclusive in one cache are not sent to another cache unless the requesting cache missed on that particular subblock. Hence, if all eight subblocks in a given block are exclusive (and perhaps dirty), another

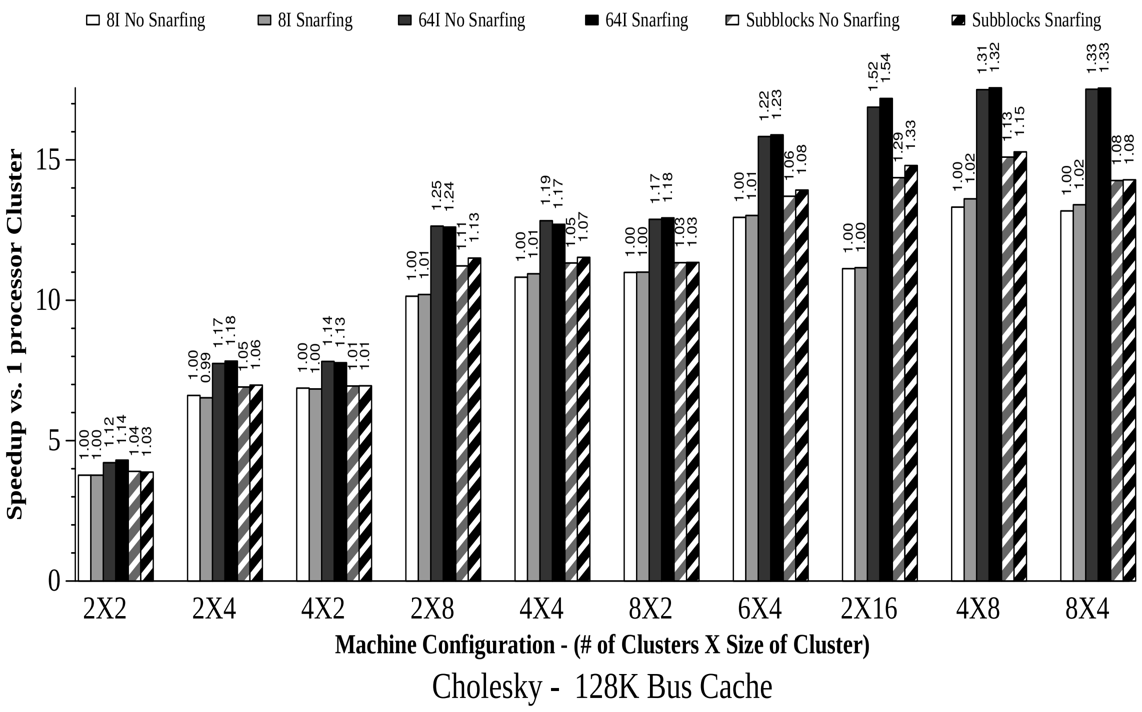


Figure 6.13: Cholesky speedups, 128K bus caches

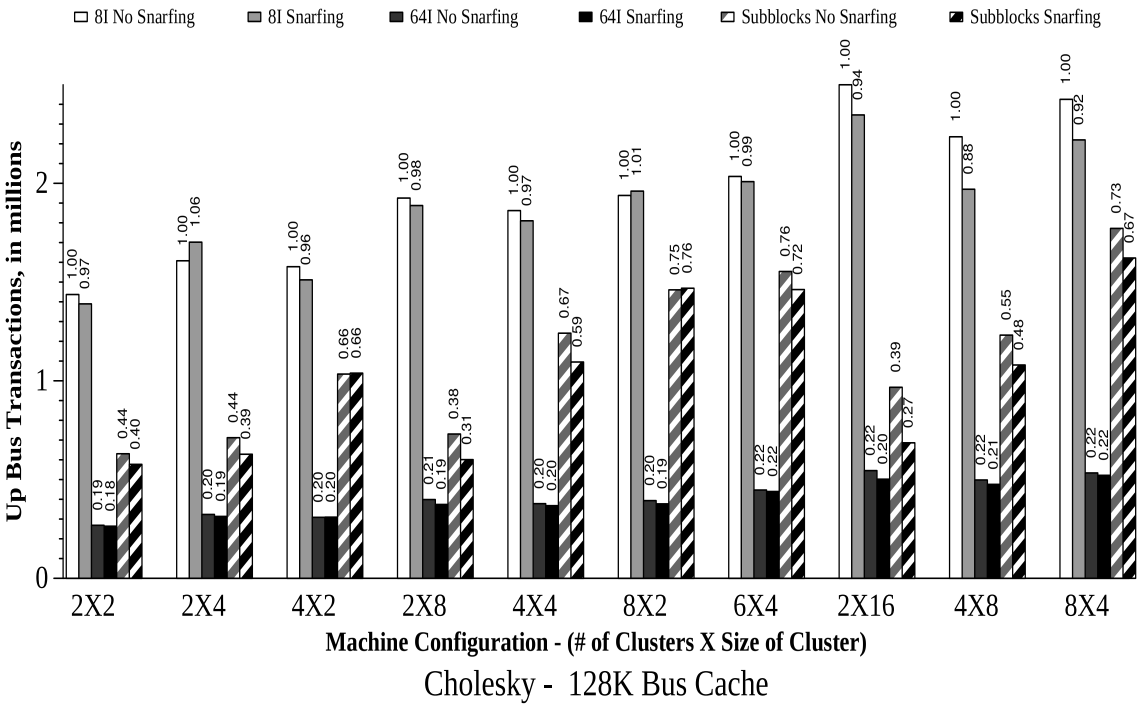


Figure 6.14: Upper Bus Transactions for Cholesky

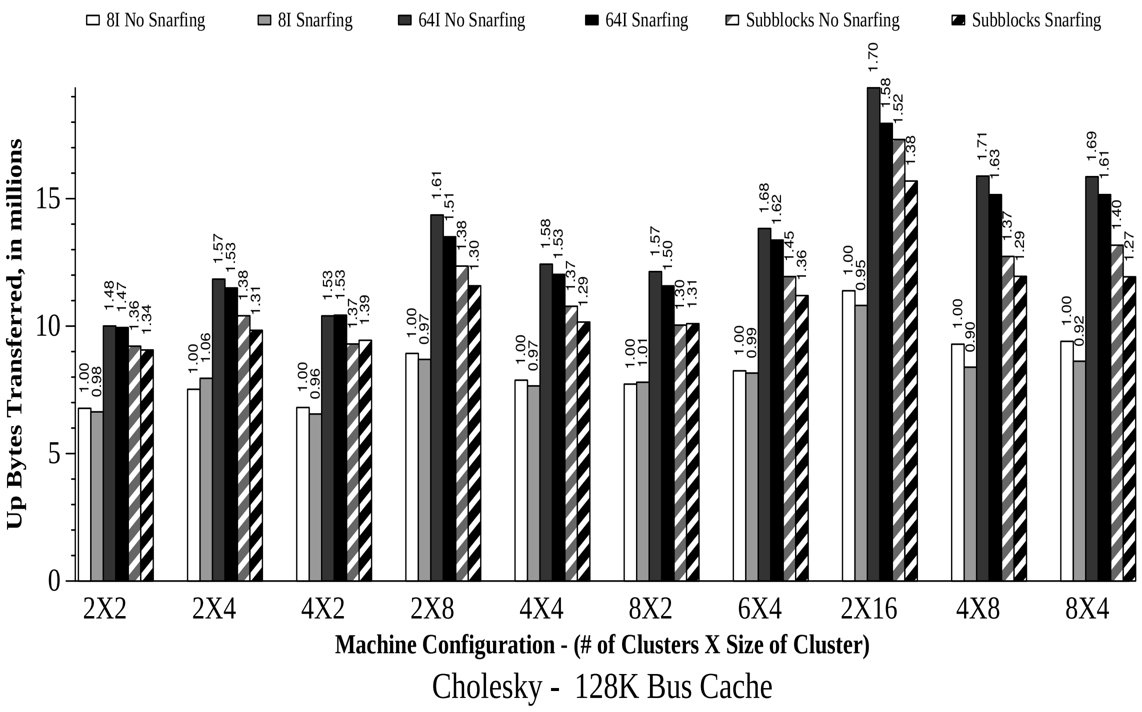


Figure 6.15: Upper Bus Bytes Transferred for Cholesky

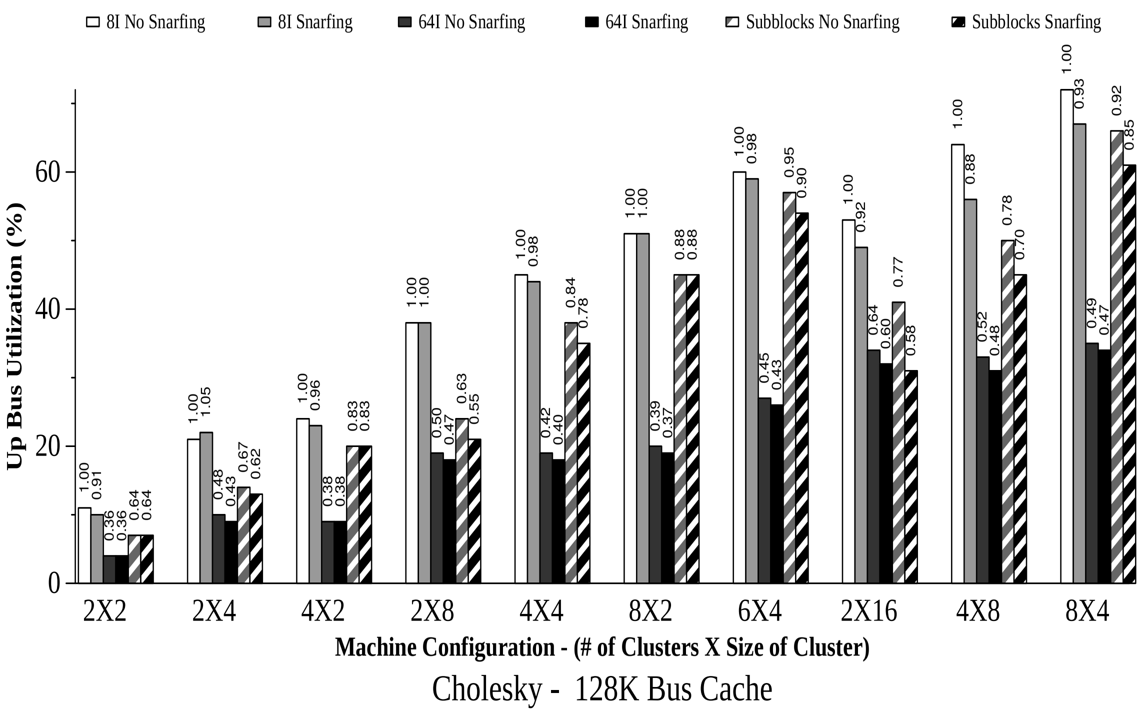


Figure 6.16: Upper Bus Utilization for Cholesky

processor will have to issue 8 separate read requests to read the entire 64 byte block. On the other hand, if the previous writer had to write back the block to memory, then a processor need issue only one read request, since main memory will respond with the entire block. Because there are more block replacements with smaller bus caches, the subblock protocol performed closer to the 64I protocol with smaller bus caches.

If we again look at the number of upper bus transactions (Figure 6.14) we see that the subblock protocol had many more transactions (up 81–275%) than the 64I protocol when using large bus caches. The differences were much smaller (up 5–123%) when the smaller bus caches were used. The subblock protocol did transfer fewer bytes than the 64I protocol, but more than the 8I protocol. Again, the subblock protocol performed closer to the 64I protocol when smaller bus caches were used.

The addition of snarfing to the three protocols had mixed effects. In nearly all the cases, snarfing affected performance by only a few percentage points, either higher or lower. Snarfing was more likely to have a positive impact when using the larger bus cache, and when more bus caches (clusters) were used. However, when snarfing was used on the 2X16 configuration with small bus caches, it had a substantial negative impact on performance on both the subblock and 64I protocols. We examined the detailed statistics produced by each simulation run, and saw that snarfing actually increased hit rates in both processor and bus caches; unfortunately, there were many more processor references under snarfing, so the total number of read and write requests increased, which led to the decrease in performance. Part of the reason why this occurred was a 200% increase in the number of times an atomic swap request had to be retried. This statistic indicates that there was much greater contention for locks (probably the work queue lock) in the program under snarfing.

MP3D

Hierarchical architecture

As was the case when using a single bus, MP3D performed best when using a 64 byte block size. When using the small block size, speedups reached a maximum of about 10, while using a 64 byte block size resulted in maximum speedups of about 15 (cf. Figure 6.17). When 16 or more processors were used, the 64I protocol increased performance from 40% to 80% over the 8I protocol. As stated before, MP3D's sublinear speedup is largely due to a great deal of true sharing. The sharing caused a large amount of traffic on the upper bus, which was saturated when using 16 or more processors. For all but the 2X8 and 2X16 configurations, the upper bus was saturated before the cluster busses were saturated. Increasing the bus cache size from 32K to 128K had little effect on MP3D's overall performance.

The reason for the difference in performance caused by increasing the block size from 8 bytes to 64 bytes can be seen in the graph for the number of upper bus transactions (cf. Figure 6.18). Using the large block size (with 128K caches) cut the number of bus transactions by 62–79% over using the small block size. The reductions were largest when fewer clusters were used. When 32K bus caches were used, the number of transactions used by 8I increased when compared to using 128K bus caches by about 10% across all configurations. Using 64I reduced the number of bus transactions vs. 8I when using 32K bus caches by about the same proportion as for the 128K case.

One disadvantage of using large block sizes is that they sometimes transfer too much information. This can be seen in Figure 6.19, where it can be observed that 64I transferred between 1.7 and 2.7 times as much data as the 8I protocol. Still, these figures aren't bad, considering that the 64I protocol transfers 8 times more data per read transactions as compared to the 8I protocol. Increasing the number of clusters while keeping the number of processors constant caused small increases in the number of bytes transferred under 8I, but caused large increases when 64 byte block sizes were used. Reducing the bus cache size to 32K increased the number of bytes transferred by about 10–20% for all configurations. Like the case with Gauss and Cholesky, false sharing was not a significant problem.

Using large blocks also reduced the total number of bus transactions on the cluster busses as well, though the reductions were smaller (49–71%). Again, the largest reductions occurred when fewer clusters were used. Using 64 byte blocks had a larger effect on the number of cluster bus bytes transferred, where the increases ranged from 2.6 to 3.2 times the number of bytes transferred under 8I.

Despite the additional data transfer caused by using the large block size, overall it performed better because it substantially reduced the number of bus transactions.

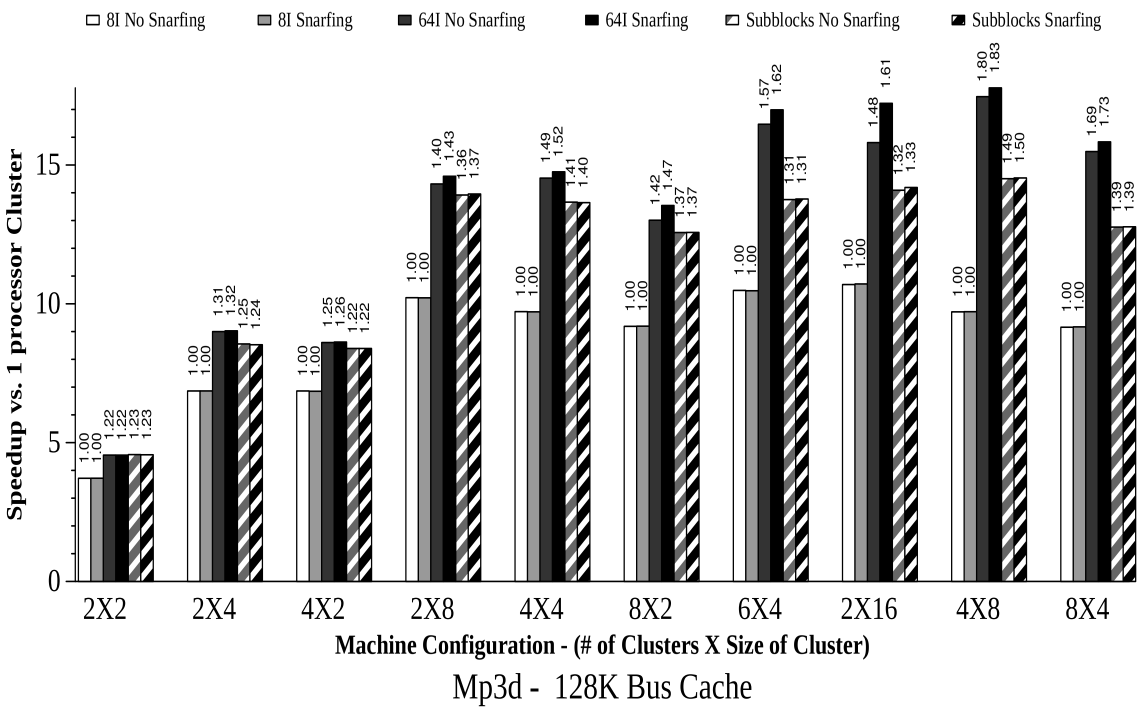


Figure 6.17: MP3D speedups, 128K bus caches

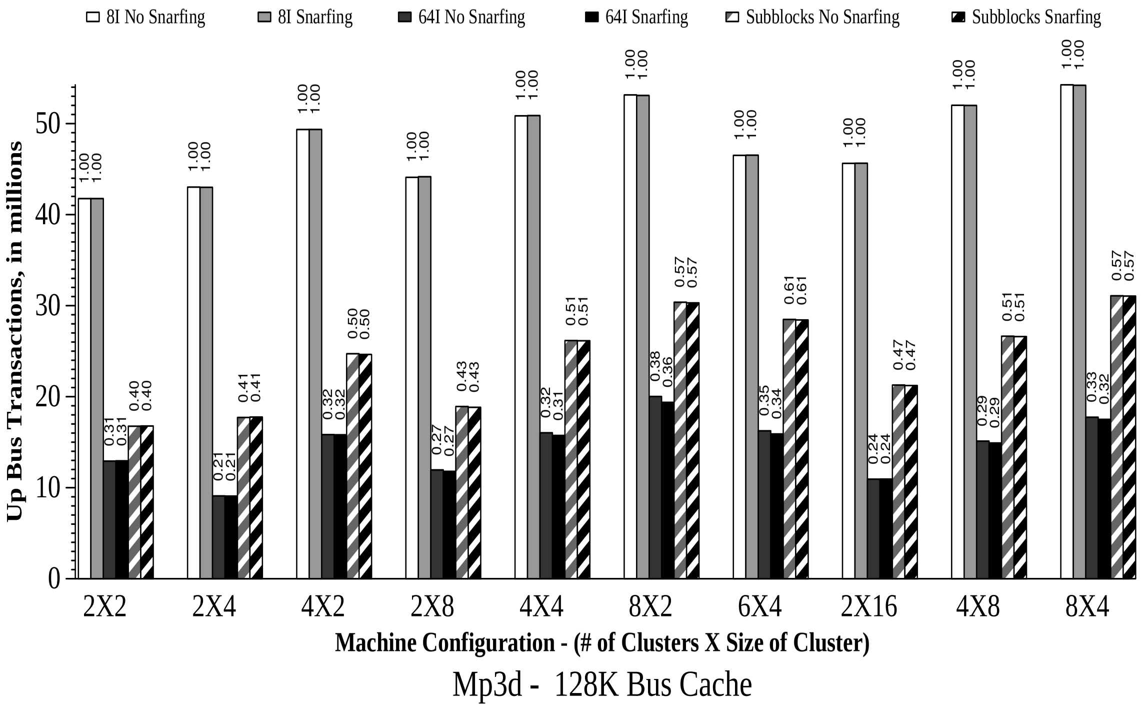


Figure 6.18: Upper Bus Transactions for MP3D

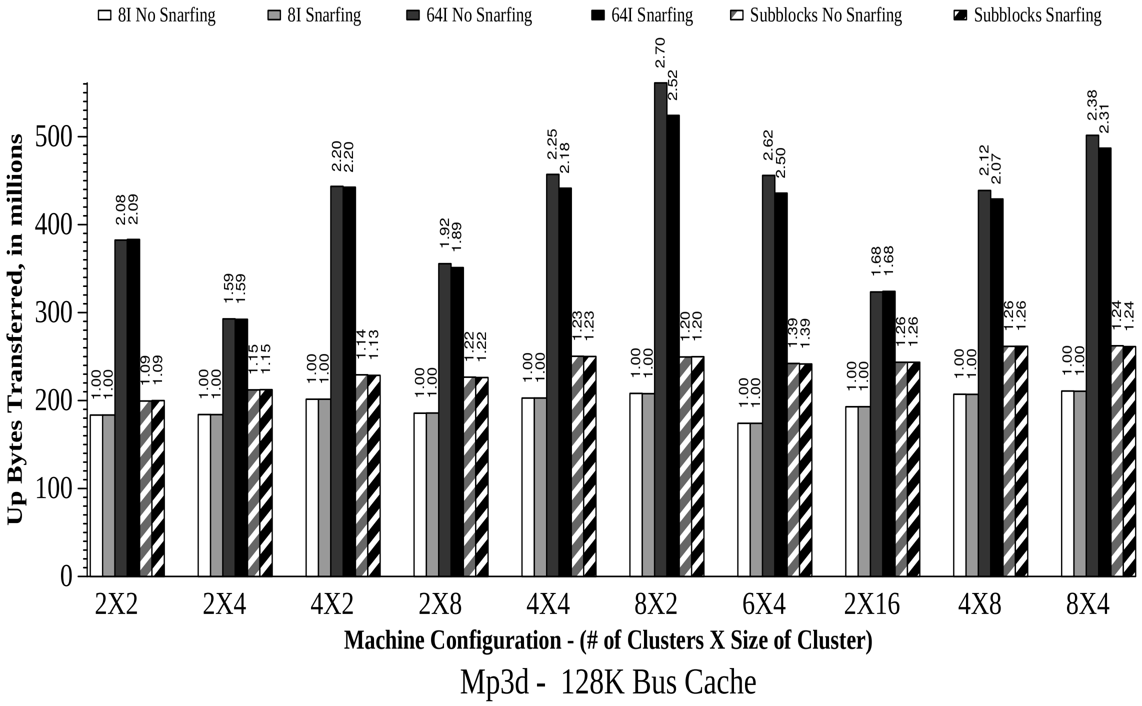


Figure 6.19: Upper Bus Bytes Transferred for MP3D

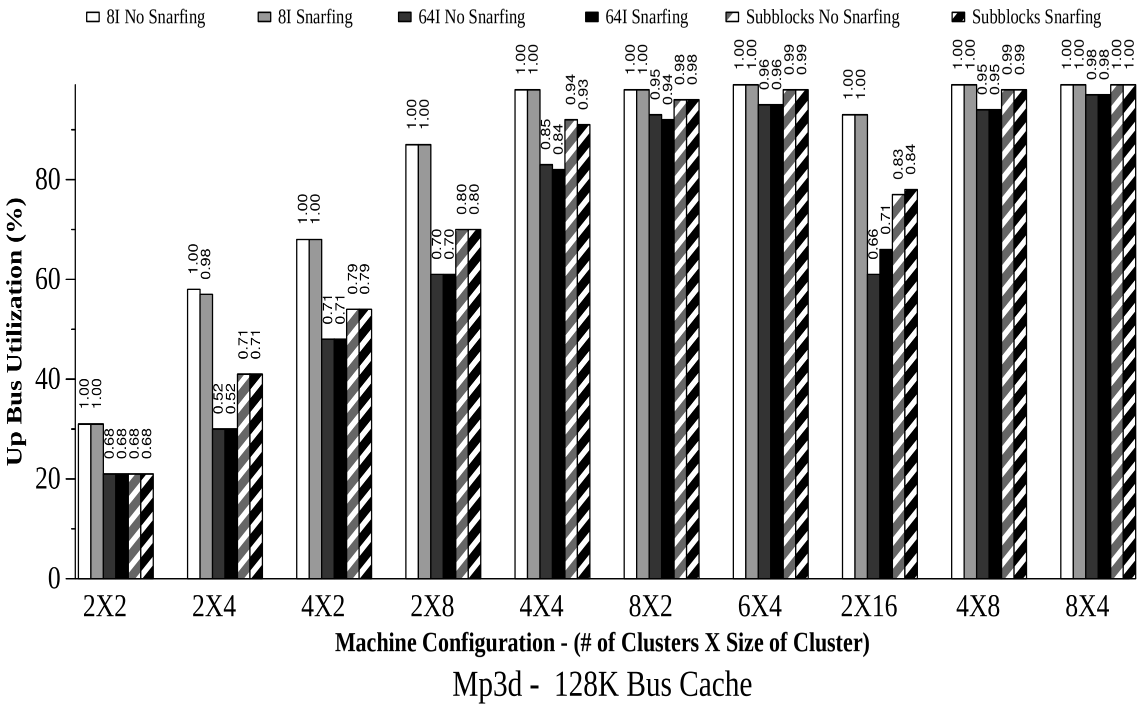


Figure 6.20: Upper Bus Utilization for MP3D

Subblocks and snarfing

The performance of the subblock protocol was, in most cases, between that of the 8I and 64I protocols. In only one case (32K bus cache, 2X8) did it exceed 64I's performance. Subblocking came close to 64I's performance (within 8 percentage points) when 16 or fewer processors were used with both cache sizes. The gap was wider (up to 30 points) when 24 or 32 processors were used. Given these results, it is not surprising that the subblock protocol used fewer upper bus transactions than 8I, but more than 64I. Subblocking used from 39–60% fewer transactions than 8I; unfortunately, it used about 30% (2X2) to 90% (2X16) more transactions than the 64I protocol. These additional transactions directly translated into poorer application speedups. The subblock protocol transferred more data across the top bus than the 8I protocol, but much less than 64I. While 64I at least doubled the number of bytes transferred over 8I, subblocking's increase was from 9% to 39% for 128K bus caches, and from 30–49% for 32K caches. The increase was larger when using 32K caches because of the greater proportion of read transactions on the upper bus due to the relatively small bus caches. Subblocking showed similar results on the cluster bus traffic metrics.

The one metric in which the subblock protocol performed worse than either the 8I or 64I protocols was in average memory queue length. While even in the worst case, the queue length was not large, it indicates that the lack of a *Dirty-Shared* subblock state hurt the subblock protocol's performance.

Adding snarfing to the architecture had little effect on the speedup of most configurations when using either the 8I or the subblock protocol, and small positive effects on speedup when large blocks were used. In most cases, snarfing improved 64I by 2–4% percent when 16 or more processors were used. Snarfing did make a larger difference in the 2X16 configuration, where it raised performance by 9%. Snarfing had similar results when 32K bus caches were used. Snarfing had very little, if any effect on upper bus traffic for any of the three protocols.

Snarfing had more of an effect on cluster bus traffic. When 4 or more clusters were used, snarfing *increased* the number of cluster bus transactions. This happened because of snarfing on the upper level bus. When a block (or subblock) is snarfed into a bus cache, it is placed in the READ SHARED state, which indicates that not only is the block valid and shared, but also indicates that it might exist in the cluster below the bus cache. A subsequent request by a processor in another cluster to write the block results in an invalidate request being sent down to the cluster whose bus cache snarfed the block. Thus snarfing on the upper bus can dramatically increase the number of invalidation bus transactions on the lower level bus. This is what happened when snarfing was used on MP3D. Snarfing increased the number of transactions for all three protocols, and had the largest effect on 8I and the subblock protocol. The effect was worse when 8 clusters and large bus caches were used. For example, snarfing increased bus transactions for the 8X4, 128K configuration by 30% for 8I, 17% for 64I, and 67% for the subblock protocol.

At the same time, snarfing decreased the number of bytes transferred on the cluster busses when 64 byte blocks were used. As expected, the drop was largest when there were a large number of processors per cluster — 13% for the 2X16, 32K bus cache configuration, and 8% using the 128K bus cache. Snarfing had almost no effect on the 8I protocol, and had a very small impact (1–2%) on the subblock protocol.

In summary, MP3D performed best when using large block sizes. Due to large amounts of true sharing, speedups were limited to about 17 when using 32 processors. The subblock protocol performed fairly close to 64I for 16 or fewer processors, but relative performance dropped for 24 and 32 processors. Snarfing had small performance benefits when using 64 byte blocks, but had little effect on the 8I and subblock protocols.

Pverify

Hierarchical architecture

Because of Pverify's false sharing, it was no surprise that the 8I protocol performed better than the 64I protocol, and that the differences between the two increased as the number of processors increased. In addition, Pverify has a large enough working set that using 32K bus caches caused many more capacity/conflict misses than when 128K bus caches were used. These misses, along with the delays caused by waiting for the data to be returned from the memory system slowed down all tested configurations when 32K bus caches were used (Figure 6.21). The maximum speedup achieved by using 32K bus caches was about 21, when 8 clusters of 4 processors and the small block size was used. Increasing the bus cache size to 128K upped the speedup for the 8I, 8X4 configuration to 29. Using the 64 byte block size reduced performance greatly over the 8I protocol. 64I's maximum speedup was 8.3 for 32K bus caches (8X2 configuration) and 15.1 for 128K bus caches (6X4, 8X4 configurations). 64I's speedups were 40–50% below 8I's for the 16 processor, 32K bus cache case; the drop was 60% for 32 processors. The performance drops were smaller when the large bus cache was used: 22–40% for 16 processors, and about 50% for the 4X8 and 8X4 configurations.

The reason for the performance drop when using 64 byte blocks was the additional upper bus traffic caused by block invalidations and rereads due to false sharing. The amount of upper bus traffic was critical to Pverify's performance, since the upper bus was saturated or near saturated when 16 or greater processors were used. This can be seen in Figure 6.25. Upper bus utilization figures were not as high for the 2X8 and 2X16 organizations because the lower busses were saturated by the many processors in each cluster. The effects of false sharing can be seen in Figure 6.23, where 64I consistently required more bus transactions than 8I. The increases ranged from 30% to 50% for 32K caches. The differences were much larger for large bus caches, especially when 16–32 processors were used. The worst increase was 228% for the 4X8 configuration; the "best" was 73% for the 8X2 organization. The differences between 8I and 64I were larger with large bus caches because false sharing misses were a bigger proportion of the total traffic mix when large bus caches were used. The large bus caches reduced the number of capacity/conflict misses reaching the upper bus.

The statistics were far worse for 64I for the number of bytes transferred (Figure 6.24). 64I transferred between 9 and 11 times more data than 8I with 32K caches, and 9 to 25 times more data with 128K caches. Part of the reason for these dramatic increases is the additional reads due to false sharing invalidations. 64I typically had between 1.5 and 2.5 times more upper bus invalidates than 8I. Another important factor is that when a processor read a 64 byte block, it rarely used much more than the 4 or 8 bytes of the entire block. Even when a single processor is used in a single cluster (thus no sharing, false or true), 64I transferred 9 times the data 8I transferred with 32K bus caches, and 12 times the data with 128K bus caches. The additional traffic caused by using a large block size directly caused the poor speedups of the application.

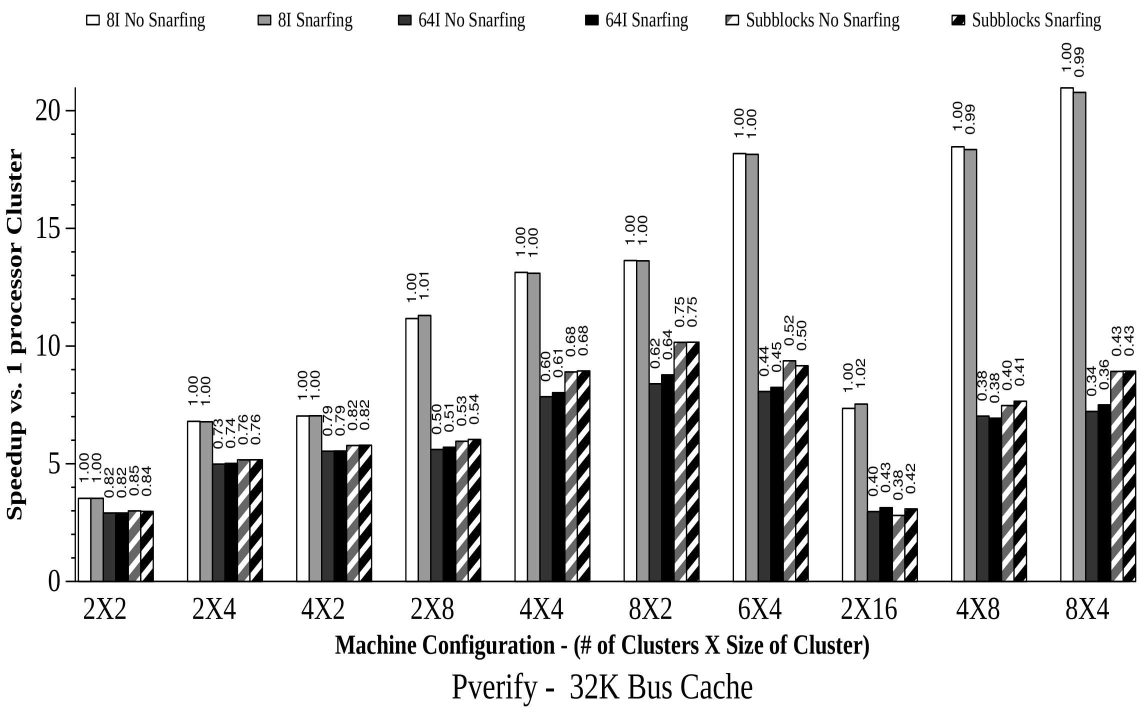


Figure 6.21: Pverify speedups, 32K bus caches

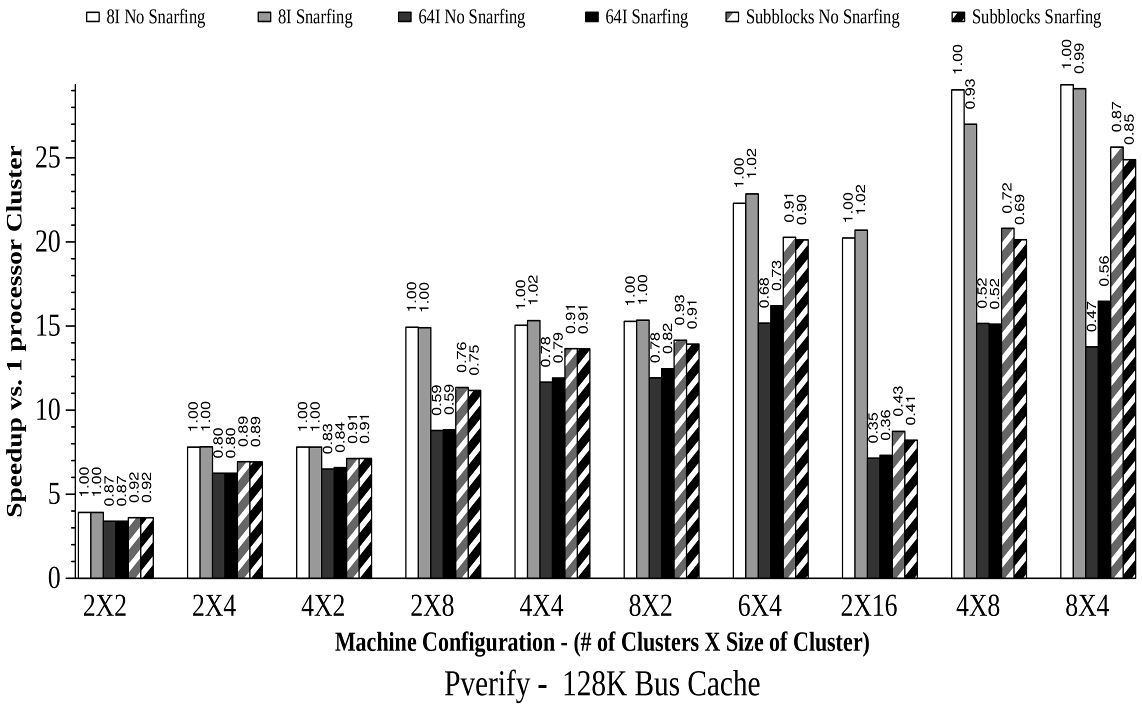


Figure 6.22: Pverify speedups, 128K bus caches

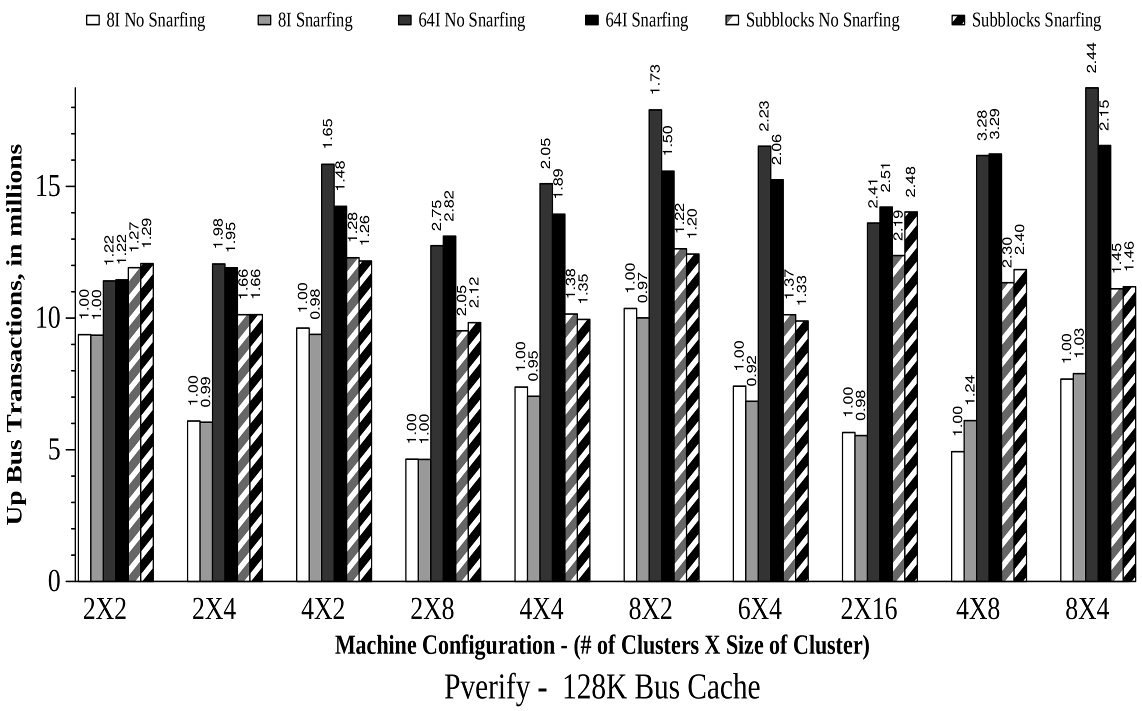


Figure 6.23: Upper Bus Transactions for Pverify

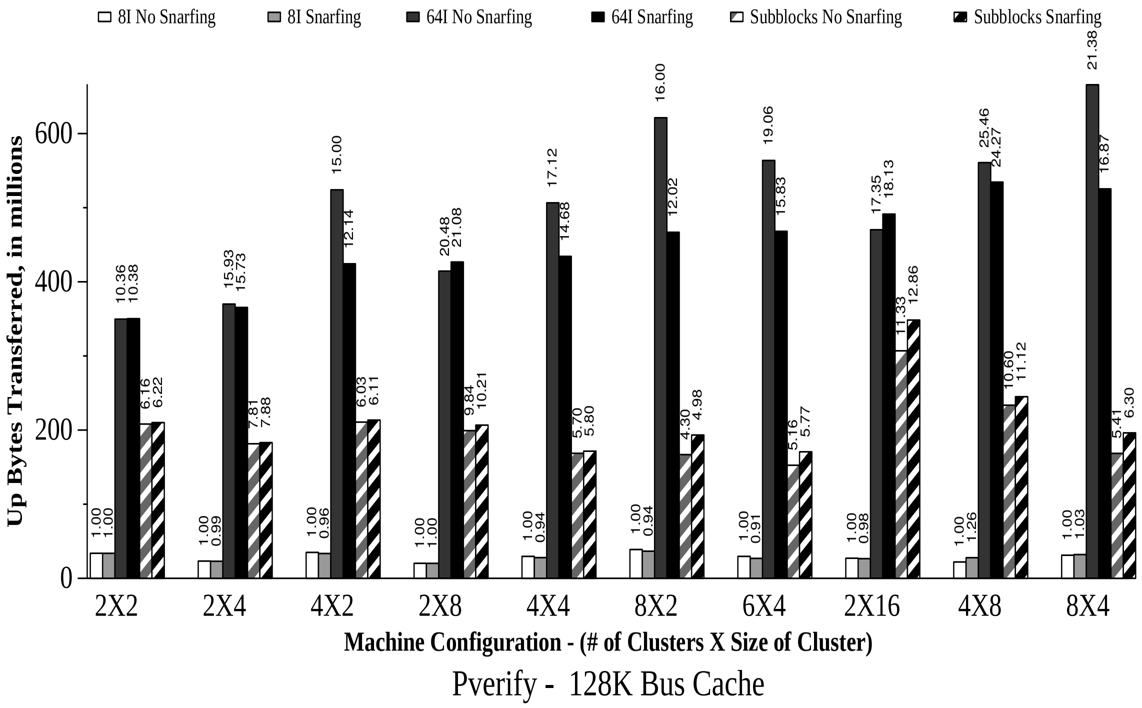


Figure 6.24: Upper Bus Bytes Transferred for Pverify

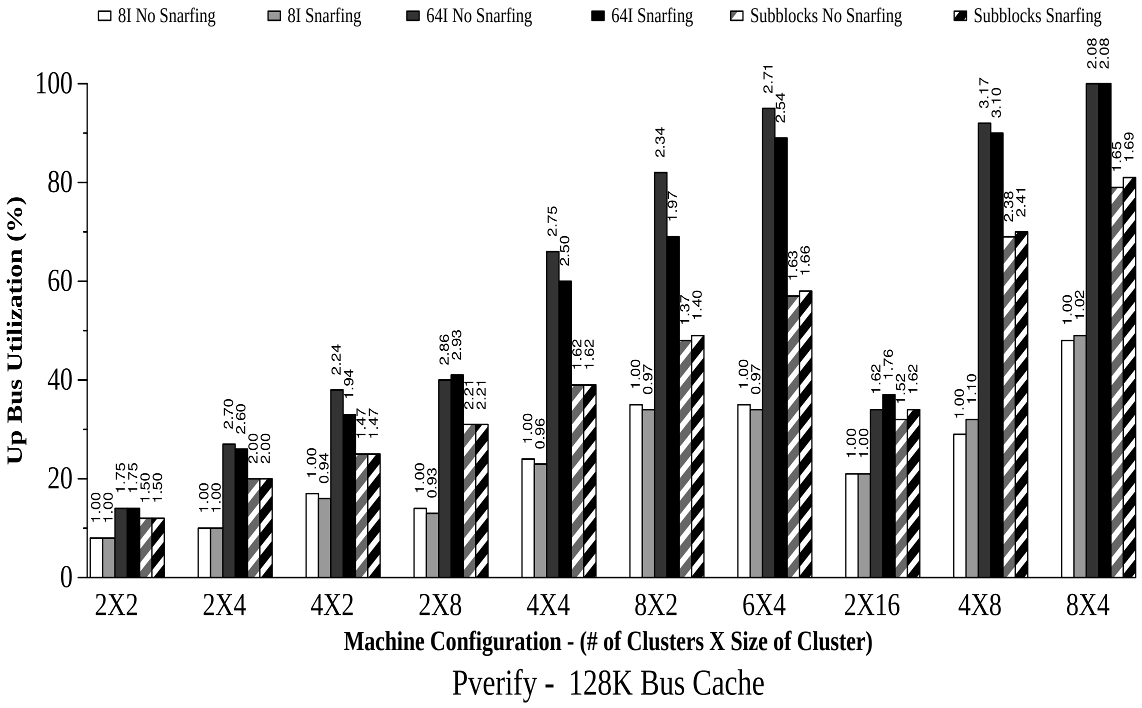


Figure 6.25: Upper Bus Utilization for Pverify

Subblocks and snarfing

The performance of the subblock protocol on Pverify varied quite a bit from configuration to configuration. In general, subblocking worked best when it worked like the 8I protocol — that is to say, when it transferred *few* subblocks in response to a read request. Since the subblock protocol was designed to transfer all of a block's shared subblocks in response to a read request, it performed well when blocks on average contained few shared subblocks. In general, a cache block contains few read-shared subblocks only after most or all of its subblocks have been written, either by the processor (or cluster) that owns the cache, making the subblock(s) exclusive, or by other processors, which makes the subblock(s) invalid. Replacing, then rereading a block tends to increase the proportion of valid subblocks, since the responder sends all other shared subblocks in the block. Thus, the subblock protocol performed similar to the 64I protocol when small bus caches were used, but performed much closer to 8I when 128K caches were used. Subblocking also worked better with fewer processors per cluster. It performed within about 10% of 8I's performance when 128K bus caches were used, except for the 2X8, 2X16, and 4X8 configurations.

Subblocking used about the same number of bus transactions as the 64I protocol for 32K caches, while it used many fewer transactions with 128K caches. For example, for the 8X4 configuration, subblocking used 95% of the transactions 64I did when using 32K caches, while it used 61% of the transactions when using 128K caches. And while subblocking didn't come close to 8I's relatively low numbers for the number of bytes transferred, it did much better than 64I when using the large bus cache. Subblocking transferred about 5–7 times the data that 8I did (except for the 2X8, 2X16, and 4X8 configurations) vs. 10–21 times for the 64I case. Again, when using the small bus cache, subblocking did more poorly, though in some cases (e.g. 4X2, 8X2, 8X4) it cut the bytes transferred by 30% or more. These reductions in network traffic led the subblock protocol to perform better than the 64I protocol in most cases.

Snarfing was not an effective technique for improving the performance of Pverify. For nearly all configurations and protocols, and for both tested bus cache sizes, snarfing had little to no effect, either positive or negative, on application performance. Snarfing did have some benefit with large block sizes and 128K bus caches. It reduced the number of bus transactions by 5–20% when 4 or more clusters were used. It also reduced the number of bytes transferred by 5–30%. These reductions increased 64I's performance for the 8X2 (5%), 6X4 (7%), and 8X4 (16%) configurations.

In certain cases, snarfing actually decreased performance. One reason is that, as explained in the section on MP3D, snarfing on the upper bus can increase the number of invalidations on the cluster busses. In addition, snarfing increased the number of upper bus read transactions for some configurations (2X16, 4X8) and protocols. We are investigating the circumstances in which snarfing can cause an increase in upper bus read transactions.

In summary, Pverify worked best when 8 byte blocks were used. Using the large block size hurt performance not only because of false sharing, but also because it transferred a great deal of data that processors didn't use. Because the subblock protocol attempts to transfer many subblocks at a time, its performance suffered as compared to the 8I protocol. Snarfing didn't help the performance of any of the protocols in a significant way.

Topopt

Hierarchical architecture

Topopt is similar in some ways to Pverify. It has a great deal of true sharing, and also exhibits false sharing. Like Pverify, the best speedups were obtained when using the 8 byte block size (Figure 6.26). Using 32K bus caches, speedups were about 7.6 using 8 processors, 12.3 using 16 processors, and 23–24 using 32 processors. Topopt is different from Pverify in other ways. Topopt has a smaller working set size, so increasing the bus cache size had a smaller effect on Topopt than Pverify. The speedups for the 8I, 128K case were about 7.8 for 8 processors, 12.7 for 16 processors, and 21–25 for 32 processors. An anomaly occurred in the results for the 4X8 configuration — increasing the bus cache size led to about a 10% decrease in Topopt's performance. This happened despite the fact that increasing the cache size led to a decrease in memory latency times, though it did increase the number of bus transactions and bytes transferred on all cluster busses and the top bus. One explanation for this phenomenon is that increasing the bus cache size increased contention for the Test and Test and Set lock guarding the barrier code. It is well known that TTS locks perform badly when many processors contend for the lock [And90]. The problem is especially bad when the critical section is short, as it is in the barrier implementation we used. Therefore, it is quite likely that bad synchronization behavior caused the slowdown in the 4X8 configuration.

The effects of false sharing can be seen in Figure 6.27, the graph of the number of upper bus transactions. Using the 64 byte block size increased the number of transactions by a factor of 1.5 to 2.1 over using the 8 byte block size. Both read and invalidate transactions increased in number when the block size was increased. The differences between 8I's and 64I's number of transactions increased as the number of clusters increased. This is to be expected, since more of the false sharing traffic must cross the top interconnect when more clusters are used. When the size of the bus cache was reduced to 32K, the number of transactions increased across all tested configurations. The increases ranged from about 9% to 37% for 8I, and from 11–53% for 64I.

Since every read transaction using 64 byte blocks transfers 8 times as much data as when 8 bytes blocks are used, it is no surprise that 64I transferred between 11 and 15 times the data on the upper bus as the 8I protocol (Figure 6.28). Again, the disparities between the two increased as the number of clusters was increased (keeping the number of processors constant). Because the subblock protocol (which does not suffer from false sharing) also transferred many times the data that 8I did, part of the large increase in data transferred by 64I was due to the fact that many of the additional words transferred with the requested word(s) were not used by the processor. Like the case with Pverify, 64I's additional bus traffic caused it to perform not as well as 8I. However, since Topopt did not saturate the upper and cluster busses as badly as Pverify did, the drop in performance caused by using the large block size was in general less with Topopt.

Subblocks and Snarfing

Like the case with Pverify, the subblock protocol performed better relative to 8I when 128K bus caches were used. Subblocking performed about 90–95% as well as the 8I protocol, except for the 2X16 case where it performed 19% worse, and the 8X4 case where it performed 6% *better* than the 8I protocol. Relative speedups were not nearly as good when the small bus cache size was used. There, the performance drop varied from 13% to 26%. Again, we attribute the subblock protocol's relatively poor performance when using 32K caches to replacement activity which increased the average number of valid subblocks per block.

Another factor is that using a large block size for cache allocation purposes tends to increase the number of cache replacement misses if a program lacks spatial locality and does not use most of the block. Since the subblock protocol, like the 64I protocol allocates 64 bytes at a time in the cache, it had more read requests (and therefore total bus requests) than the 8I protocol. The bus request increases of subblocking over 8I were much worse when 32K bus caches were used. They ranged from 30–60%, except for the 2X16 case (7%). With 128K caches, the increases when using fewer than 32 processors were 10% or less; for the 2X16 and 4X8 the increase was about 21%, while the 8X4 case saw a 5% *decrease*. In all cases, the subblock protocol used significantly fewer upper bus transactions than 64I. If we examine the cumulative number of lower bus transactions, we find that the patterns of increase are comparable to the upper bus traffic patterns.

A similar pattern occurred with the counts of the number of bytes transferred. With 32K caches, the subblock protocol transferred between 5 and 9 times the data

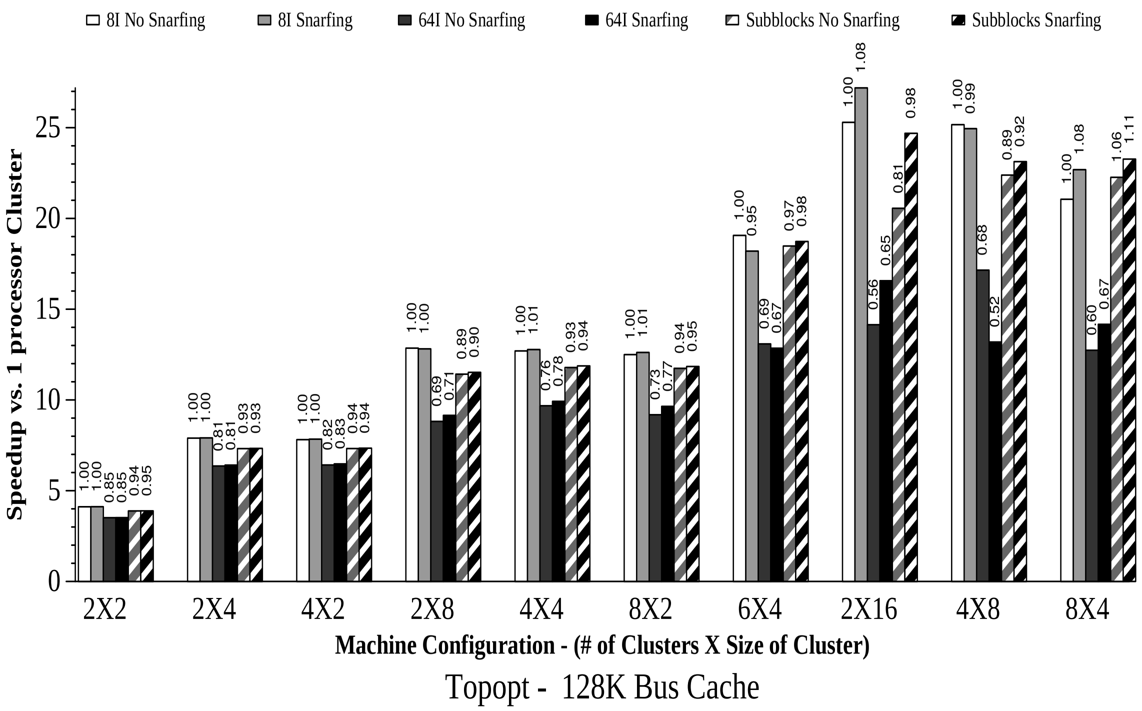


Figure 6.26: Topopt speedups, 128K bus caches

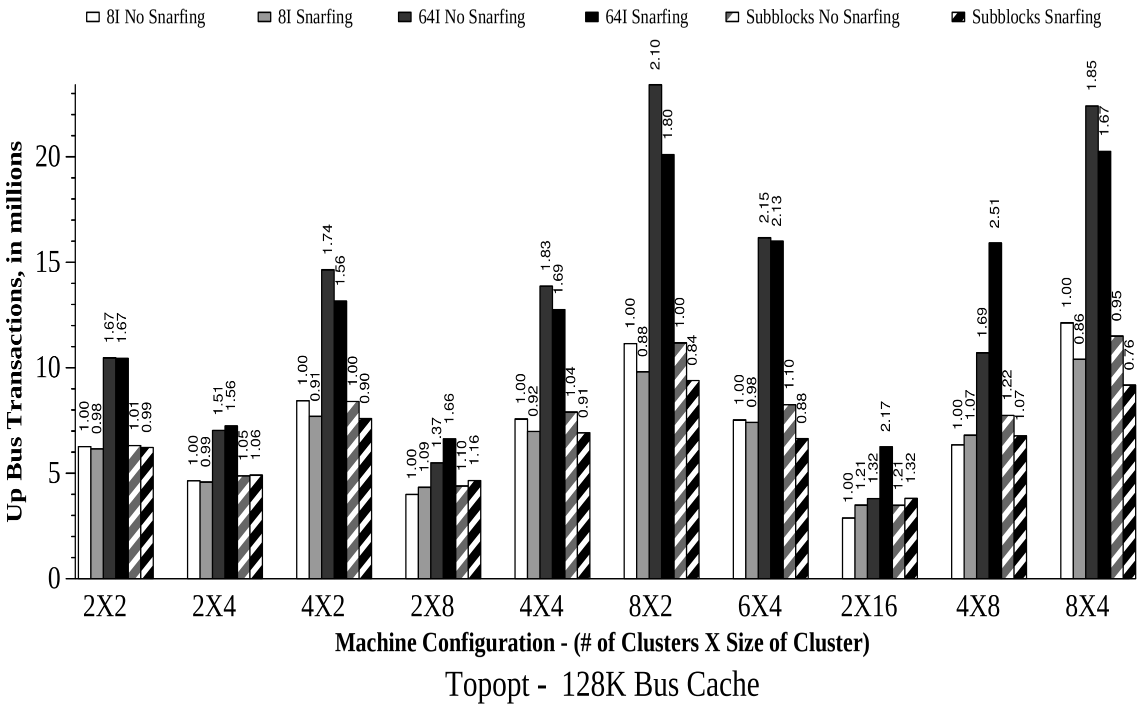


Figure 6.27: Upper Bus Transactions for Topopt

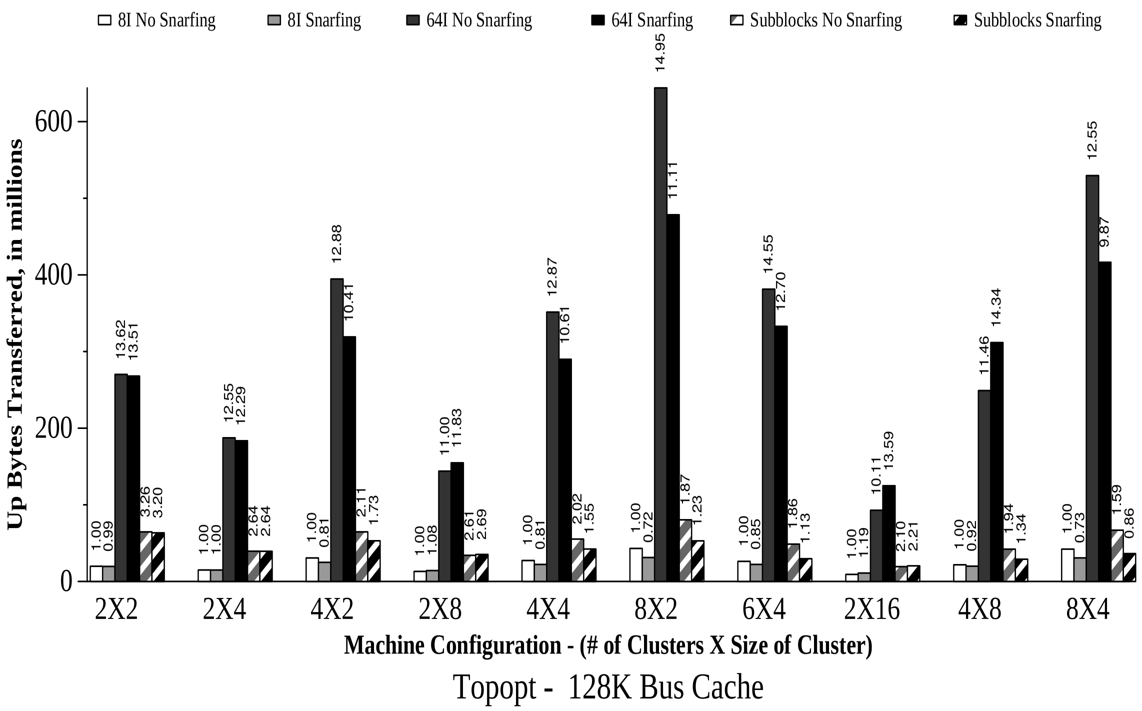


Figure 6.28: Upper Bus Bytes Transferred for Topopt

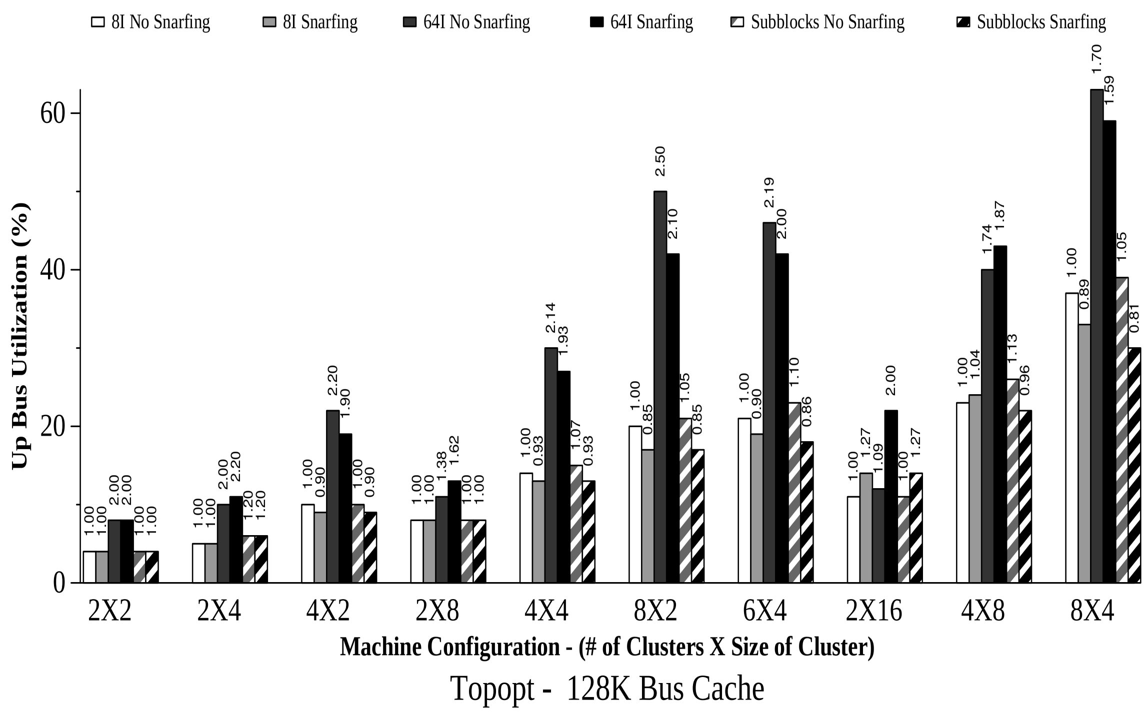


Figure 6.29: Upper Bus Utilization for Topopt

that the 8I protocol did, while with 128K caches the increases were from 1.6 to 3.2 times. Again, the lower bus traffic bytes transferred statistics show a similar pattern.

Snarfing had mostly positive effects on the performance of all three protocols. In a majority of cases for both cache sizes, the improvements were small, i.e. 5% or less. In a few cases, the improvements were larger. Snarfing improved performance of the 8I protocol when 24 or more processors were used with 32K bus caches by 6–11%. When 128K caches were used, snarfing had a strong benefit on the 2X16 organization, improving the performance of the 8I (up 8%), 64I (up 16%) and the subblock protocol (up 21%). Snarfing also caused some small (5% or less) performance drops, primarily in the 6X4 and 4X8, 128K organizations. The most interesting result occurred with the 64I, 4X8 configuration. Snarfing increased performance of the 32K machine by 20%, while it decreased performance of the 128K machine by 24%. It is instructive to examine why this occurred, since snarfing caused smaller performance drops in other cases. When 32K caches were used, activating snarfing decreased both the number of bus transactions and the number of bytes transferred, while the opposite occurred with 128K caches. The number of invalidation transactions on both the upper and cluster busses increased for both cache sizes, though the increase was much larger when 128K caches were used. The increase on the lower busses was partially due to the increased number of invalidations caused by bus cache snarfing. An even bigger factor was the number of retried bus requests, which jumped a great deal when snarfing was used with the large bus caches. Snarfing caused an increase in the number of retried atomic swap requests from 15 million to 51 million. This is a direct result of increased contention for the barrier lock; swap requests are only used when doing a lock or unlock operation. The increased bus traffic caused by these two factors is the reason why snarfing decreased performance in a small number of tested configurations. All in all, though, snarfing helped performance more than it hurt it.

In summary, Topopt performed best using an 8 byte block size. Using a 64 byte block size hurt performance, though not as bad as it did in the case of Pverify. The subblock protocol was close to matching 8I performance when using 128K caches, but did not do as well with the smaller bus cache because block replacements increased the average number of valid subblocks per block. Snarfing had small, beneficial effects on performance in most cases; however, the poor behavior of TTS locks under contention sometimes caused snarfing to hurt performance.

6.3 Results summary and Conclusions

The cluster concept performed well on our varied mix of applications. Speedups were generally very good, except for Cholesky and MP3D. Cholesky did not speedup well because the data set we chose to simulate was small (to keep simulation times manageable) and lacked sufficient parallelism. MP3D, on the other hand, had so much sharing that it saturated the upper bus, long before it saturated the lower busses except when only two clusters were used. Like the case for single bus systems, using the "wrong" block size for an application substantially reduced performance because of increased bus traffic. The subblock protocol, which was designed to take advantage of an application's spatial locality while avoiding false sharing problems, functioned well on most of the applications. Though in only a few cases did it exceed the performance of the best choice of block size, its execution times were nearly always closer to the faster of either the 8I or 64I protocol. However, its behavior was not quite as good as it was for the single bus case. Some explanations for this include:

- The subblock protocol lacked a *Dirty Shared* subblock state at the bus cache level. This caused some level of increase in the number of cluster bus transactions.

- The subblock protocol allocated cache space on a block (64 byte) by block basis. This works well if a good portion of the block is used, or if the cache is large enough that capacity/conflict misses are not a problem. This was often not the case for Topopt and Pverify, especially at the processor level where small caches (8K) were used.

- The smaller caches used in the hierarchical system increased the replacement of cache blocks. This had the effect of increasing the average number of valid subblocks per block. For most applications, this would speed execution, but it did just the opposite for Topopt and Pverify since the additional valid subblocks were often not used. In addition, the subblock protocol's policy of transmitting more than just the requested subblock in response to a read request wasted bus bandwidth because in many cases the additional subblocks were not used by the requester.

Unfortunately, snarfing did not have the large benefit it did on single bus systems, as we saw in Chapter 3. There are several reasons for this:

- The processor caches in our hierarchical systems were much smaller (8K) than in our single bus evaluations. With small caches, it is more likely that an invalidated cache block will be replaced by another block before there is an opportunity to read the contents via a snarf operation.

- The limited number of processors per cluster limited the amount of snarfing that could be done at the cluster level.

- As stated before, snarfing on the upper bus *increased* the number of cluster bus invalidations. The solution to this problem would be to have a bus cache state which indicates that a block is clean shared, but is not present below the bus cache. Regrettably, adding such a state could require an additional block (or subblock) state bit. In the absence of this state, snarfing on the upper level is probably not a good idea, especially when an application has a great deal of migratory data.

One of the most important goals in designing a cluster architecture is that of *balancing* the traffic that will be carried on each level. If a system designed for a given number of processors has many processors per cluster, and few clusters, then traffic on the upper bus will be relatively small, since a given processor can access a good proportion of the other processors in the system merely by accessing the cluster bus. In addition, snarfing will work well on the lower level, since there are many other processors to snarf from. The biggest drawback is that the cluster busses can be saturated by the traffic from the many processors per cluster, while the upper bus is relatively unloaded. This often happened with the 2X16 configuration. On the other hand, if few processors per cluster are used, then the upper level bus becomes saturated long before the cluster level busses. In an ideal situation, cluster bus utilization would increase at the same rate as upper bus utilization. In this situation, the utilization of bus resources would be perfectly balanced across all busses. In practice, this is nearly impossible to achieve, since a configuration that works ideally well on one application might perform very poorly on most others. However, cluster systems can be constructed that achieve good balance on most applications. To construct such a system, an architect must consider many factors, including processor speed, typical memory reference patterns, the latency and bandwidth of the various levels of interconnect, and the size(s) of various caches in the system. If upper level caches maintain the inclusion property with regard to the lower level caches, then the upper level cache should be large enough so that replacements in the upper level cache do not force the replacement of useful data from the lower level caches.

To summarize our results, we present the outcomes from three balanced configurations for each program (Figures 6.30 and 6.31). Each performed consistently well on the tested benchmarks. Barnes had so few shared references that both bus levels were lightly loaded, even in the unbalanced configurations. Much of Gauss's traffic was to and from memory, because of conflict misses. Therefore, upper bus utilization was high, even for the balanced architectures. The balanced architectures did not experience cluster bus saturation, unlike, for example, the 2X16 configuration. Cholesky's lack of speedup even on the balanced systems was due to the absence of application parallelism due to the relatively small input data set. Bus utilization when using 64 byte blocks and 128K caches never exceeded 40% on either upper or lower busses. MP3D's poor speedup, on the other hand, was due to large amounts of true sharing which caused upper bus saturation for all but the most imbalanced of configurations long before linear speedup was reached. When using the 8 byte block size and 128K bus cache with Pverify, the 8X4 and 4X4 configurations had very well balanced bus loads. On Topopt, the 4X8 and 4X4 configurations overall were the most balanced, especially when the large (i.e. wrong) block size was used.

Despite the advantages of cluster architectures, they do have one major drawback compared to flat (non-cluster) architectures: namely, the increased time required for memory requests to traverse the hierarchy if they cannot be satisfied in the originating cluster. Because of this, a flat (non-cluster) architecture will perform better than an "equivalent" cluster architecture if the application does not heavily load the flat architecture's bus. Flat architectures will also do better if an application has poor communication locality, i.e. there is a great deal of data sharing between all the threads of an application. Still, cluster memory architectures can support more processors than flat architectures because memory traffic is localized, rather than being broadcast to all processors.

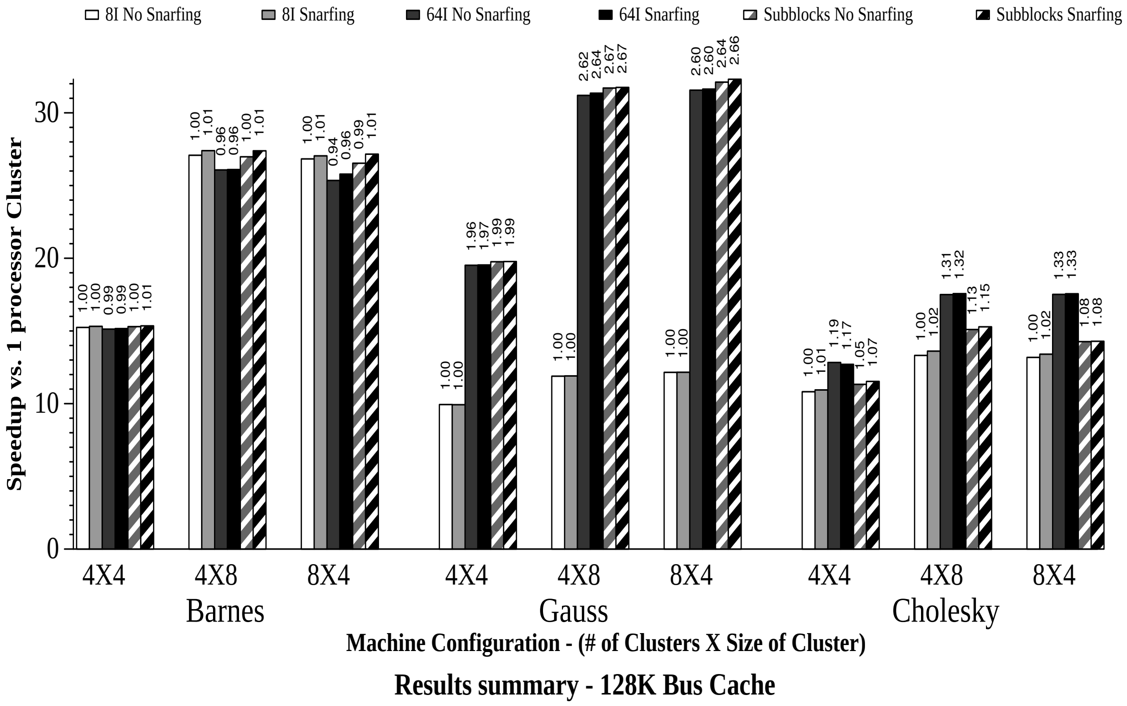


Figure 6.30: Results summary

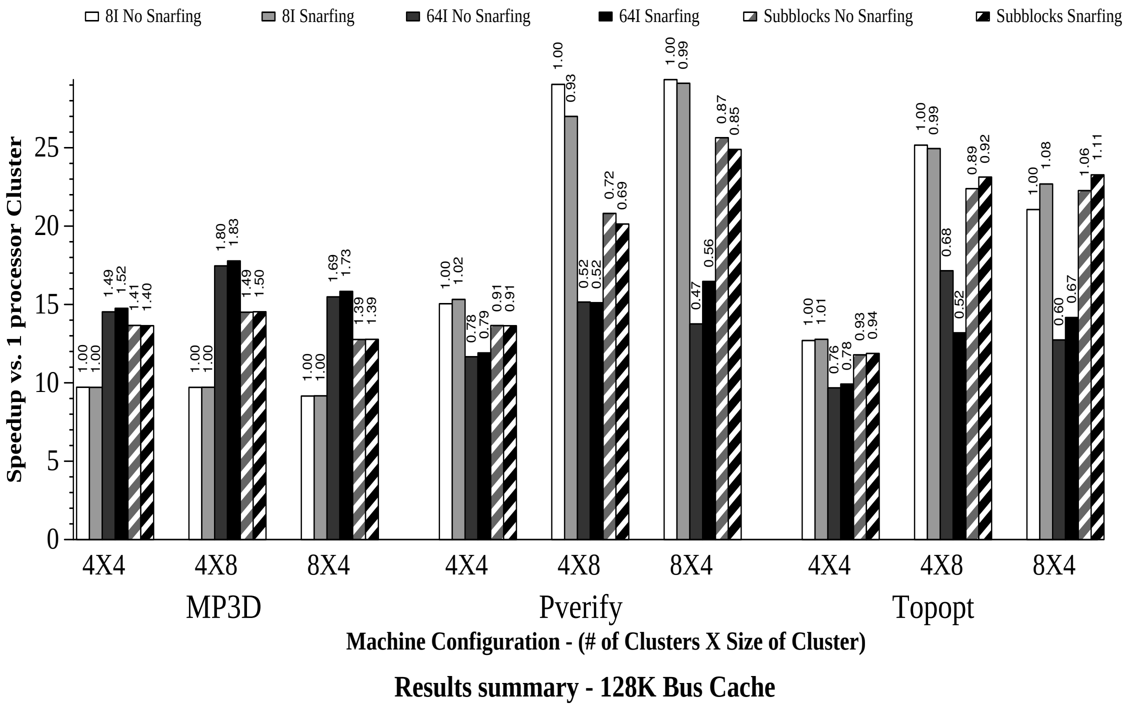


Figure 6.31: Results summary

Chapter 7

DATA PLACEMENT IN HIERARCHICAL ARCHITECTURES

7.1 Introduction

A key feature which determines how long a memory access will require to complete is how far away from the processor the request will have to go in order to be completed. Because of this, the issue of data placement is important when physical memory is distributed among the processors or clusters in a system. To investigate data placement in a hierarchy of busses, we modified the hierarchical architecture detailed previously, moving physical memory into the clusters. We also modified some of the applications so that they allocated data in the cluster in which it is most heavily used.

7.2 Cluster Memory and Memory Allocation

In this chapter, we are interested in answering two questions. First, what effect would moving main memory into clusters have on overall performance? Also, how much benefit would be gained if an application placed its data structures so that misses to memory could be satisfied by the memory located in the cluster. To answer these questions, we modified the architecture shown in Figure 5.1 so that memory was distributed among the clusters. We show a two level system in Figure 7.1. We assumed that each cluster's memory bank could observe and respond to memory requests, both on the upper bus and the appropriate cluster bus. Further, associated with each memory block is a bit indicating if memory has an up-to-date copy of the block. When the system is initialized, all such bits are initially set to *true*. The bit is set to *false* when the memory block is accessed by a cache outside the memory's cluster; it is reset to true when the block is written back. It is necessary to turn the bit off on both read and read exclusive requests from remote caches, since a remote processor may write to a block which has been read from memory without a bus transaction, since the block is loaded in the VALID EXCLUSIVE state. The end-to-end latency time for accessing a 64 byte block in the base (non-memory cluster) system is 36 cycles. Since we assumed that cluster memory banks could respond to requests from outside the cluster in the same manner as the when all memory banks were at the top of the system (i.e. the base system), non-local requests in the cluster system also require 36 cycles. Local requests are satisfied in 23 cycles.

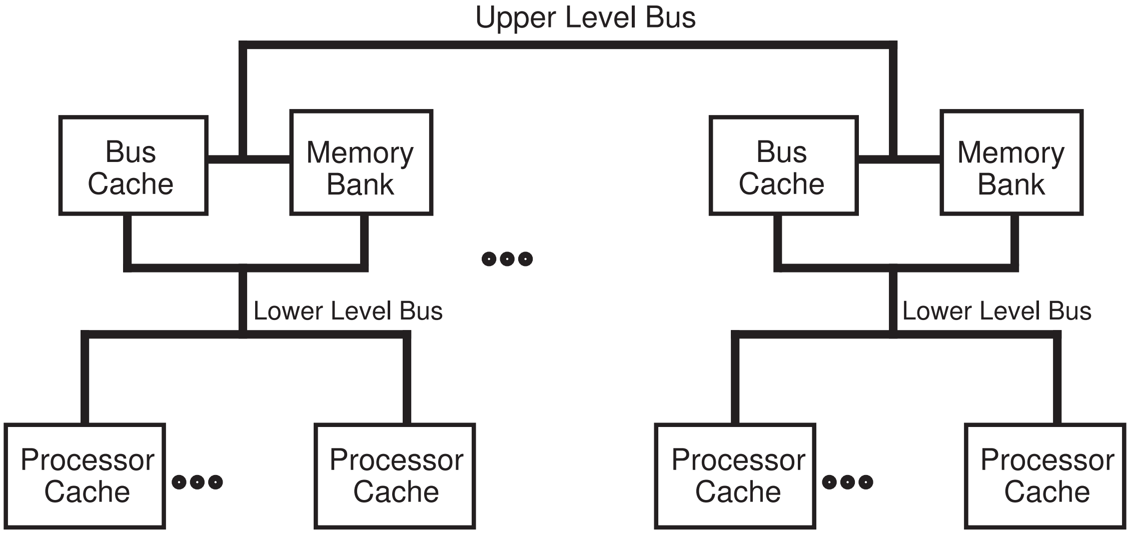


Figure 7.1: Cluster Architecture

The method we used to direct data placement was simple. We modified the memory allocator to provide another function call which would allocate memory specifically in the bank in the processor's cluster. This necessitated a change from fine grain (i.e. block) memory interleaving to coarse grain (page) interleaving. The problem with fine grain interleaving is that successive blocks belonging to the same data structure are spread out amongst successive memory banks, which is undesirable when the entire data structure is primarily accessed by processors in the same cluster. This problem largely goes away when coarse-grain interleaving is used. To this end, we changed the simulator to allow the interleaving size to be any multiple of the block size. In particular, the stride could be set to that of a *page*, which for our experiments was 4 Kbytes. Successive pages are mapped to successive memory banks.

One problem with using coarse-grain interleaving and allocating data in "local" pages is that it can increase conflict misses. This occurs when the bits comprising the page number are part of the cache index function, which happens when the cache size divided by the cache associativity is larger than the page size. Thus, when a processor explicitly allocates data in its cluster's memory, that data will be mapped to only $1/n$ of the cache's space, where n is the number of clusters.

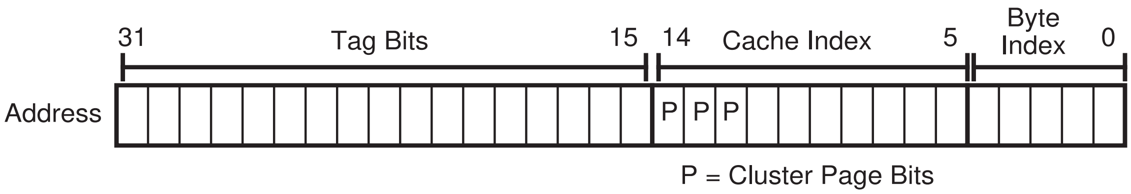


Figure 7.2: Original Cache Index Function

An example follows (cf. Figure 7.2). Suppose the page size is 4K, and there are 8 clusters. Pages 0, 8, 16, 32, etc. will be stored in cluster 0's memory. Now suppose that processor caches are 32K and direct mapped, and a block size of 32 bytes is used. All data structures which are allocated in cluster 0's memory will map to the first 4K of a processor's cache, instead of being spread around the entire 32K cache. The conflict misses caused by mapping large amounts of data to a small portion of the cache can overwhelm any performance increases gained by cluster allocation.

Our solution to this was simple: we used additional address bits in the cache indexing function so that successive pages mapped in the same cluster would map to different parts of the cache. The additional bits were taken from the portion of the address just above the bits normally used to index the cache (see Figure 7.3). Using the above example, the new indexing function would map address 0 and address 32K to different cache blocks, thus eliminating the cache conflict problem caused by cluster allocation.

7.3 Results

In the following sections, we describe the modifications made to each program, then present the results of running the program on the simulator. For comparison purposes, we varied several architectural parameters for our evaluation. One parameter is whether or not memory was distributed in clusters. Another parameter is whether or not the modified cache index was used. Finally, either the original or the modified version of the program could be used.

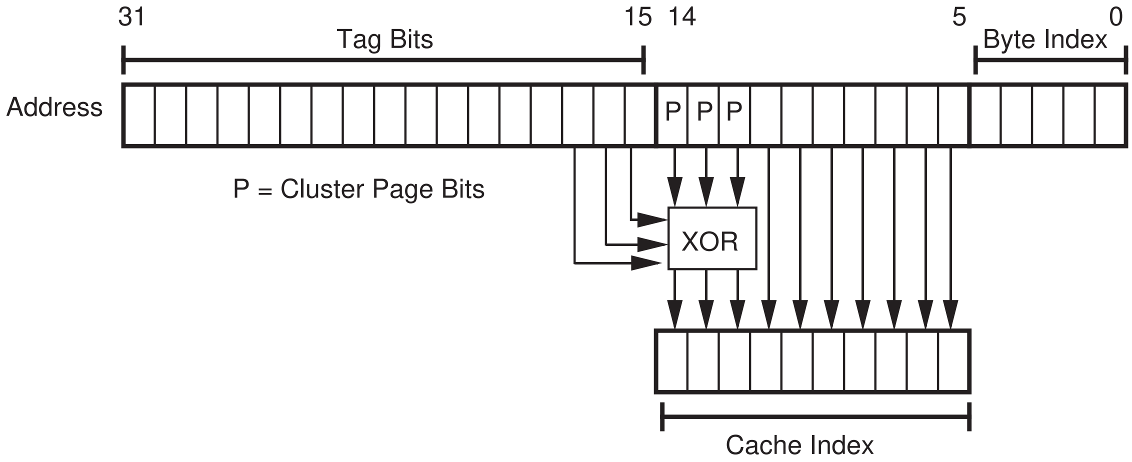


Figure 7.3: Modified Cache Index Function

Table 7.1 lists the configurations that appear in the graphs to follow. We simulated both the regular and modified index (MI) function for all three applications. However, to reduce the number of graphs presented, we will present only one data set for Unmodified Cluster (UC). For Relax and MP3D, we will present UC using the normal index function, since using the modified index function had little effect. On the other hand, we will show data for Gauss on UC with MI, since using MI made a significant difference in UC's performance. For the speedup curves, we normalized to the performance of the Base architecture for the same configuration and block size. Hence a bar of length 1.0 indicates that the configuration performed the same as the Base architecture.

Gauss

Modifications

As described in Section 2.3, the Gauss benchmark solves the system $A \times x = b$. Rows are statically assigned to processors to work on. Since the memory for the rows is allocated dynamically, it is straightforward to allocate each row in the proper cluster. Though the b elements are also statically assigned to processors, we did not allocate them into a specific cluster, since each element would have required a separate call to the allocator, increasing the memory overhead for the b vector by a large amount.

Table 7.1: Evaluated Architectures

Program	Memory Clusters	Modified Index	Modified Program
Base (Gauss, Relax, MP3D)	No	No	No
Unmodified Cluster (Relax, MP3D)	Yes	No	No
Unmodified Cluster/MI (Gauss)	Yes	Yes	No
Modified Cluster (Gauss, Relax, MP3D)	Yes	Yes	Yes

Results

One complication in examining the results for Gauss is the fact that it had many conflict misses, even when many 128K caches are used. Reducing the number of conflict misses by using a different function to index the cache could increase system performance, irrespective of whether or not cluster allocation is used. Because of this, any performance benefits gained by using cluster allocation could be due to the use of the additional address bits when indexing the cache when cluster allocation is used. In order to separate out this effect, when we ran the unmodified program on the cluster architecture, we used the additional address bits to index the cache. We also ran the unmodified program without using the additional address bits for comparison purposes, but these results will not be presented in the graphs that follow.

Using the modified cache index function alone on the unmodified Gauss program had a small, positive effect when small bus cache sizes were used, because most misses were capacity, not conflict misses. When the larger, 128K bus caches were used, many capacity misses turned into conflict misses, some of which were eliminated by a change in the cache index function. Because of this situation, using cluster allocation (and the modified index function) provided a larger benefit when 128K bus caches were used when compared to 32K bus caches (Figure 7.4).

Cluster allocation also had a much bigger impact when the smaller of the two

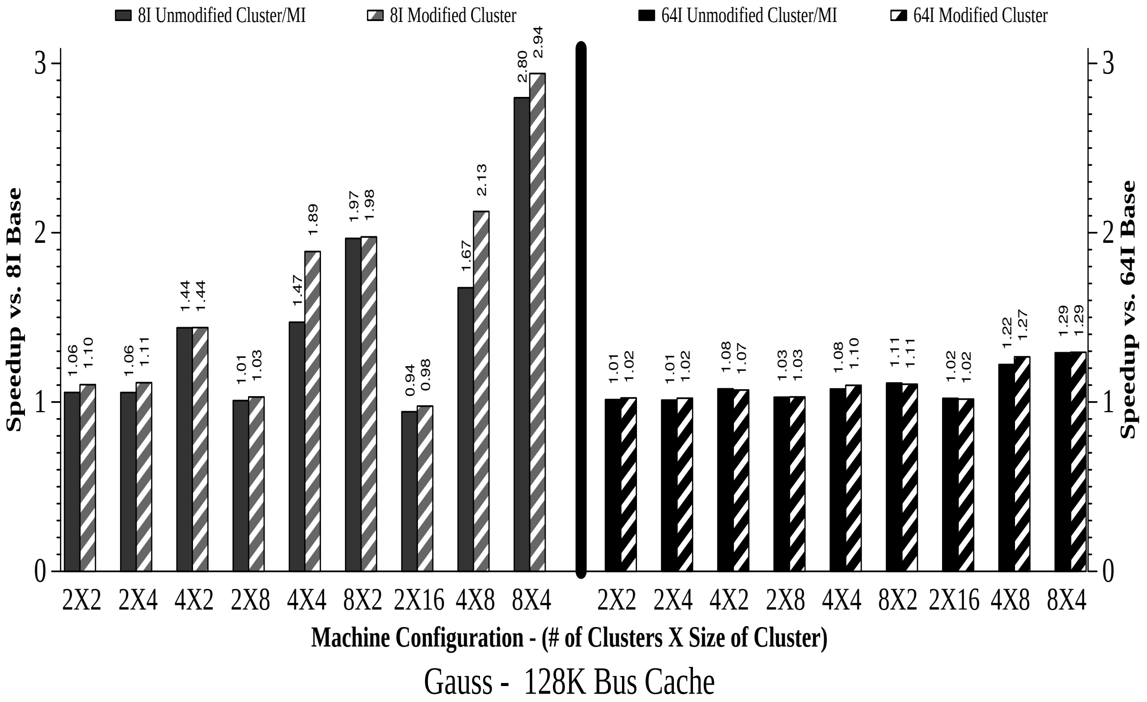


Figure 7.4: Gauss Speedup

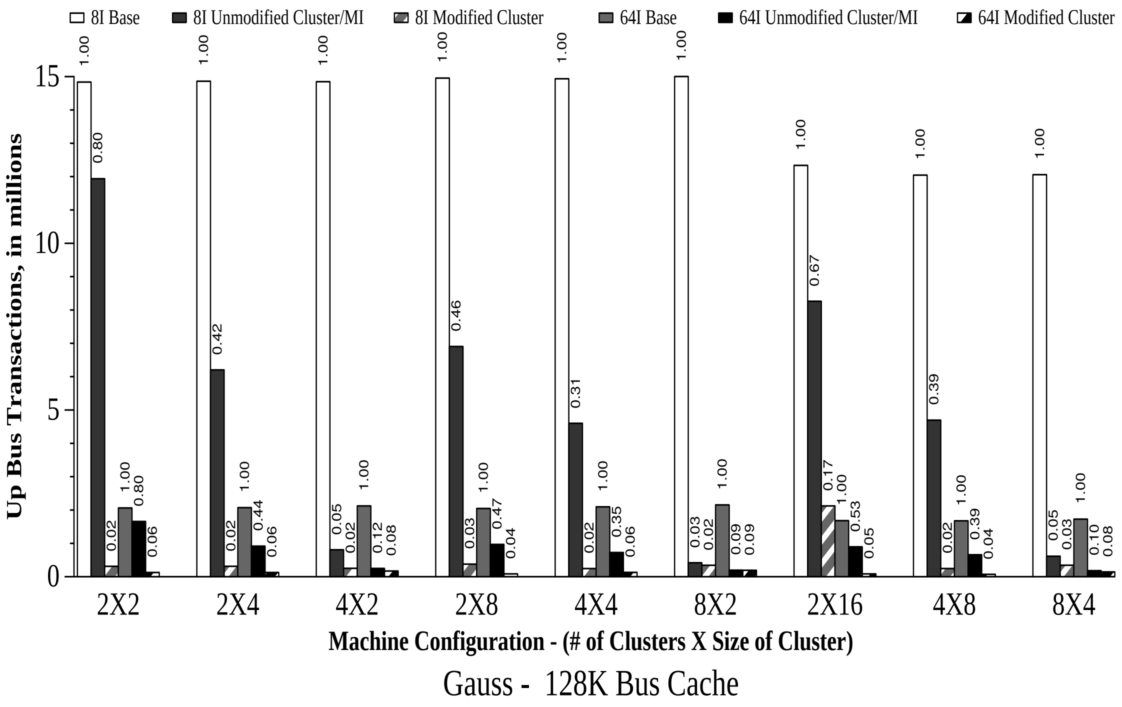


Figure 7.5: Upper Bus Transactions for Gauss

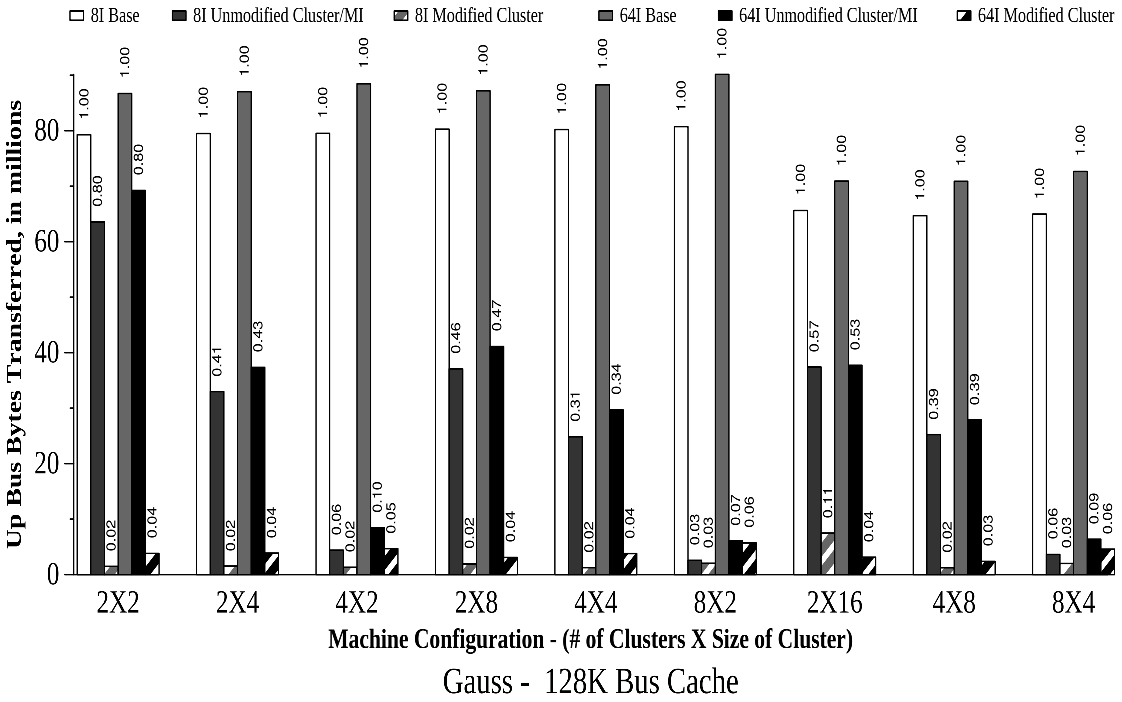


Figure 7.6: Upper Bus Bytes Transferred for Gauss

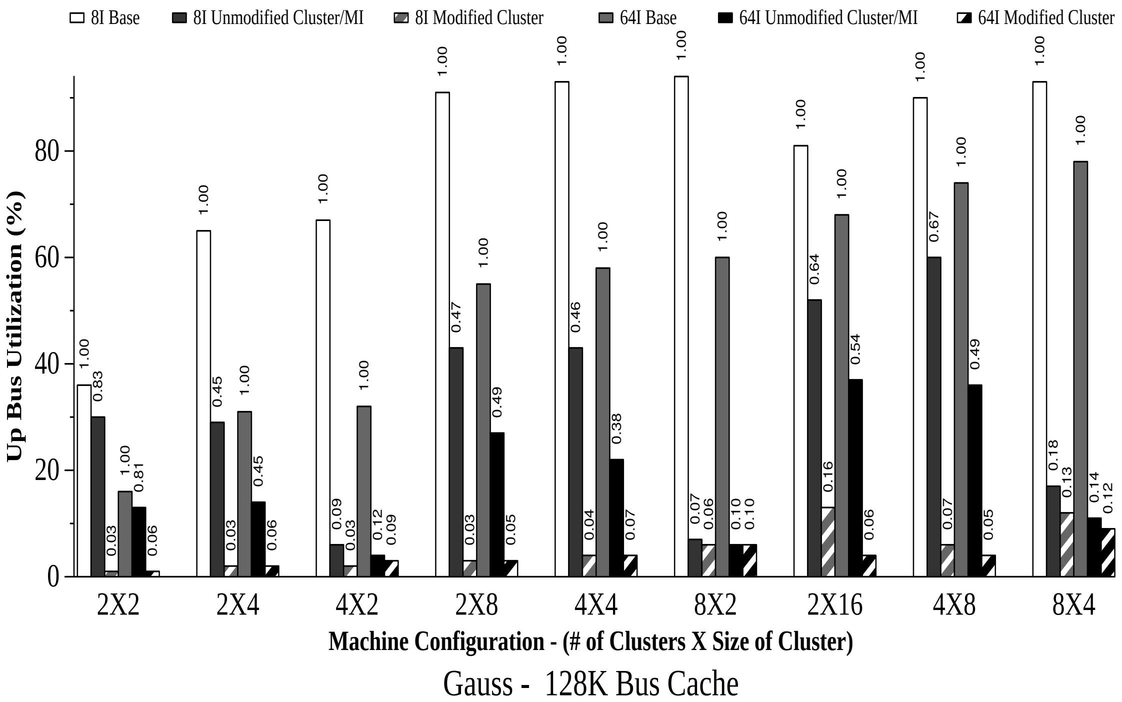


Figure 7.7: Upper Bus Utilization for Gauss

block sizes were used. Because of Gauss's high spatial locality, a miss to a large cache block brings in other needed data as well as the requested data, which reduces the number of bus transactions when compared to using a small block size. Since cluster allocation potentially saves bus cycles on every memory transaction, it has more opportunities to improve performance when small block sizes are used.

Given the above trends, it is not surprising that the biggest gains in overall performance occurred when using small blocks and large bus caches (cf. Figure 7.4). Modified Clusters (MC) increased performance by 98% or more when 8 clusters were used with small blocks and large bus caches; however, Unmodified Clusters (UC) did nearly as well as MC, indicating that most of the improvement was due to reduced conflict misses. MC's performance improved by 44% or more when 4 clusters were used, and UC matched MC's performance only when 2 processors were used in each cluster. The gains when using 4 or 8 clusters were large enough that the small block performance came close to or exceeded Base's performance when using large blocks. Both MC and UC had a small impact on performance when 2 clusters were used, partially due to the reduced number of memory banks serving memory transactions.

The gains when using large caches were lower when large block sizes were used. This is not surprising, since the Base architecture performed so well that there wasn't a great deal of room for improvement. Again, the largest gains (11–29%) occurred when using 8 clusters, though the 4 cluster gains were not far behind (7–27%). Most gains were due to reduced conflict misses. Gains when using 2 clusters were 3% or less.

When 32K bus caches were used, performance gains were roughly equal or lower than when 128K bus caches were used. Modified Clusters on the 4 or 8 cluster organizations and using the small block size increased performance over Base from 11% for the 4X2 configuration to 131% for the 4X8 organization. When using large block sizes, the improvement was 6% or less, except for the 8X4 and 4X8 cases where the increase was 30%. Unmodified Clusters also improved performance over Base, but did not do as well as MC, especially when 4 or 8 clusters or small block sizes were used. Unlike the case with 128K caches, using the modified cache index function with UC sometimes hurt performance, and sometimes helped performance when compared to using the standard function. The modified index function had less of an impact because capacity misses dominated conflict misses.

The performance gains by both MC and UC were due largely to reducing the number of bus transactions (Figure 7.5) and bytes transferred (Figure 7.6) on the upper bus. This was a direct consequence of reducing conflict misses and accessing data in the local cluster's memory. When UC was used with the cluster architecture, greater improvement was seen when using fewer clusters, since with 2 clusters there is a 50% chance a miss to memory will be satisfied inside the cluster, while with 8 clusters the probability drops to 12.5%. MC reduced the number of upper bus transactions by 83–98% across all choices of bus cache size, block size, and machine configuration. The reductions were largest for small block sizes. UC came close to these reductions only when using the large block size with the large bus cache and 4 or 8 clusters (because of a reduction in conflict misses). The reductions in the number of bytes transferred mirrored the reductions in upper bus transactions.

As seen in Figure 7.7, MC's reduction in upper bus traffic had a large effect on upper bus utilization. In almost all cases, utilization dropped by more than 80% over the Base architecture. When the small bus cache was used, UC's reductions did not come close to MC's reductions. When the 128K bus cache was used, only on the 4X2, 8X2, and 8X4 configurations did UC come close to MC's reduction.

Relax

In the Relax benchmark (see Section 2.3), we cluster allocated each processor's intermediate array. On each iteration, each processor computes the next iteration's values into its intermediate array; when all processors are finished computing, the intermediate arrays are written back into the single primary array.

Results

The overall performance of the modified version of Relax was mixed (Figure 7.8). When the small block size was used, MC performed better than Base for both large and small bus cache sizes and all cluster configurations, except the 8X2 128K bus cache organization where it equalled Base's performance. The improvements were largest when the small bus cache was used, since there were more replacement misses than when 128K bus caches were used. The 32K bus cache improvements ranged from 1% for the 2x4 organization, to 42% for the 8x4 organization; generally, the improvements were larger when more processors were used. When the large bus cache was used, improvements ranged from 1% for the 2X4 configuration to 36% for the 4X8 configuration. The large improvements were possible because the speedups when using the Base configuration were not linear, hence there was room for improved performance.

In contrast, using the large block size with the Base architecture led to near-linear speedup, making it difficult for any architectural improvement to have any positive effect.

In fact, while using MC increased performance when using small block sizes, it decreased performance in most cases when large block sizes were used. The size of the performance loss varied from 3% to 8% across all simulated configurations. Only the 8X4, 32K configuration showed any real improvement (6%). The diminished performance of MC was not due to using the modified cache index function. When the modified index function was used with UC, performance changed very little. Rather, using MC dramatically increased the number of read misses in the processor caches. The increases ranged from 13% (2X16, 32K cache) to 251% (8X2, 128K cache). These additional misses caused additional traffic on the cluster busses, which lowered overall performance. The increased misses were accompanied by increased number of spills from the processor caches, consequently it is likely that using the cluster allocator mapped the data in the small processor caches such that higher capacity or conflict misses occurred.

Despite the bad performance of MC when using large block sizes, it substantially reduced both the number of transactions (Figure 7.9) and the number of bytes transferred (Figure 7.10) when either the large or small block size was used. Bus transactions were reduced by 30–70%. Reductions were largest with small numbers of clusters. UC also reduced the number of transactions, because some memory references would (by chance) be satisfied by a processor's cluster memory. As expected, these reductions were on the order of 50% for 2 cluster systems, 25% for 4 cluster systems, and 12% for 8 cluster systems. As was the case for Gauss, the reductions in the number of bytes transferred closely mirrored the reductions in the number of bus transactions.

Unlike the case with Gauss, changing the cache index function had minor effects on performance. When large block sizes were used, the difference in UC's performance was never more than 2 percentage points. The differences were larger and mixed when

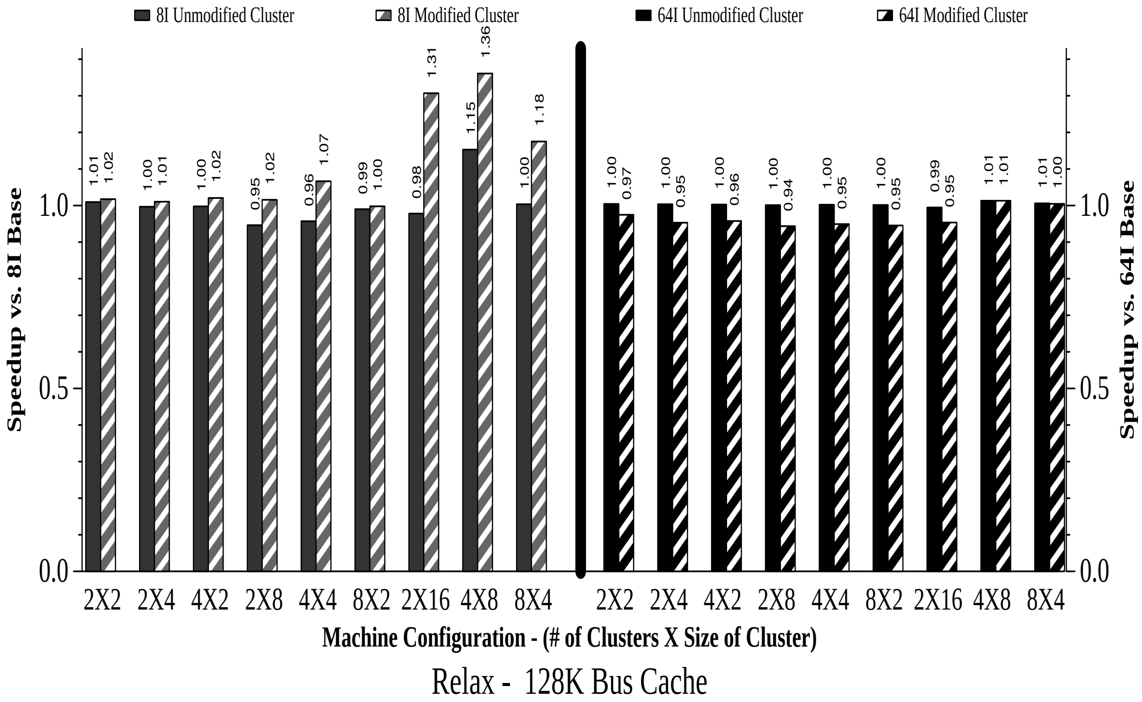


Figure 7.8: Relax Speedup

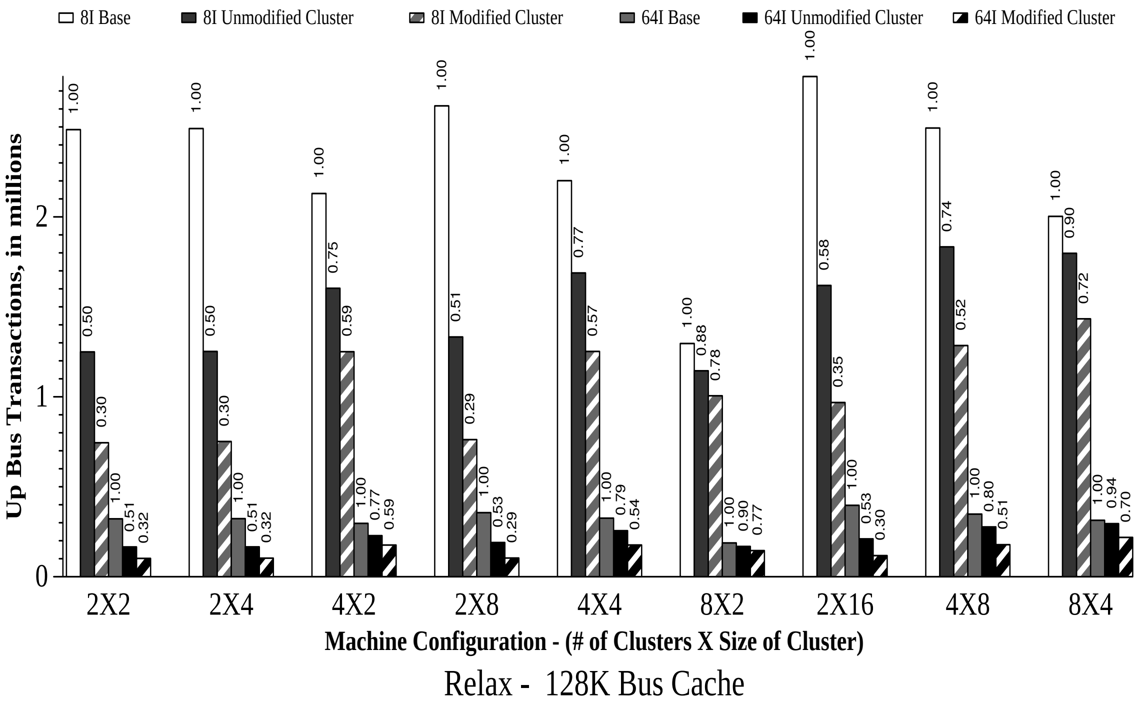


Figure 7.9: Upper Bus Transactions for Relax

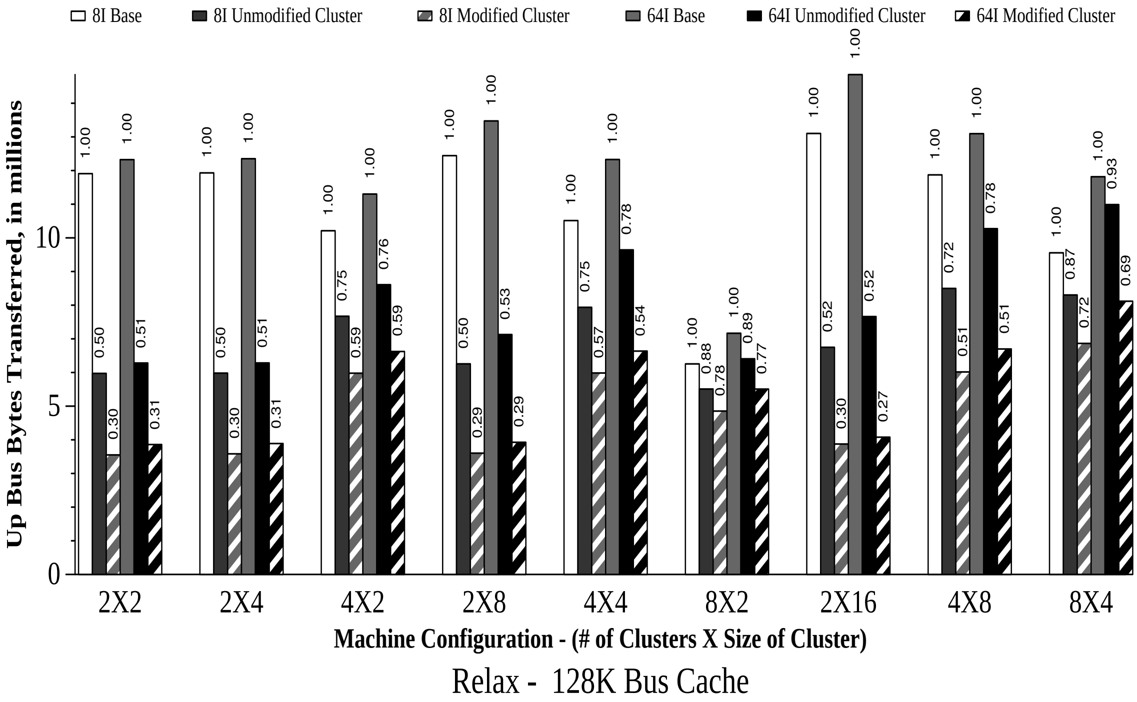


Figure 7.10: Upper Bus Bytes Transferred for Relax

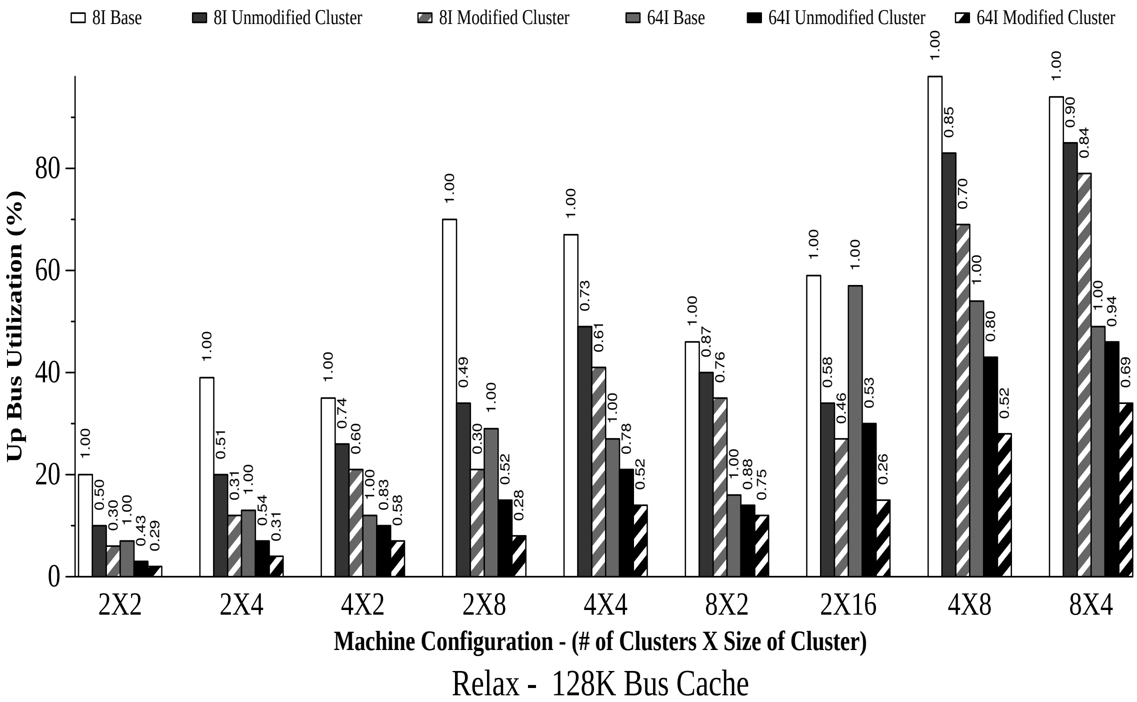


Figure 7.11: Upper Bus Utilization for Relax

the small block size was used. The most extreme case was the performance of the 2x16 configuration, where using the modified index function decreased performance by 12 percentage points when 32K bus caches were used, but *increased* performance by 12 points when 128K bus caches were used.

MP3D

For MP3D, we cluster allocated the particles, since they are statically assigned to processors at the beginning of the program. Unfortunately, the original program allocated all particles as a contiguous single dimensional array, which cannot be spread out among memory clusters so that particles are allocated in the proper cluster. In addition, processors are not assigned a contiguous block of particles to manipulate; instead, small groups of particles are assigned to processors in a round-robin fashion. In order to maintain the same assignments of particles to processors, we converted the single-dimension array into a dynamically allocated, three-dimensional array of particles. This had the effect of increasing the total working set size, because additional storage was required for the arrays of pointers. The change also increased the number of instructions needed to access the particle structure. Thus, in order for the modified program to show a speed increase, it would have to reduce memory access times enough to make up for the additional cycles spent on accessing the particle data structure.

Results

Like the case with Relax, the effect of cluster allocation on the overall performance of MP3D was mixed (Figure 7.12). It always decreased performance for some organizations (2X2, 2X4, 2X8), while for others it either increased performance (8X2, 4X8, 8X4) or had mixed results, depending on the bus cache size and the block size (4X2, 4X4, 2X16). MC showed the greatest improvements over Base when using small block sizes with the 8X2 (39% for 32K, 36% for 128K), 4X8 (73% for both), and 8X4 (70% for 32K, 66% for 128K) configurations. The biggest losses (18% for 32K, 13% for 128K) happened with the 2X16 organization.

MC reduced both the number of bus transactions (Figure 7.13) and the number of bytes transferred (Figure 7.14) on the upper bus for nearly every tested configuration. The reduction in the number of bus transactions ranged from 46–65% for 8 byte

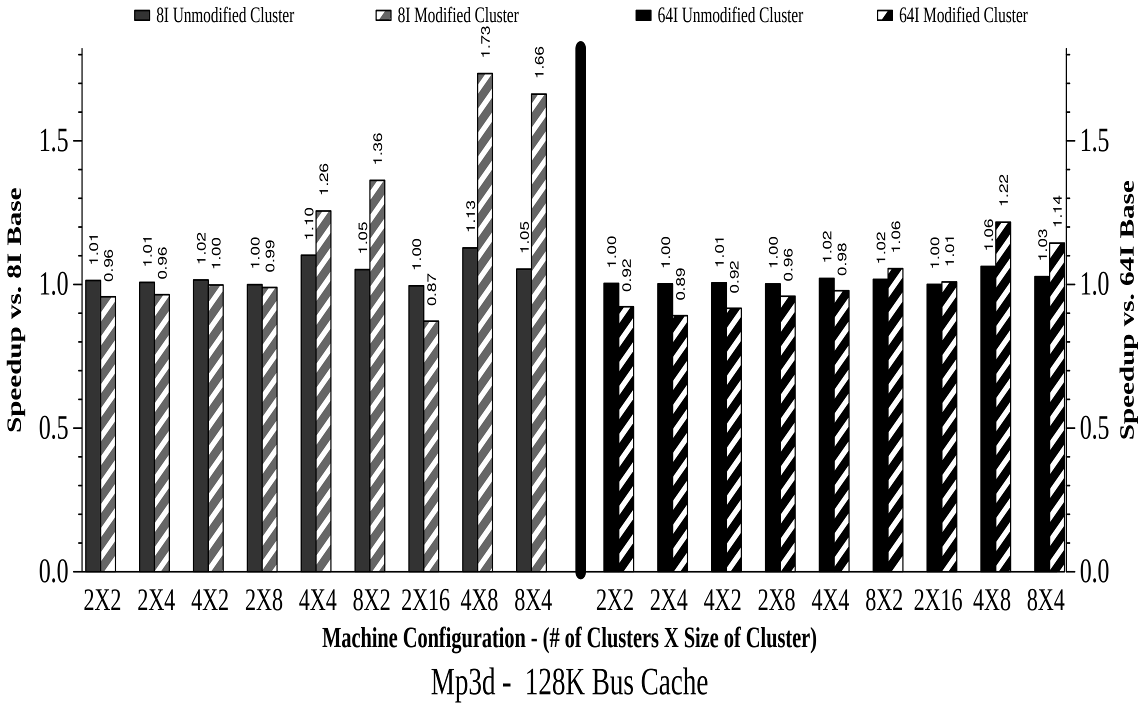


Figure 7.12: MP3D Speedup

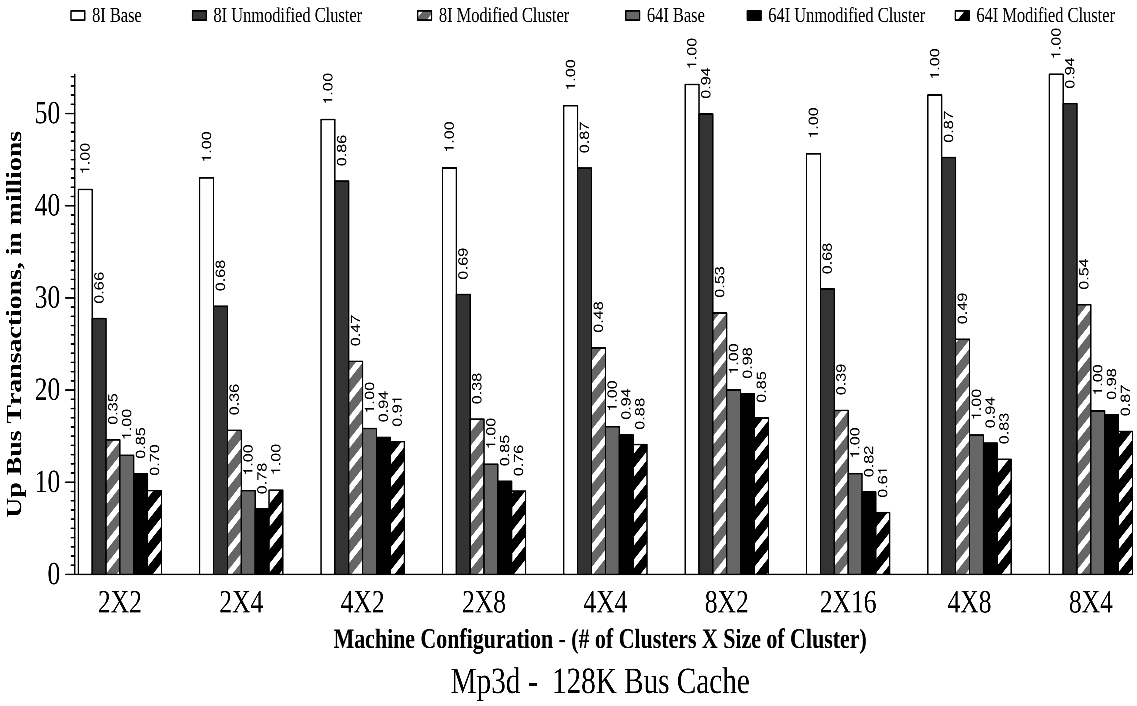


Figure 7.13: Upper Bus Transactions for MP3D

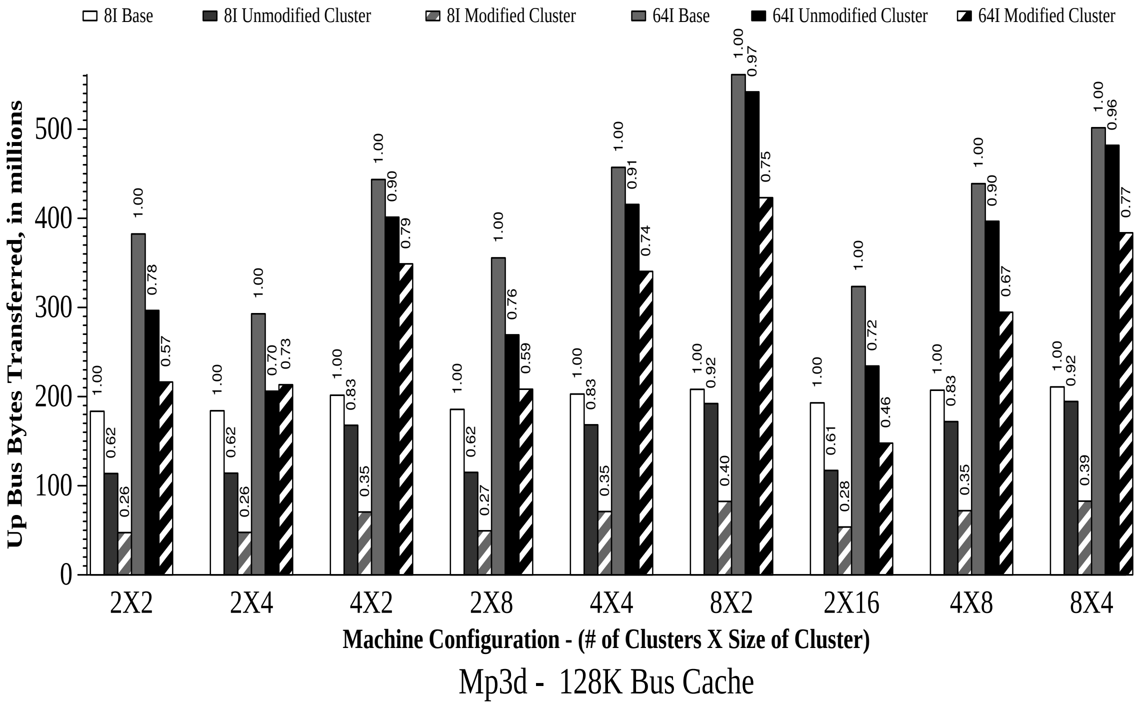


Figure 7.14: Upper Bus Bytes Transferred for MP3D

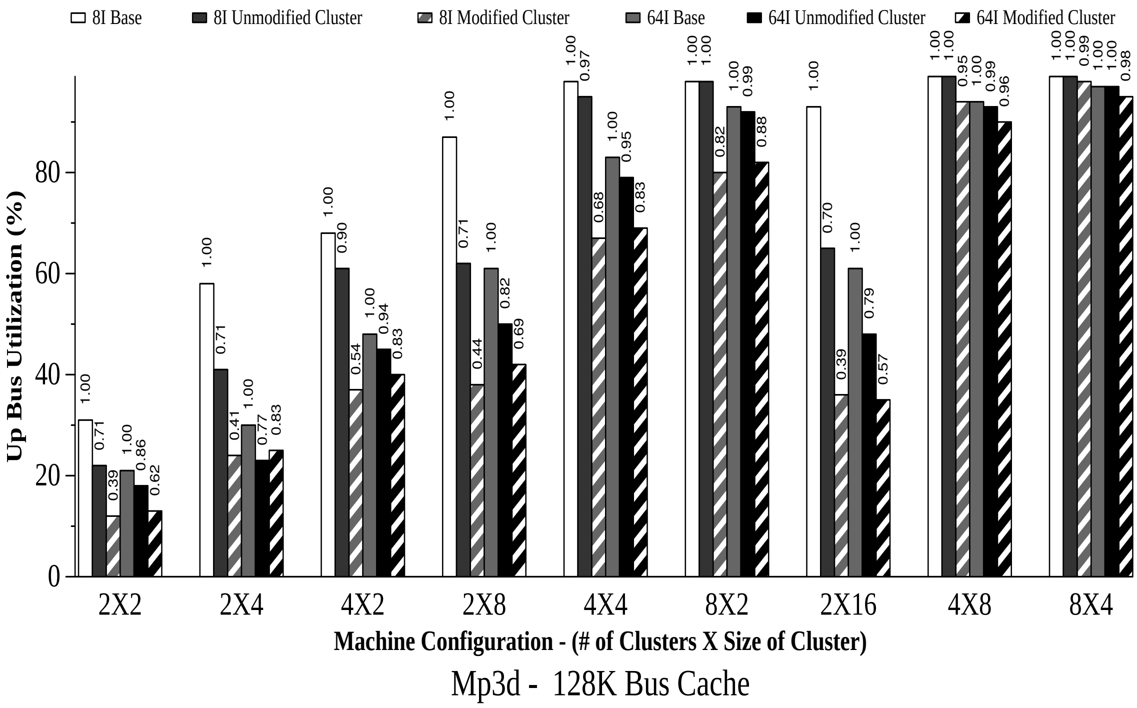


Figure 7.15: Upper Bus Utilization for MP3D

blocks, and 6–43% for 64 byte blocks. Only the 2X4 organization with large blocks and bus cache failed to show any improvement. The reductions in the number of bytes transferred were generally larger. For small blocks, the reductions ranged from 60–74%; for large blocks, 25–45%. Though the savings in upper bus traffic were in most cases substantial, it did not always lead to performance gains. The reasons why include:

Additional instructions Because the particle data structure was changed to accommodate cluster allocation, MC executed more instructions than the unmodified version of the program. The extra time required to execute the additional instructions diminished any gain derived from reducing upper bus traffic.

Higher number of cluster bus reads MC performed more read transactions on the cluster busses than either Base or UC. In all but one case, the increase was only a few percent. The lone exception was the 2X16 case when small blocks were used; there, the increase was 29% using 32K bus caches, and 23% for 128K bus cache. Since lower bus utilization in the 2X16 case was near 100% in the Base architecture, the additional transactions caused memory latency to increase, and decreased MP3D's performance.

Low upper bus utilization The use of cluster allocation increases performance by reducing traffic on the upper bus. The number of cycles saved depends on how busy the upper bus is. In those cases where the Base architecture did not saturate the upper bus, MC's potential gain from reducing upper bus traffic is smaller than those cases where Base did saturate the bus. Despite the large amount of true sharing in MP3D, it did not saturate the bus when 8 or fewer processors were used, even with 8 byte blocks (Figure 7.15). It also did not saturate the upper bus when 64 byte blocks were used in the 2X8 and 2X16 configurations.

Running the unmodified version of MP3D on the cluster architecture (UC) changed performance by 6% or less in most of the tested organizations. UC did improve performance by 10–15% when using the small block size with both the 4X4 and the 4X8 configuration. It also improved performance for the 4X8 configuration by 9% using the large block size and 32K bus cache. The gains were largest when using 4 clusters because there was more memory traffic going to the cluster's memory than the case with 8 clusters, and the memory bank latency did not increase much over Base, unlike the case when 2 clusters (and therefore only 2 memory banks) were used.

We examined the difference in performance caused by changing the cache index function when using the unmodified program with clusters, and found that it made little difference in performance. The maximum performance difference was 3 percentage points; in most cases, the difference was 1 point or less. The modified index function did cause small (about 5%) changes (both higher and lower) in the number of spills from the bus caches in some configurations using the large block size. When using small block sizes, the differences were negligible.

7.3.1 Results summary

Cluster allocation succeeded in reducing upper bus traffic on all three applications. It also increased application performance for all configurations when running the Gauss application, and had mixed results when running Relax and MP3D. While in some cases, cluster allocation provided a large performance benefit for MP3D and Relax, in some cases it led to small performance declines (1–11%) due to increased misses in the processor cache (Relax) or additional instructions needed to access modified data structures (MP3D).

7.4 Previous work

As stated in a previous chapter, one approach to multiprocessor memory design is to treat physical memory as a very large cache, as in a COMA architecture. The advantage of the COMA approach is that initial data placement is not important, since data will be moved to where it is being used; the downside is that since the data address to data location mapping is dynamic, the underlying connection architecture must support efficient broadcast[1].

Singh *et al.* examined the effects of data placement on the DASH, where the granularity of data distribution to home nodes is the page [SJGH93]. A general

[1] The flat COMA as proposed by Stenström *et al.* [SJG92] does not require broadcast to find a block. However, this is at the expense of keeping a sharing list for every block.

technique they tried was the *first touch*, in which data was allocated in the home cluster which first accessed it. This technique has the potential to work well only when data structures are initialized in parallel by the processor which is going to access it the most. In most cases, first touch improved application performance over round-robin allocation, but occasionally it was worse. Restructuring the application's data worked even better than the "blind" first-touch method, though it did make the code more difficult to write, and decreased performance in one instance when a small data set was used.

7.5 Conclusion

Using cluster allocation succeeded in reducing the number of transactions and bytes transferred on the upper bus for all three applications. Reducing upper bus traffic is in general a good way to increase application performance, since saturation on the upper bus often limits application performance. However, lowering upper bus traffic is not always enough to increase application performance. At times other factors, such as changing the code and data structures to do cluster allocation, and changing the cache index function can have an adverse impact on cache hit rates and can increase the total number of instructions executed. These elements can reduce the performance gain of cluster allocation or even lower performance below that of the Base architecture. In addition, cluster allocation is unlikely to benefit applications that already exhibit near linear speedup. Hence using cluster allocation, especially if it requires changes to the application's data structure, doesn't always result in faster programs.

Despite the fact that memory clusters and clusters allocation did not provide improved program performance in all cases, overall both were a benefit. Distributing memory into the clusters is beneficial because by chance some memory accesses will be satisfied by the cluster's memory bank, which is faster than going to the top of the hierarchy to access memory. Using cluster allocation improves the probability that a miss will be satisfied in the local memory. For reasonable choices of cluster size and number of clusters, such as the 8X4, 4X8, and 4X4 configurations, MC performed better than the Base configuration.

Chapter 8

CONCLUSION

8.1 Summary

We have developed two major themes in improving the performance of bus-based multiprocessors. We first explored the concept of improving on conventional snoopy-based protocols so that they perform well even when faced with a wide variety of memory reference patterns.

8.1.1 Single bus architectures

In Chapter 3, we examined two techniques for improving memory system performance on single bus multiprocessors. The first was a subblock cache coherence protocol that addressed the problem of picking a cache block size that works well on a wide variety of applications. As our results showed, no single block size works well for all applications. One possible solution, that of varying the block size dynamically [DL92], was not easily adaptable to snoopy-based coherence protocols. Instead, we adapted a static policy of using a large transfer unit (block), while using a small coherence unit (subblock). We found that the subblock protocol which implemented this policy worked well on our benchmark applications. In some cases, however, the static nature of the subblock protocol's data transfer policy reduced its performance below that of the standard protocol when using a small block size. Section 8.2 discusses a possible remedy.

In addition, we investigated the benefits of read snarfing, also called read broadcasting. This technique takes advantage of the "free" broadcast nature of busses by allowing processors to read data bound for another processor on the speculation that it might be useful at a later time. Because read snarfing is done without interfering with the processor's access to the cache, and never replaces useful data, one might assume that the technique only increases performance. In most cases, this assumption is correct. Snarfing reduces the performance penalty associated with enforcing coherence on blocks that are actively write shared by 3 or more processors. Snarfing is especially effective in increasing performance when an application exhibits false sharing when using a large block size. However, there were isolated cases where snarfing hurt performance. Snarfing has a tendency to increase contention for synchronization variables. Though this did not cause an increase in execution time on single bus systems, it did cause problems when used with hierarchical systems. Snarfing also interacted with the subblock protocol's policy of using a large transfer size and increased bus traffic in one instance. In spite of these minor problems, snarfing is a valuable technique for increasing performance.

Just as no one block size is best for all applications, the write-invalidate protocol doesn't give the best speedups for all applications; sometimes a write-update protocol is more appropriate. In Chapter 4, we examined two adaptive coherence protocols that switch the coherence strategy between write update and write invalidate. Unlike other hybrid protocols, which switch from write update to write invalidate after a fixed number of writes to a block, the adaptive protocols vary the point at which the switch occurs, depending on the prior reference behavior for that particular block. We profiled the benchmarks used in our study and as expected we found a great deal of variability in the write run length distributions between applications. We also found that the write length distribution in many applications was well spread out, indicating that neither a fixed update or invalidate protocol would provide the best performance. We then evaluated the two adaptive protocols against both pure invalidate and update schemes, as well as a static hybrid scheme[1]. We found that both dynamic adaptive schemes performed well when compared to the non-adaptive protocols. In particular, the cheaper, less complicated adaptive hybrid scheme (Random Walk) performed better than the more expensive adaptive scheme. Both did much better than the static adaptive scheme.

Both of these investigations showed that even in the limited class of applications we used as benchmarks (scientific/engineering applications), there is much variability in memory access patterns. Our experiments showed that with the proper hardware and protocol modifications, a shared bus machine can provide good performance on a variety of memory reference behaviors.

---

[1] The static hybrid scheme was evaluated on one configuration only.

8.1.2 Cluster Architectures

We first designed and implemented an invalidation-based cache coherence protocol for a tree hierarchy of busses. Because of the success of the subblock protocol and snarfing on single bus systems, we also extended the subblock protocol to a two level bus hierarchy, as well as implemented read snarfing on both levels of the system. We then ran our benchmarks on both the standard and subblock protocols on a variety of two level cluster configurations, varying both the number of clusters and the number of processors per cluster.

Like the case for single bus systems, some applications had very different performance, depending on the block size used with the standard invalidate protocol. Overall, though, the benchmarks performed well when using the "correct" block size. When configurations were used that balanced bus traffic between the upper and lower levels of the system, saturation of the upper bus sometimes occurred, but in all but one of the benchmarks did not noticeably impair application speedup. Only on MP3D did upper bus saturation (due to a great deal of true sharing between clusters) strongly affect performance. In spite of our good results using identical busses at both the upper and lower levels of the system, it would be advisable to have a "beefier" interconnect above the cluster level. Since it is unlikely that non-cluster busses will be shorter than cluster busses, clocking the non-cluster busses faster than the cluster busses will probably not be an option. Instead, wider busses (to improve bandwidth) or multiple (possibly interleaved) busses [BBW92] would allow more clusters (hence more processors) to be effectively connected in a hierarchical system.

The hierarchical subblock protocol performed relatively well on the six benchmarks. Unlike the conventional protocol used with a fixed block size, its performance across the tested benchmarks was much more consistent, just as the single bus subblock protocol was more consistent than the Illinois protocol. The hierarchical subblock did not do quite as well as the single bus subblock protocol (when compared to the respective non-subblock protocol) because it lacked a bus cache state that the conventional protocol possessed, and also because the smaller processor caches sometimes induced bad subblock protocol behavior on some of the benchmarks.

Finally in Chapter 7, we examined the issue of data placement in hierarchical systems when physical memory is distributed among the system's clusters. We found that providing a simple method for allocating data in the cluster nearest the processor was effective in reducing upper bus traffic in all three applications studied. Unfortunately, the reductions in upper bus traffic led to consistent performance gains for only one of the three applications. The other two applications sometimes experienced performance declines because of increased processor cache misses due to changes in workings of the memory allocator, or because the application's algorithm had to be changed to accommodate cluster allocation. Cluster allocation could not be used on some of the benchmarks we used in the previous chapters because they do not statically assign work to threads. Instead, threads commonly take a portion of the work to be done off of a global queue; when a thread finishes with its work portion, it takes the next available chunk of work. While this method of work assignment reduces the chance that threads (hence processors) will be idle when there is work left to be done, it makes it very difficult for the programmer or compiler to determine statically which thread will access which shared data structures. In order to take advantage of cluster allocation, at least some of the application's work must be statically scheduled. Thus, in order to get the best performance, an application writer might have to weigh the need to balance the amount of work among threads against the need to reduce the load on the memory system.

8.2 Future work

There are several avenues for extending this work:

- At times, the subblock protocol's policy of transferring more subblocks than just the one requested decreases application performance, because the other subblocks are not used. This problem could be remedied by having either the programmer or the compiler pass hints to the memory system that transferring small amounts of data in response to a memory request would give the best performance. The hints would not be required for the program to execute correctly.

- An important factor in hierarchical system performance is the size of non-processor, i.e. bus caches. Bus caches should be large enough so that replacements in the bus cache do not force the replacement of useful blocks in the processor caches. A problem with this goal is that if the lower level caches are large or numerous, then the upper level cache become quite large in order to avoid such replacements. One way to save space is to cache tags only above the processor level, which the Data Diffusion Machine [HLH92] does. The drawback to this approach is that read requests which miss in the local cluster must be transmitted down to another cluster, if the data is dirty with respect to memory. A compromise approach would cache all tags to maintain inclusion, but only cache some of the data corresponding to the tags present. However, this would complicate the cache coherence protocol further.

- A variation on the hierarchical subblock protocol would be to use a larger block size (perhaps 256 bytes) at the upper level and have upper level subblock size be the same as lower level block size. This would save expensive tag bits in the upper level caches, and improve the running times of applications with spatial locality. On the down side, such a protocol would be more complex than the subblock protocol we used. In addition, using a large allocation size could increase miss rates for those applications with poor spatial locality.

Appendix A

SINGLE BUS SUBBLOCK PROTOCOL SPECIFICATION

Table A.1: Direct Processor Actions

Processor Cache Status	Processor Request			
Line Status / Subblock Status	READ	WRITE	SWAP	REPLACE
INVALID/ Invalid	Allocate line; Q Read◇	Allocate Line Q Read Ex	Allocate Line Q Read Ex	None
CLEAN SHARED/ Invalid	Q Read	Q Read Ex	Q Read Ex	None
VALID EXCL./ Invalid	Q Read	Q Read Ex	Q Read Ex	None†
DIRTY SHARED/ Invalid	Q Read	Q Read Ex	Q Read Ex	None†
CLEAN SHARED/ Clean Shared	Hit	Q Invalidate	Q Invalidate	None
VALID EXCL./ Clean Shared	Hit	Hit → Dirty	Hit → Dirty	None†
DIRTY SHARED/ Clean Shared	Hit	Q Invalidate	Q Invalidate	None†
CLEAN SHARED/ Dirty Shared	Hit	Q Invalidate	Q Invalidate	Write Back
DIRTY SHARED/ Dirty Shared	Hit	Q Invalidate	Q Invalidate	Write Back
VALID EXCL./ Dirty	Hit	Hit	Hit	Write Back
DIRTY SHARED/ Dirty	Hit	Hit	Hit	Write Back

†If other blocks are dirty, they are written back.
◇'Q' means to enqueue a request on the bus.

Table A.2: Induced Processor Cache Actions

Processor Cache Status Line Status / Subblock Status	Other Request		
	Read	Read Ex	Invalidate
INVALID/Invalid	Snarf → **CLEAN SHARED** → Clean Shared‡	-	-
CLEAN SHARED/ Invalid	Snarf → Clean Shared‡	-	-
VALID EXCL./Invalid	-	-	-
DIRTY SHARED/ Invalid	Snarf → Clean Shared‡	-	-
CLEAN SHARED/ Clean Shared	Supply* ⊙	Supply* →Invalid	→ Invalid

‡The snarf occurs if and only if the tag matches, and another cache (not memory) supplies the subblock. Note that all writebacks go directory to memory. One optimization that we did not do would be to snarf writebacks to memory.

*At most one cache will supply the subblock.

⊙ The supplier also sends along all other non-exclusive subblocks in that block.

Table A.3: Induced Processor Cache Actions (continued)

Processor Cache Status Line Status / Subblock Status	Other Request		
	Read	Read Ex	Invalidate
VALID EXCLUSIVE/ Clean Shared	Supply ⊙ → **DIRTY SHARED**	Supply →Invalid	Error
DIRTY SHARED/ Clean Shared	Supply* ⊙	Supply* →Invalid	→ Invalid
CLEAN SHARED/ Dirty Shared	Supply* ⊙	Supply* →Invalid⋄	→ Invalid⋄
DIRTY SHARED/ Dirty Shared	Supply* ⊙	Supply* →Invalid⋄	→ Invalid⋄
VALID EXCLUSIVE/ Dirty	Supply ⊙ →**DIRTY SHARED** →Dirty Shared	Supply →Invalid	Error
DIRTY SHARED/ Dirty	Supply ⊙ →Dirty Shared	Supply →Invalid	Error

*At most one cache will supply the subblock.

⊙ The supplier also sends along all other non-exclusive subblocks in that block.

⋄The requester will be responsible for writing the subblock back

Table A.4: Processor cache response to successful request

<table>
<tr><th rowspan="2">Processor Cache Status
Line / Subblock</th><th colspan="3">Successful Request</th></tr>
<tr><th>Read</th><th>Read Ex</th><th>Invalidate</th></tr>
<tr><td rowspan="2">INVALID/Invalid</td><td>(Cache to Cache)
→ CLEAN SHARED
→ Clean Shared</td><td>(Cache to Cache)
→ DIRTY SHARED
→ Dirty</td><td rowspan="2">N/A</td></tr>
<tr><td>(From Memory)
→ VALID EXCL.
→ Clean Shared</td><td>(From Memory)
→ VALID EXCL.
→ Dirty</td></tr>
<tr><td rowspan="2">CLEAN SHARED/
Invalid</td><td>(Cache to Cache)
→ Clean Shared</td><td>(Cache to Cache)
→ DIRTY SHARED
→ Dirty</td><td rowspan="2">N/A</td></tr>
<tr><td>(From Memory)
→ Clean Shared</td><td>(From Memory)
→ DIRTY SHARED
→ Dirty</td></tr>
<tr><td rowspan="2">VALID EXCL./
Invalid</td><td>(Cache to Cache)
→ CLEAN SHARED
→ Clean Shared</td><td>(Cache to Cache)
→DIRTY SHARED
→ Dirty</td><td rowspan="2">N/A</td></tr>
<tr><td>(From Memory)
→ Clean Shared</td><td>(From Memory)
→ Dirty</td></tr>
</table>

Table A.5: Processor cache response to successful request (continued)

Processor Cache Status	Successful Request		
Line / Subblock	Read	Read Ex	Invalidate
DIRTY SHARED/ Invalid	→ Clean Shared	→ Dirty	N/A
CLEAN SHARED/ Clean Shared	N/A	N/A	→ **DIRTY SHARED** → Dirty
VALID EXCL./ Clean Shared	N/A	N/A	N/A
DIRTY SHARED/ Clean Shared	N/A	N/A	→ Dirty
DIRTY SHARED/ Dirty Shared	N/A	N/A	→ Dirty

Appendix B

HYBRID ADAPTIVE GRAPHS

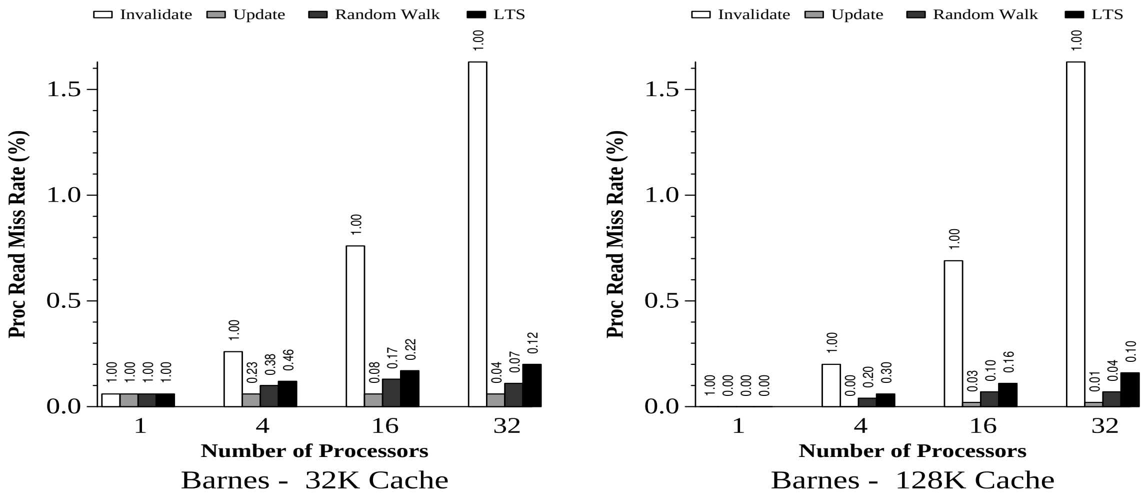


Figure B.1: Read miss rates for Barnes (with snarfing)

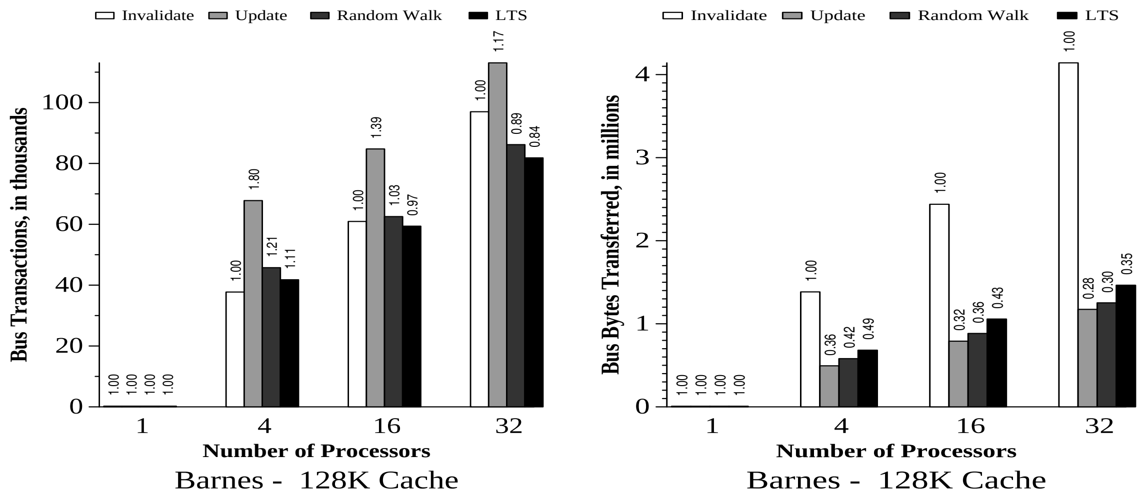


Figure B.2: Bus transactions (L) and Bytes Transferred for Barnes (with snarfing)

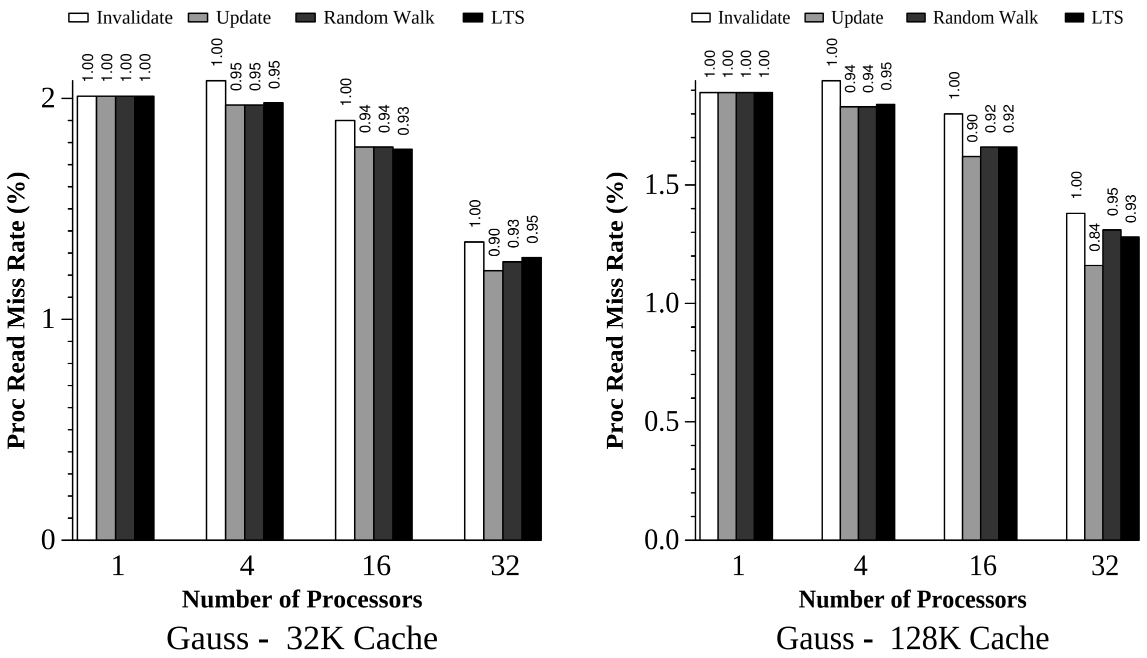


Figure B.3: Read miss rates for Gauss (with snarfing)

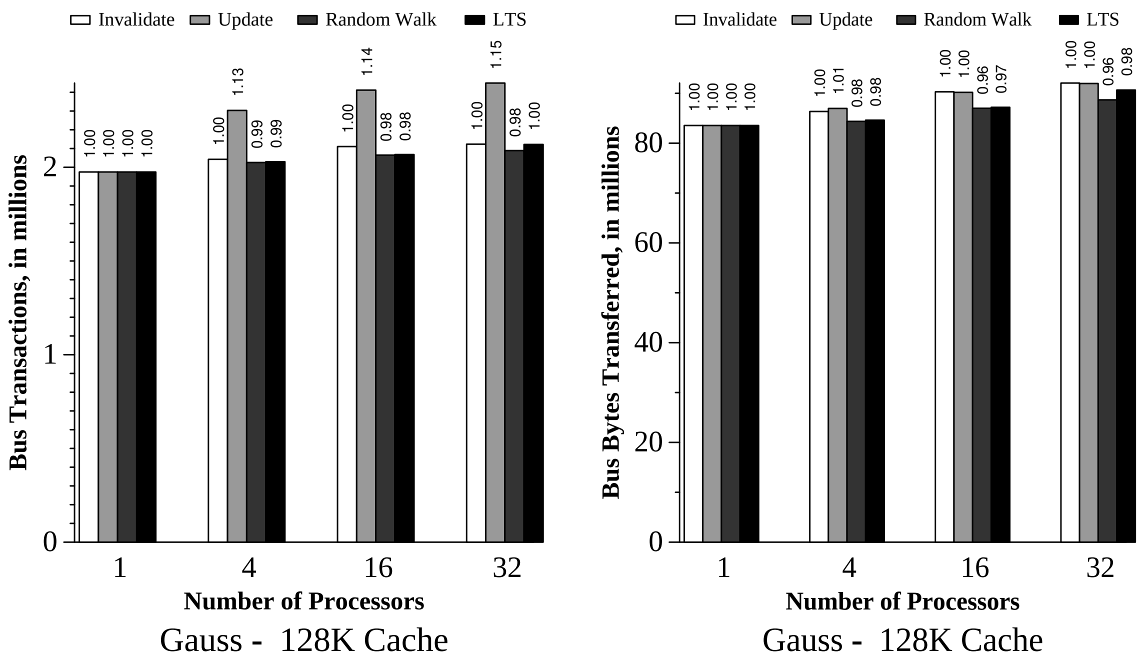


Figure B.4: Bus transactions and Bytes transferred for Gauss (with snarfing)

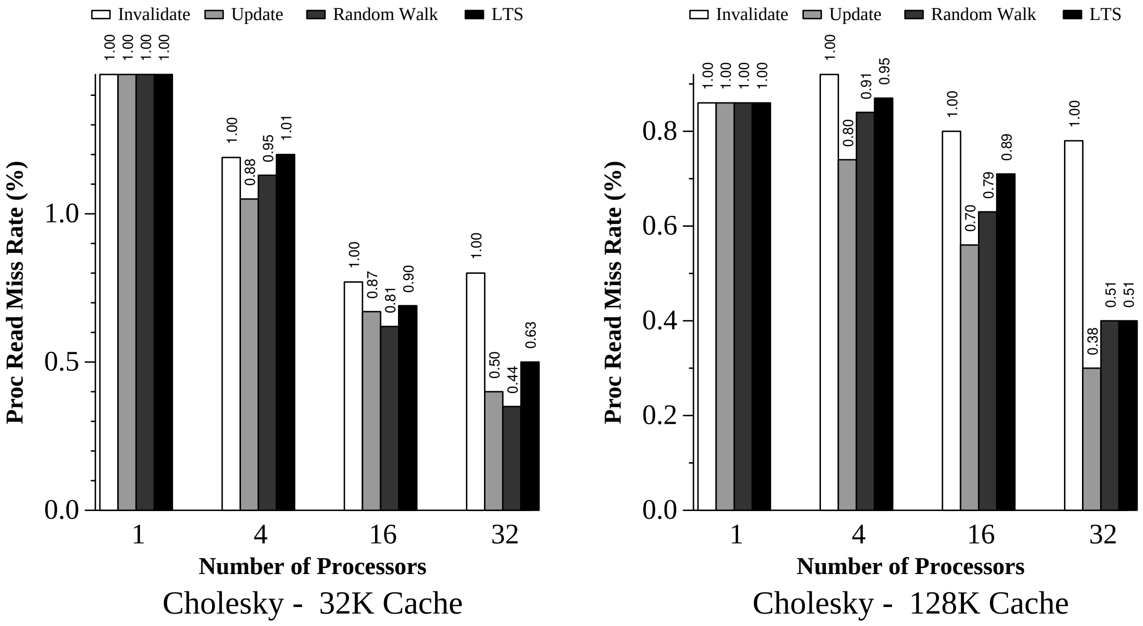


Figure B.5: Read miss rates for Cholesky (with snarfing)

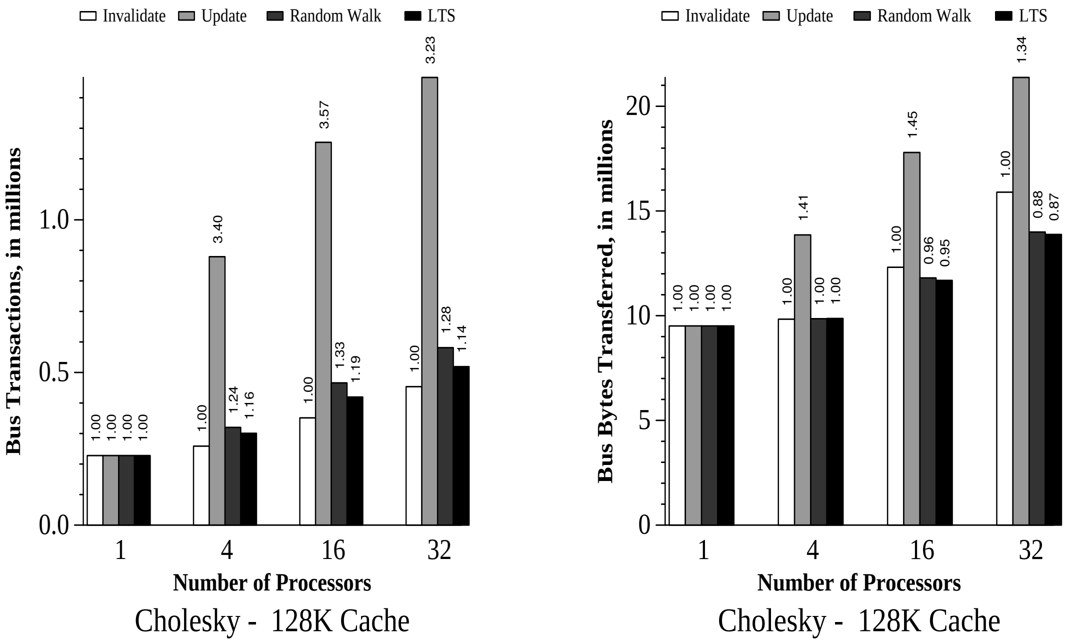


Figure B.6: Bus transactions and Bytes transferred for Cholesky (with snarfing)

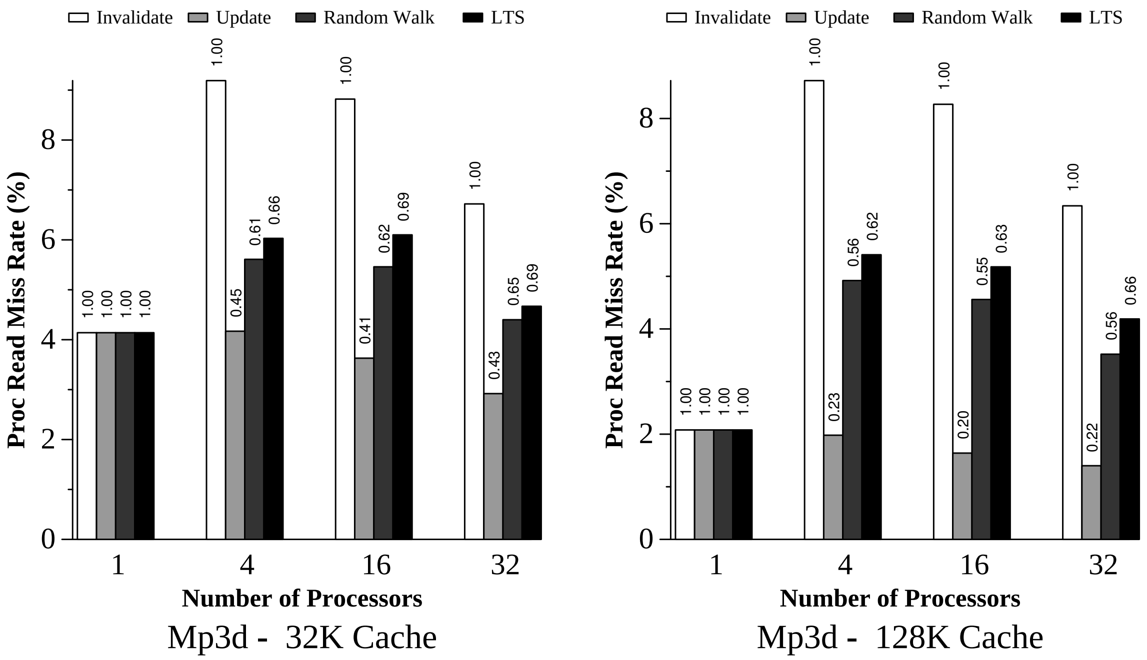


Figure B.7: Read miss rates for MP3D (with snarfing)

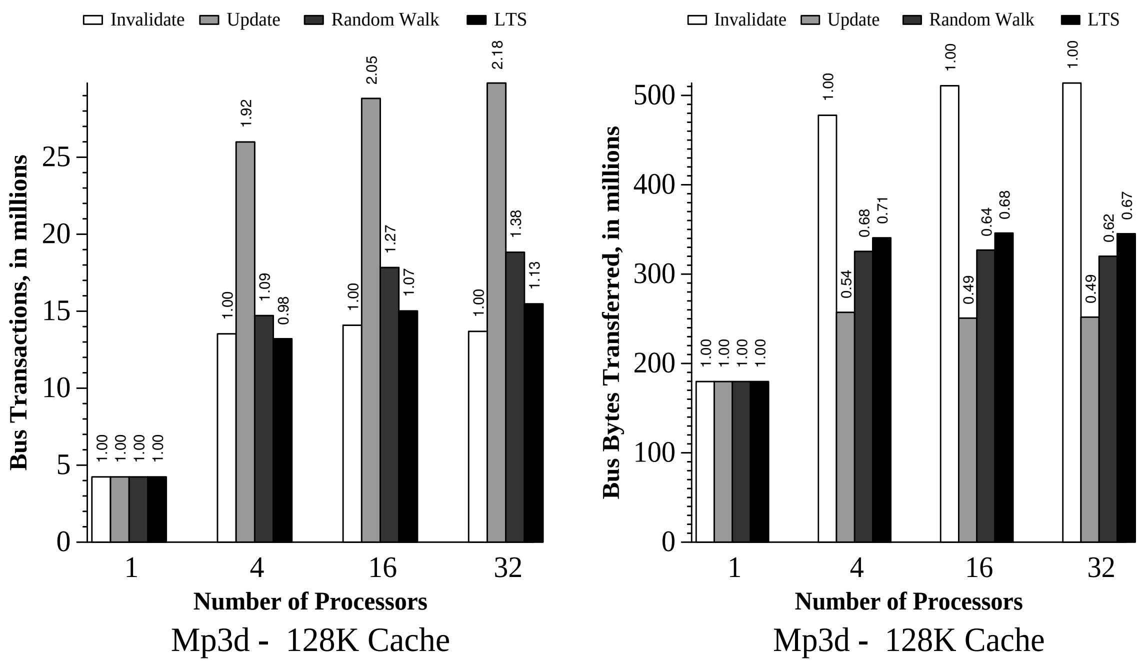


Figure B.8: Bus transactions and Bytes transferred for MP3D (with snarfing)

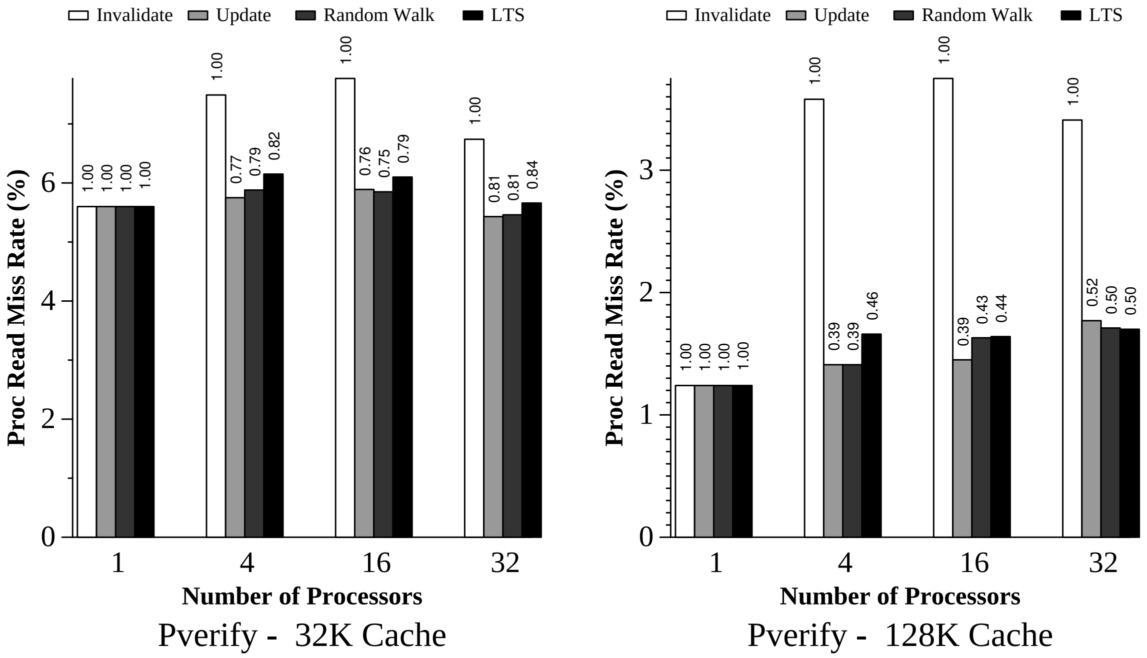


Figure B.9: Read miss rates for Pverify (with snarfing)

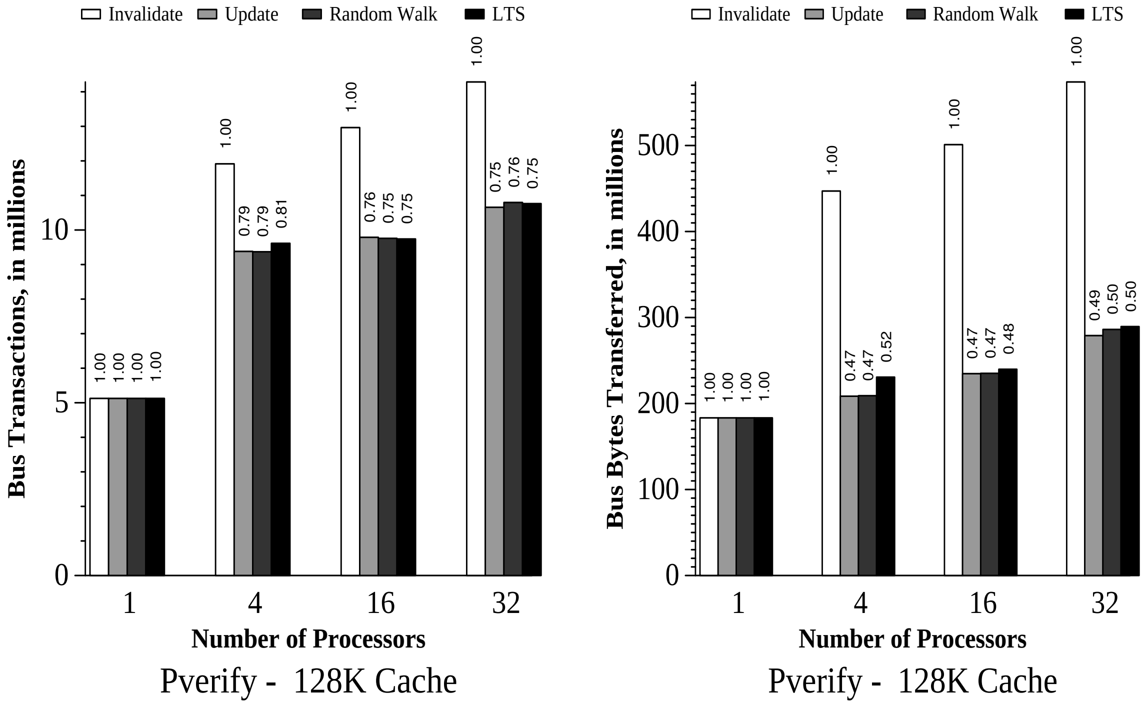


Figure B.10: Bus transactions and Bytes transferred for Pverify (with snarfing)

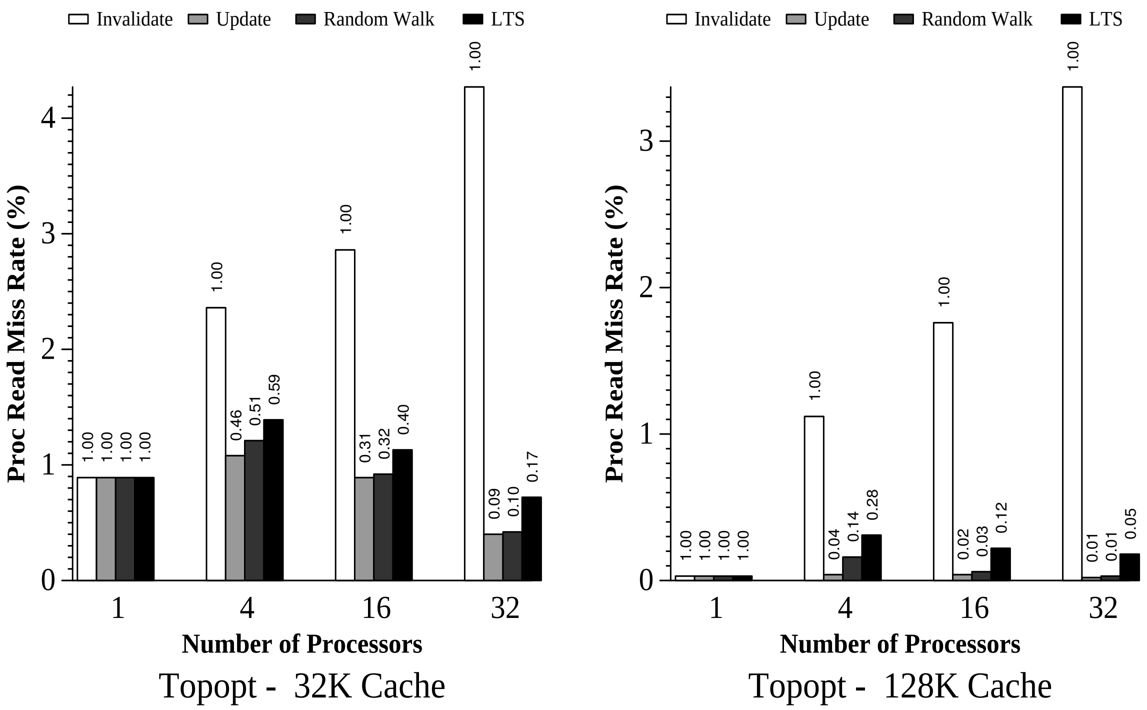


Figure B.11: Read miss rates for Topopt (with snarfing)

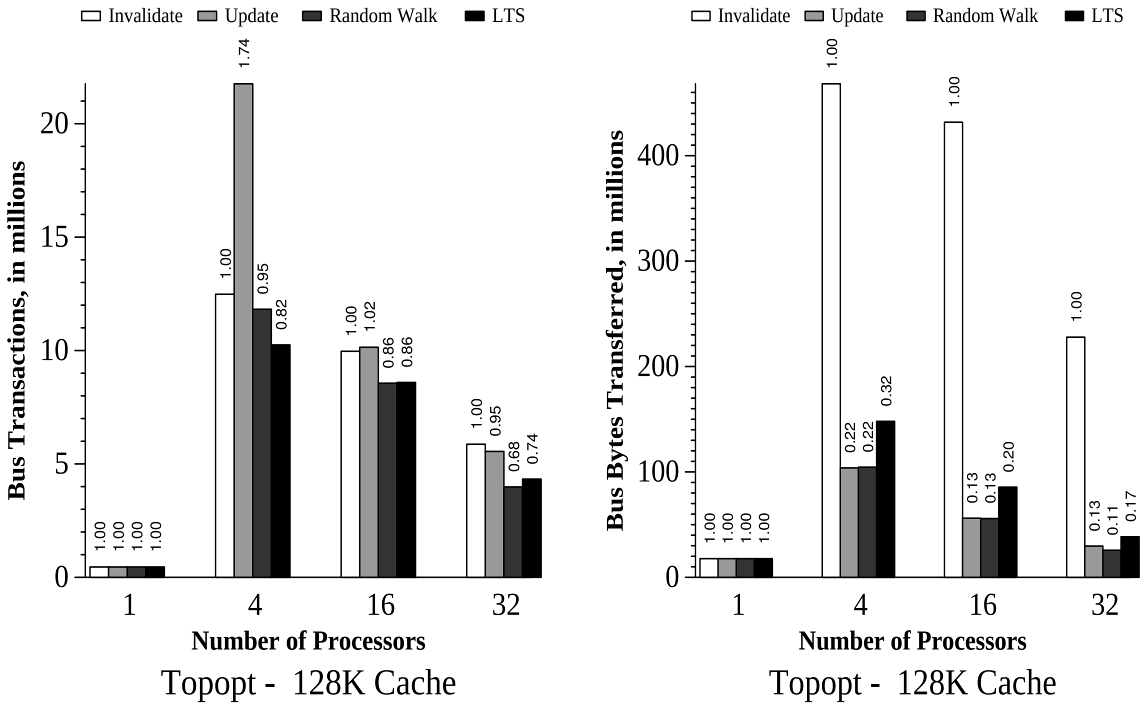


Figure B.12: Bus transactions and Bytes transferred for Topopt (with snarfing)

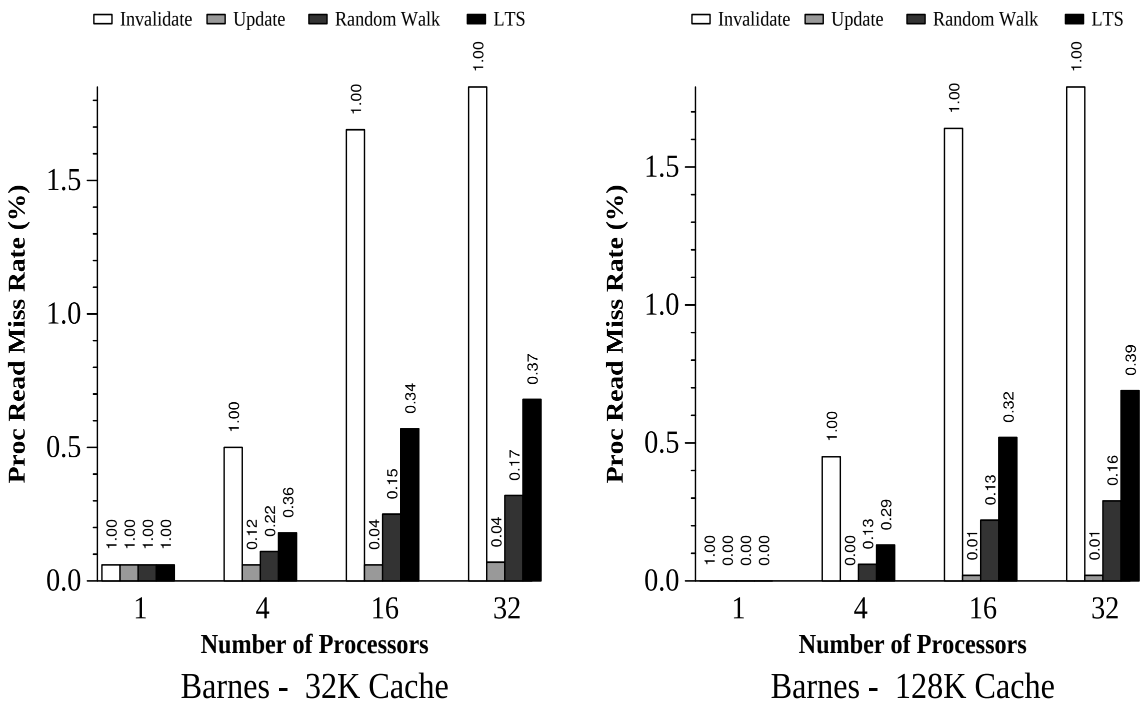


Figure B.13: Read miss rates for Barnes (without snarfing)

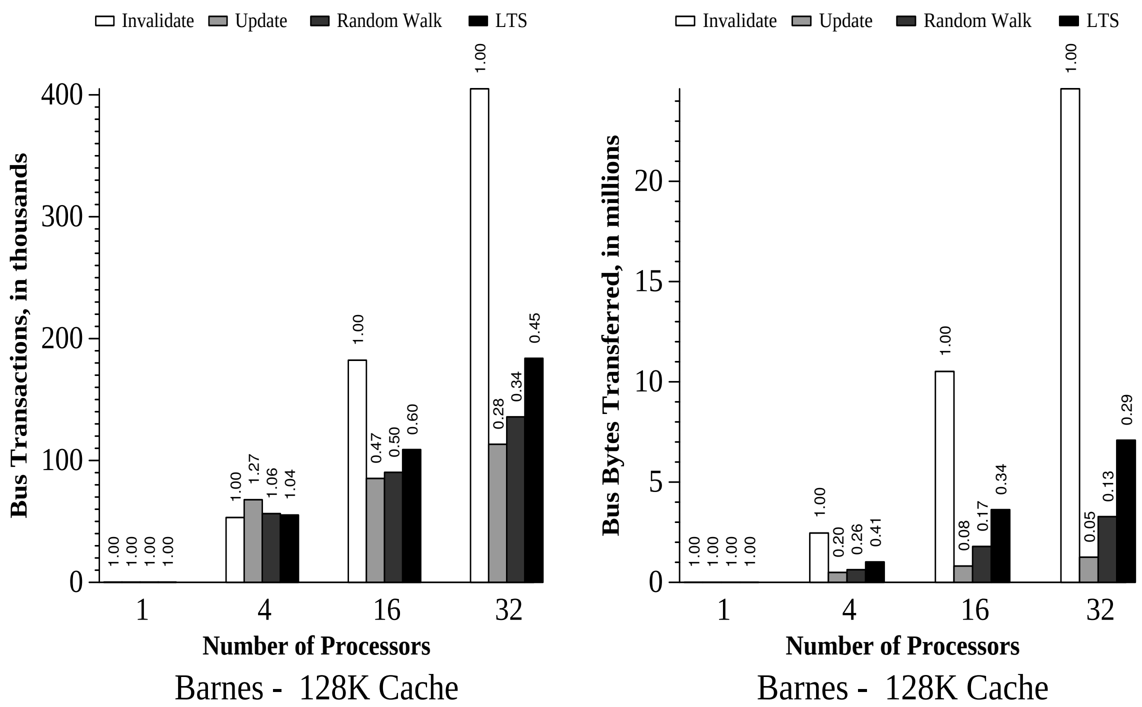


Figure B.14: Bus transactions and Bytes transferred for Barnes (without snarfing)

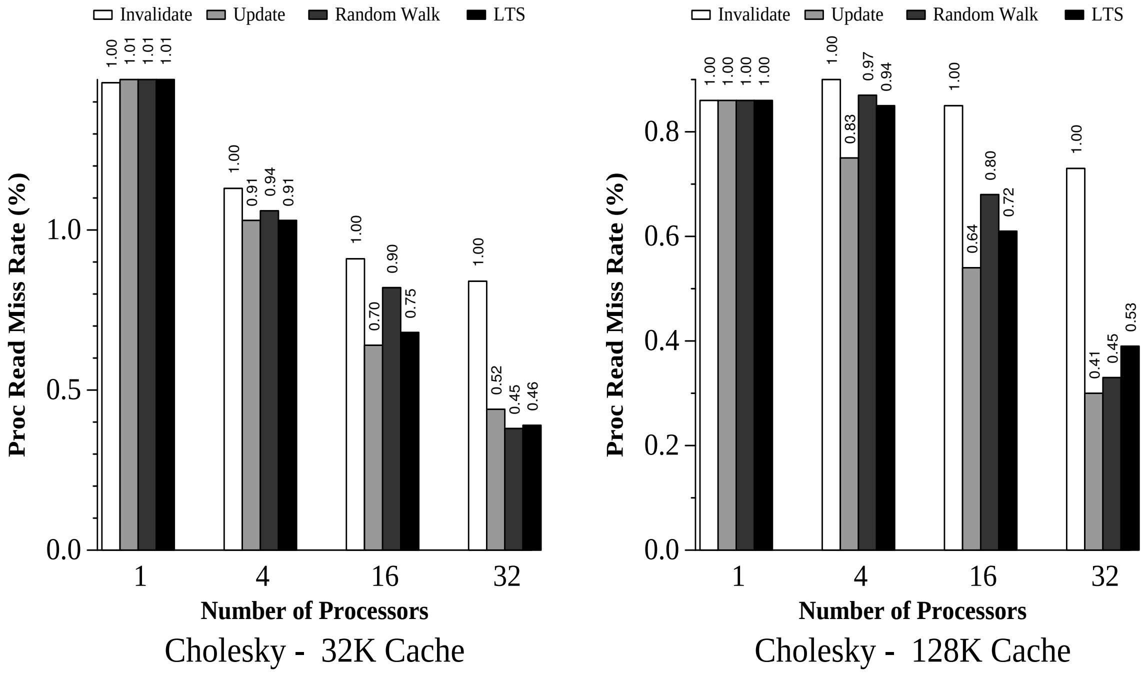


Figure B.15: Read miss rates for Cholesky (without snarfing)

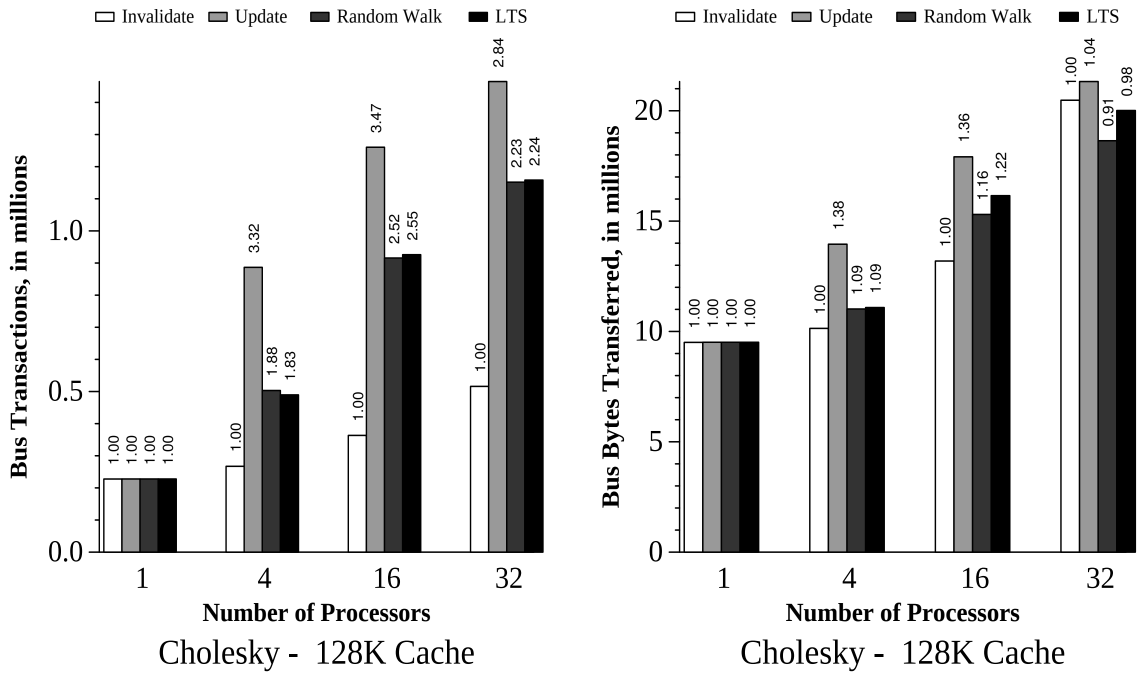


Figure B.16: Bus transactions and Bytes transferred for Cholesky (without snarfing)

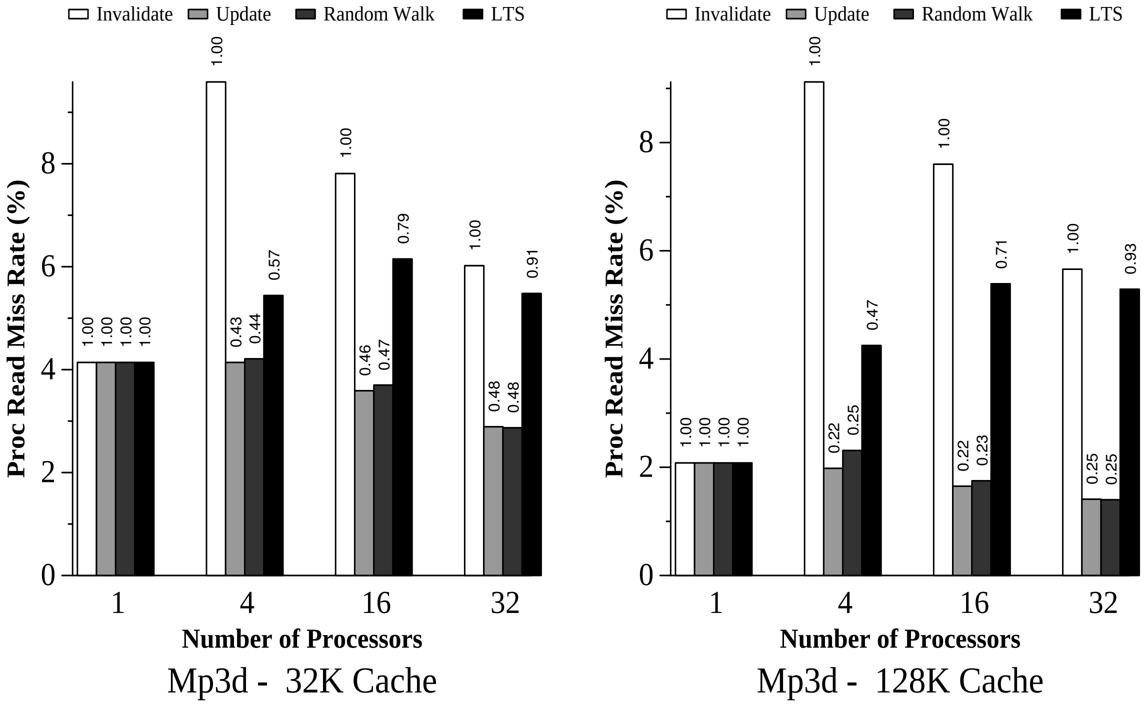


Figure B.17: Read miss rates for MP3D (without snarfing)

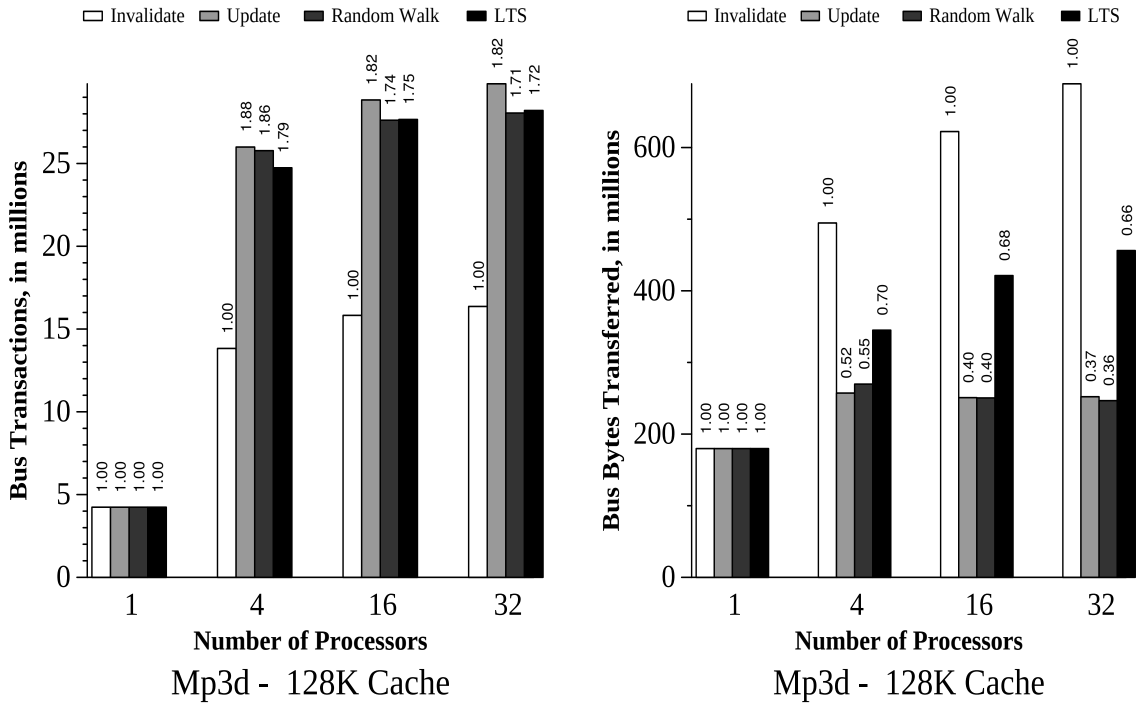


Figure B.18: Bus transactions and Bytes transferred for MP3D (without snarfing)

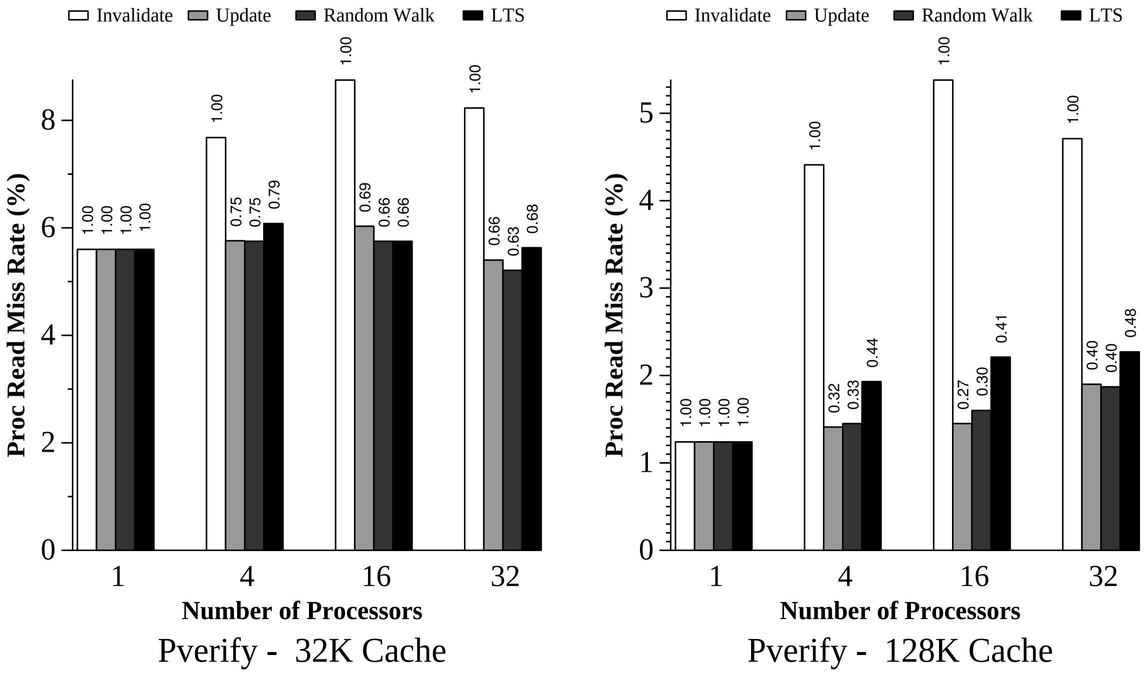


Figure B.19: Read miss rates for Pverify (without snarfing)

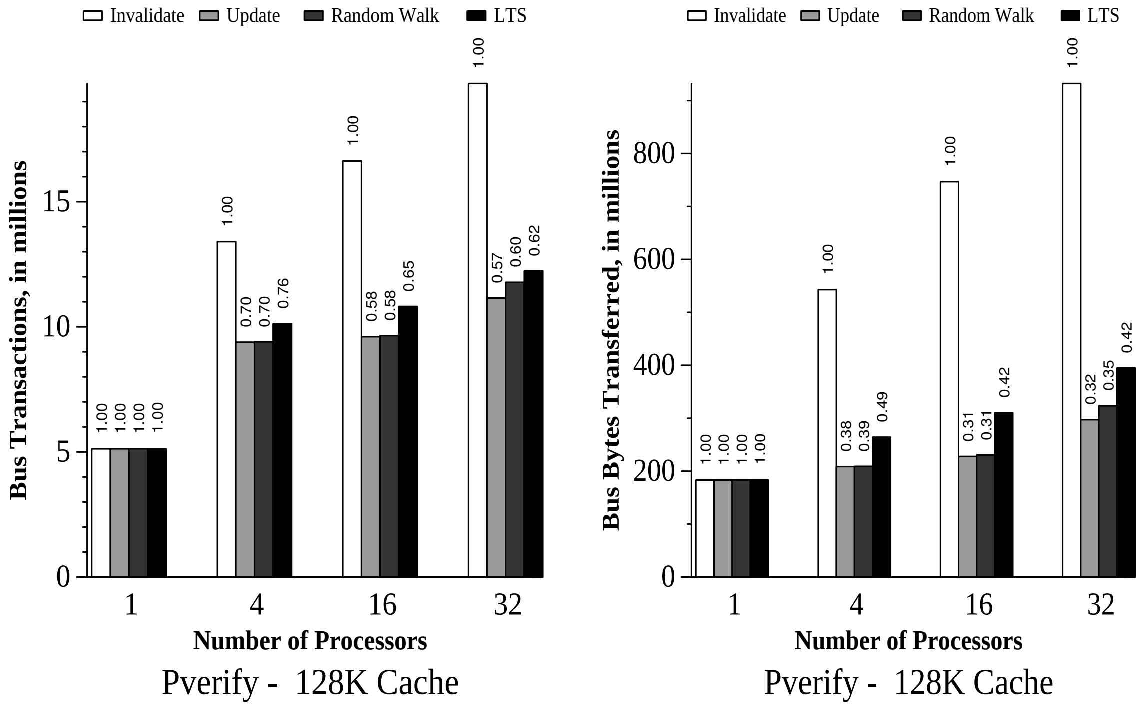


Figure B.20: Bus transactions and Bytes transferred for Pverify (without snarfing)

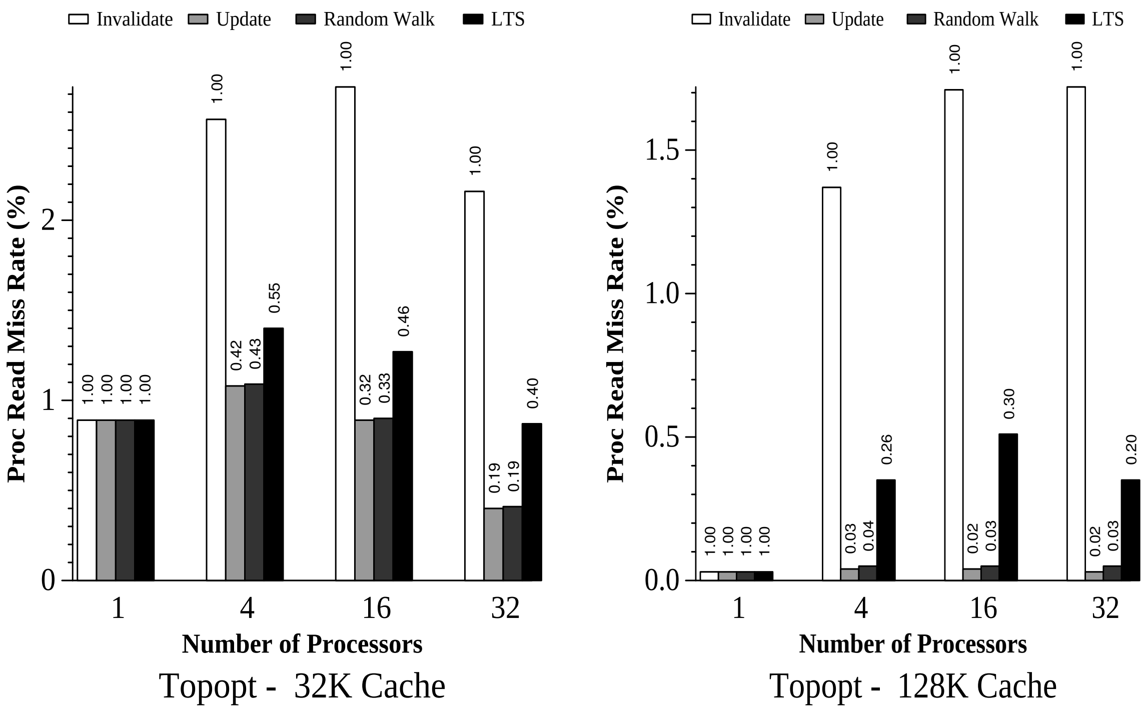


Figure B.21: Read miss rates for Topopt (without snarfing)

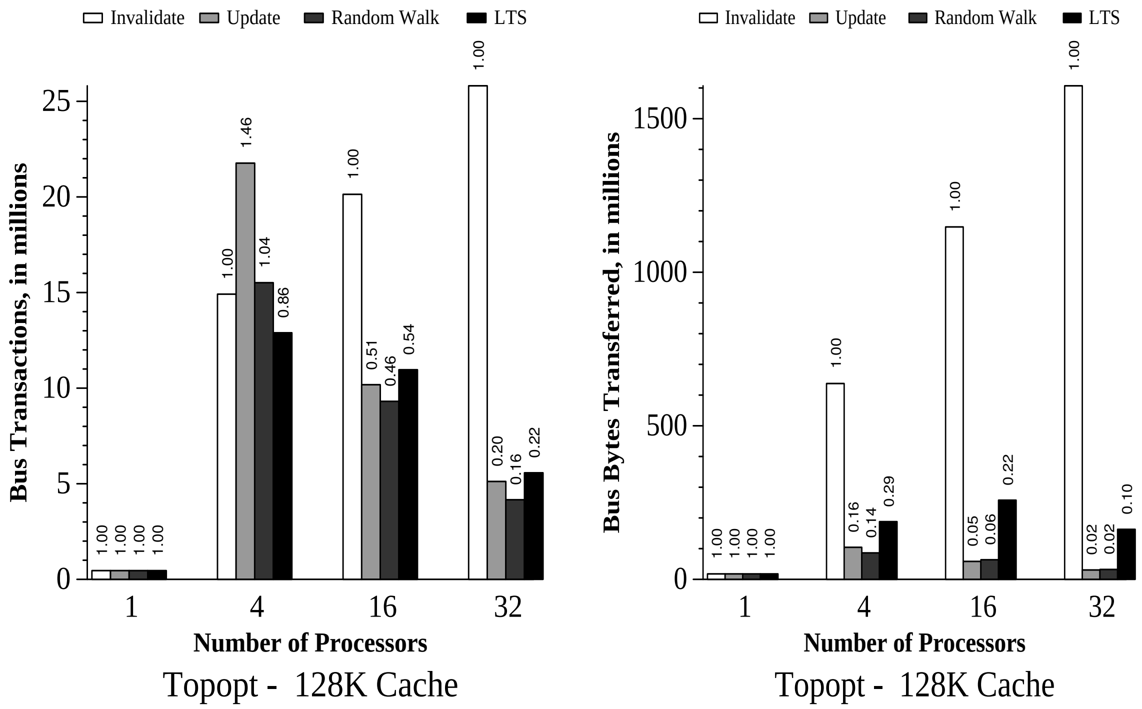


Figure B.22: Bus transactions and Bytes transferred for Topopt (without snarfing)

Appendix C

HIERARCHICAL BUS PROTOCOL SPECIFICATION

Table C.1: Direct Processor Actions

Processor Cache Status	Processor Request			
	READ	WRITE	SWAP	REPLACE
INVALID	Allocate line Pin Line† Q Read◇	Allocate Line Pin Line† Q Read Ex	Allocate Line Pin Line† Q Read Ex	N/A
CLEAN SHARED	Hit	Q Invalidate Pin Line†	Q Invalidate Pin Line†	→ INVALID
VALID EXCLUSIVE	Hit	Hit → DIRTY	Hit → DIRTY	→ INVALID
DIRTY	Hit	Hit	Hit	Q Write Back Purge; → TRANS INVALID
TRANS SHARED	Error	Error	Error	Error
TRANS EXCLUSIVE	Error	Error	Error	Error
TRANS INVALID	Error	Error	Error	Error

†Note that the status of the cache line is not changed to a transition state until *after* the request has been handled (see Table C.8).
◇'Q' means to enqueue a request on the bus.

Table C.2: Induced Processor Cache Actions - Level 1

Processor Cache Status	Other Request (From Level 1 Cache)		
	Read	Read Ex	Invalidate
INVALID	—	—	—
CLEAN SHARED	Supply Data†	Supply Data† → INVALID	→ INVALID
VALID EXCLUSIVE	Supply → CLEAN SHARED	Supply → INVALID	Error
DIRTY	Supply → CLEAN SHARED Level 2 Cache: → DIRTY SHARED	Supply → INVALID	Error
TRANS SHARED	Kill Other Request	Kill Other Request	Error
TRANS EX	Kill Other Request	Kill Other Request	Error
TRANS INVALID	Kill Other Request	Kill Other Request	Error
INVALID/ Pinned	Retry My Request Unpin Line	Retry My Request; Unpin Line	Retry My Request; Unpin Line
CLEAN SHARED/ Pinned	Supply Data† Retry My Request	Supply Data† Retry My Request; Unpin Line; → INVALID	Retry My Request; Unpin Line; → INVALID

†At most one will supply block.

Table C.3: Induced Processor Cache Actions - Level 2

Processor Cache Status	Other Request (Down From Level 2 Cache)		
	Read Down	Read Ex Down	Invalidate Down
INVALID	—	—	—
CLEAN SHARED	Supply Data;†	Supply Data† → INVALID	→ INVALID
VALID EXCLUSIVE	Supply Data;† → CLEAN SHARED	Supply Data† → INVALID	Error
DIRTY	Supply; → CLEAN SHARED	Supply; → INVALID	Error
TRANS SHARED	Error	Error	Error
TRANS EX	Error	Error	Error
TRANS INVALID	Error	Error	Error
INVALID/ Pinned	Retry My Request; Unpin Line	Retry My Request; Unpin Line	Retry My Request; Unpin Line
CLEAN SHARED/ Pinned	Retry My Request; Unpin Line	Retry My Request; Unpin Line; → INVALID	Retry My Request; Unpin Line; → INVALID

†At most one will supply block.

Table C.4: Induced Processor Cache Actions - Level 2 (Continued)

Processor Cache Status	Other Request (Down From Level 2 Cache)	
	Purge	Purge Reply
INVALID	—	—
CLEAN SHARED	→ INVALID	Reply† → INVALID
VALID EXCLUSIVE	Error	Reply† → INVALID
DIRTY	Error	Supply Data → INVALID
TRANS SHARED	Error	Error
TRANS EX	Error	Error
TRANS INVALID	Error	Error
INVALID/ Pinned	Retry My Request Unpin Line	Retry My Request Unpin Line
CLEAN SHARED/ Pinned	Retry My Request Unpin Line → INVALID Reply†	Retry My Request Unpin Line → INVALID Reply†

†A one-cycle reply is sent, not the block itself.

Table C.5: Non-Processor Cache Actions - From Level 1

Non-Processor Cache Status	Other Request (From Level 1 Cache)			
	Read	Read Ex	Invalidate	Write Back Purge
INVALID	Q Read (on Level 2) Allocate Line Pin Line†	Q Read Ex (on Level 2) Allocate Line Pin Line†	Error	Error
CLEAN SHARED	Supply* Request Succeeds	Supply* Q Invalidate† (On Level 2)	Q Invalidate† (On Level 2)	Error
VALID EXCL.	Supply; Request Succeeds	Supply; Request Succeeds	Request Succeeds	→ DIRTY OWNED
DIRTY OWNED	Supply; Request Succeeds	Supply; Request Succeeds	Error	Error
	→ DIRTY SHARED	→ VALID EX.		
DIRTY SHARED	Supply* Request Succeeds	Supply* Request Succeeds → VALID EX.	Request Succeeds → VALID EX.	Error
TRANS SHARED	—	—	—	Error
TRANS EX	—	—	—	Error
TRANS INVALID	—	—	—	Error

*At most one will supply block

†Note that the status of the cache line is not changed to a transition state until *after* the request has been handled (See Table C.9).

Table C.6: Non-Processor Cache Actions - From Level 2

Non-Processor Cache Status	Other Request (From Level 2 Cache)		
	Read	Read Ex	Invalidate
INVALID	—	—	—
CLEAN SHARED	Supply Data†	Supply Data† Q Invalidate Down → TRANS INVALID	Q Invalidate Down; → TRANS INVALID
VALID EXCLUSIVE	Q Read Down → TRANS SHARED	Q Read Down → TRANS INVALID	Error
DIRTY OWNED	Supply Data Request Succeeds → CLEAN SHARED Write Back (upwards)	Supply Data Request Succeeds → INVALID	Error
DIRTY SHARED	Supply Data† Request Succeeds → CLEAN SHARED Write Back (upwards)	Supply Data† Request Succeeds → TRANS INVALID Q Invalidate Down	Error
TRANS SHARED	Kill Other Request	Kill Other Request	Error
TRANS EX	Kill Other Request	Kill Other Request	Error
TRANS INVALID	Kill Other Request	Kill Other Request	Error

†At most one will supply block.

Table C.7: Non-Processor Cache Actions - From Level 3

Non-Processor Cache Status	Other Request (From Level 3 Cache)		
	Read Down	Read Ex Down	Invalidate Down
INVALID	—	—	—
CLEAN SHARED	Supply Data†	Supply Data† Q Invalidate Down Request Succeeds → TRANS INVALID	Q Invalidate Down; Request Succeeds; → TRANS INVALID
VALID EXCLUSIVE	Q Read Down → TRANS SHARED	Q Read Down → TRANS INVALID	Error
DIRTY OWNED	Supply Data Request Succeeds Do Writeback → CLEAN SHARED	Supply Data Request Succeeds → INVALID	Error
DIRTY SHARED	Supply Data Request Succeeds Do Writeback → CLEAN SHARED	Supply Data Request Succeeds Q Invalidate Down → TRANS INVALID	Error
TRANS SHARED	Kill Other Request	Kill Other Request	Error
TRANS EX	Kill Other Request	Kill Other Request	Error
TRANS INVALID	Kill Other Request	Kill Other Request	Error

†At most one will supply block.

Table C.8: Processor Cache Action in Response to Handled Bus Request

Request Type	Bus Action		
	Request Enqueued Higher	Request Canceled	Request Succeeds
Read	→ TRANS SHARED	Unpin Block; → INVALID; Retry Instruction; Level 2 Cache: → INVALID†	Unpin Block; → CLEAN SHARED; Restart CPU
Read Ex	→ TRANS EX	Unpin Block; → INVALID; Retry Instruction Level 2 Cache: → INVALID†	Unpin Block; → DIRTY; Restart CPU
Invalidate	→ TRANS EX	Unpin Block; → INVALID; Retry Instruction; Level 2 Cache: → CLEAN SHARED†	Unpin Block; → DIRTY; Restart CPU;
Write Back Purge (Replacement)	Error	Error	→ INVALID

†Level 2 cache is changed only if this cancelled request was enqueued higher (at some point).

Table C.9: Non-Processor Cache Action in Response to Handled Bus Request

Request Type	Bus Action		
	Request Enqueued Higher	Request Canceled	Request Succeeds
Read	→ TRANS SHARED	Unpin Block; Q Canceled Request Down; Level 3 Cache: → INVALID†	Unpin Block; → CLEAN SHARED; Q Request Succeeds Down
Read Ex	→ TRANS EX	Unpin Block; → INVALID; Q Canceled Request Down; Level 3 Cache: → INVALID†	Unpin Block; → VALID EXCL.; Q Request Succeeds Down
Invalidate	→ TRANS EX	Unpin Block Q Canceled Request Down; Level 3 Cache: → CLEAN SHARED†	Unpin Block; → VALID EXCL.; Q Request Succeeds Down
Write Back Purge (Replacement)	Error	Error	→ INVALID

†Level 3 cache (if it exists) is changed only if this cancelled request was enqueued higher (at some point).

Table C.10: Non-Processor Cache Purge Actions

Non-processor Line Status	Actions	
	Replacement From This Cache	Request Down Handled Action
INVALID	Error	N/A
READ SHARED	Q Invalidate Down → TRANS INVALID	Q Invalidate Down → TRANS INVALID
VALID EXCLUSIVE	Q Purge Reply (Down) → TRANS INVALID	Q Purge Reply (Down) → TRANS INVALID
DIRTY OWNED	Q Write Back Purge (up) → TRANS INVALID	Q Write Back Purge (up) → TRANS INVALID
DIRTY SHARED	Q Write Back Purge (up) Q Purge (down) → TRANS INVALID†	Q Write Back Purge (up) Q Purge (down) → TRANS INVALID†
TRANS SHARED	Error	Error
TRANS EXCLUSIVE	Error	Error
TRANS INVALID	Error	Error

†The transition to INVALID is delayed until both upper and lower requests are serviced.

Appendix D

HIERARCHICAL BUS SUBBLOCK PROTOCOL SPECIFICATION

In this appendix, we give the protocol tables for hierarchical subblock protocol. The protocol is a blend of the single bus subblock protocol given in Appendix A and the hierarchical protocol as described in Appendix C. The single bus subblock protocol is very similar to this protocol at the cluster level. As such, Tables A.1 through A.5 also specify some of the lower level actions of the hierarchical protocol, except that "Write Back" actions are done differently in the hierarchical case. In particular, Tables A.2 and A.3 have an additional column, shown in Table D.1. Table D.2 shows how processor caches respond to requests from processors in other clusters. The rest of the tables are derived from those of the standard hierarchical protocol, and are presented in the same order as found in Appendix C. Some tables in Appendix C do not have a corresponding table in this Appendix, since the hierarchical subblock protocol is designed for two level systems only.

Table D.1: Induced Processor Cache Actions

Processor Cache Status Block Status / Subblock Status	Response to Cluster Write Back request
INVALID/Invalid	—
CLEAN SHARED/Invalid	Read data† → **DIRTY SHARED**/Dirty Shared
VALID EXCLUSIVE/Invalid	Read data† → **DIRTY SHARED**/Dirty Shared
DIRTY SHARED/Invalid	Read data† → Dirty Shared
CLEAN SHARED/Clean Shared	Read data† → **DIRTY SHARED**/Dirty Shared
VALID EXCLUSIVE/Clean Shared	Read data† → **DIRTY SHARED**/Dirty Shared
DIRTY SHARED/Clean Shared	Read data† → Dirty Shared
CLEAN SHARED/Dirty Shared	Error
DIRTY SHARED/Dirty Shared	Error
VALID EXCLUSIVE/Dirty	Error
DIRTY SHARED/Dirty	Error

†At most one cache will change its state

Table D.2: Induced Processor Cache Actions - (Level 2)

Processor Cache Status	Other Request		
Block / Subblock	Read Down	Read Ex Down	Invalidate Down
Any/Invalid	—	—	—
CLEAN SHARED/ Clean Shared	Supply* $\oplus$	Supply* $\rightarrow$Invalid	$\rightarrow$ Invalid
VALID EXCLUSIVE/ Clean Shared	Supply $\oplus$ $\rightarrow$ **DIRTY SHARED**	Supply $\rightarrow$Invalid	Error
DIRTY SHARED/ Clean Shared	Supply* $\oplus$	Supply* $\rightarrow$Invalid	$\rightarrow$ Invalid
CLEAN SHARED/ Dirty Shared	Supply* $\oplus$	Supply* $\rightarrow$Invalid$\diamond$	$\rightarrow$ Invalid$\diamond$
DIRTY SHARED/ Dirty Shared	Supply* $\oplus$	Supply* $\rightarrow$Invalid$\diamond$	$\rightarrow$ Invalid$\diamond$
VALID EXCLUSIVE/ Dirty	Supply $\oplus$ $\rightarrow$**DIRTY SHARED** $\rightarrow$Clean Shared‡	Supply $\rightarrow$Invalid$\diamond$	Error
DIRTY SHARED/ Dirty	Supply $\oplus$ $\rightarrow$Clean Shared‡	Supply $\rightarrow$Invalid$\diamond$	Error

*At most one cache will supply the block.

$\diamond$The requester will be responsible for writing the subblock back.

‡The subblock will be written back to memory.

$\oplus$The requester sends back all non-exclusive subblocks. All dirty subblocks are written back to main memory. The responder has no Dirty Shared subblocks following the transaction.

Table D.3: Induced Processor Cache Actions - Level 2 (Continued)

Processor Cache Status	Other Request	
Block / Subblock	Purge	Purge Reply
INVALID/Invalid	—	—
CLEAN SHARED/ Any	→INVALID	→INVALID
VALID EXCLUSIVE/ Any	Reply† →INVALID	Reply† →INVALID
DIRTY SHARED/ Any	Reply† →INVALID	Reply† →INVALID

† The cache responds with its Dirty or Dirty Shared subblocks (if any)

Table D.4: Non-Processor Cache Actions - From Level 1

Non-Processor Cache Status Block/Subblock	Other Request (From Level 1 Cache)			
	Read	Read Ex	Invalidate	Write Back Purge
Any/ Invalid	Q Read (on Level 2); Allocate Block⊗; Pin Block†	Q Read Ex (on Level 2) Allocate Block⊗; Pin Block†	Error	Error
Any/ Clean Shared	Supply*; Request Succeeds	Supply*; Q Invalidate† (On Level 2)	Q Invalidate† (On Level 2)	Error
Any/ Valid Exclusive	Supply; Request Succeeds	Supply; Request Succeeds	Request Succeeds	→ **DIRTY SHARED** → Dirty Owned
Any/ Dirty Owned	Supply Request Succeeds ⋄	Supply Request Succeeds	Error	Error
TRANS SHARED	—	—	—	Error
TRANS EX	—	—	—	Error
TRANS INVALID	—	—	—	Error

*At most one will supply block

†Note that the status of the cache line is not changed to a transition state until *after* the request has been handled (See Table D.6).

⊗ A block is allocated only if the block does not exist in the cache.

⋄ The requesting cache will place the subblock in the Dirty Shared state.

Table D.5: Non-Processor Cache Actions (From Level 2)

Non-Processor Cache Status Block / Subblock	Other Request (From Level 2 Cache)		
	Read	Read Ex	Invalidate
Any/Invalid	Snarf† →**CLEAN SHARED**‡ →Clean Shared	—	—
Any/ Clean Shared	Supply*	Supply* Q Invalidate Down →TRANS INVALID →Invalid	Q Invalidate Down →TRANS INVALID →Invalid
Any/ Valid Exclusive	Q Read Down → **DIRTY SHARED**	Q Read Ex Down →Invalid	Error
Any/ Dirty Owned	Supply; →Clean Shared Write Back Subblock	Supply◇ →Invalid	Error

†The snarf occurs if and only if the tag matches, and another cache (not memory) supplies the subblock.
‡If the block's previous state was **DIRTY SHARED**, it remains **DIRTY SHARED**. *At most one cache will supply the block.
◇The requester will be responsible for writing the subblock back.

Table D.6: Non-Processor Cache Action in Response to Handled Bus Request

Request Type	Bus Action: Memory Request Enqueued	Bus Action: Request Canceled	Bus Action: Request Succeeds
Read	→ TRANS SHARED	Unpin Block; Q Canceled Request Down; Revert Status† →Invalid	Unpin Block; Q Request Succeeds Down; →CLEAN SHARED‡ →Clean Shared
Read Ex	→ TRANS EX	Unpin Block; → INVALID; Q Canceled Request Down; Revert Status† →Invalid	Unpin Block; Q Request Succeeds Down; →DIRTY SHARED →Valid Exclusive
Invalidate	—	Unpin Block; Q Canceled Request Down; Revert Status† →Invalid	Unpin Block; Q Request Succeeds Succeeds Down →DIRTY SHARED →Valid Exclusive
Write Back Purge (Replacement)	Error	Error	→ INVALID

†The block status reverts to the state it was in previous to the request.
‡If the previous state was DIRTY SHARED, the block status becomes DIRTY SHARED.

Table D.7: Non-Processor Cache Purge Actions

Non-processor Subblock Status	Replacement Action From This Cache
Invalid	None
Clean Shared	Q Purge (down) $\rightarrow$ TRANS INVALID
Valid Exclusive	Q Purge Reply (Down) $\rightarrow$ TRANS INVALID
Dirty Owned	Q Write Back Purge (up) $\rightarrow$ TRANS INVALID

Note: Blocks can contain subblocks of various states. When a non-processor cache replaces a block, it must gather all dirty (with respect to memory) subblocks from below it, combine them with the subblocks that are dirty in the non-processor cache, and write them back to memory. Thus if a block contains both Dirty Owned and Valid Exclusive subblocks, the Write Back Purge operation (due to the Dirty Owned subblocks) will be delayed until the reply (if any) to the Purge Reply request has been received.

Bibliography

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM TOCS*, 4(4):273–298, November 1986.

[AB94] Craig Anderson and Jean-Loup Baer. Design and evaluation of a subblock cache coherence protocol for bus-based multiprocessors. Technical Report 94-05-02, University of Washington, 1994.

[AB95] Craig Anderson and Jean-Loup Baer. Two techniques for improving the performance of bus-based multiprocessors. In *International Symposium on High-Performance Computer Architecture*, pages 264–275, 1995.

[ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk Johnson, and David Krantz *et. al.* The MIT Alewife machine: Architecture and performance. In *Proc. of 22nd Int. Symp. on Computer Architecture*, pages 2–13, 1995.

[And90] Thomas Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[Arc88] James Archibald. A cache coherence approach for large multiprocessor systems. In *International Conference on Supercomputing*, pages 337–345, 1988.

[BAD89] Eugene D. Brooks III, Tim S. Axelrod, and Gregory A. Darmohray. The Cerberus multiprocessor simulator. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 384–390. SIAM, 1989.

[BBW92] Jonathan Bertoni, Jean-Loup Baer, and Wen-Hann Wang. Scaling shared-bus multiprocessors with multiple buses and shared caches: a performance study. *Microprocessors and Microsystems*, 16(7):339–50, 1992.

[BDCW91] Eric Brewer, Chrysanthos Dellarocas, Adrian Colbrook, and William Weihl. PROTEUS: A high performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT, September 1991.

[BDNS93] Mats Brorsson, Fredrik Dahlgren, Hakan Nilsson, and Per Stenstrom. The CacheMire test bench — a flexible and effective approache for simulation of multiprocessors. In *Proceedings of the 26th Annual Simulation Symposium*, pages 41–49, 1993.

[Bed94] Robert Bedichek. The Meerkat multicomputer: Tradeoffs in multicomputer architecture. Technical Report 94-06-06, University of Washington, 1994.

[Bed95] Robert Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 14–24, 1995.

[BLRC94] Brian Bershad, Dennis Lee, Theodore Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proc. of ASPLOS VI*, pages 158–170, 1994.

[BW88] Jean-Loup Baer and Wen-Han Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 73–80, 1988.

[BW92] Jean-Loup Baer and Wen-Han Wang. Architectural choices for multi-level hierarchies. Technical Report 87-01-04, University of Washington, 1992.

[CD93] Yung-Syau Chen and Michel Dubois. Cache protocols with partial block invalidations. In *7th International Parallel Processing Symposium*, pages 16–24, 1993.

[CF78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27:1112–1118, December 1978.

[CF93] Alan Cox and Robert Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proc. of 20th Int. Symp. on Computer Architecture*, pages 98–108, 1993.

[CK94] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proc. of ASPLOS IV*, pages 224–234, 1991.

[Con94] Convex Computer Corporation. *Convex Exemplar Scalable Parallel Processing System*, 1994.

[CYS+93] Michel Cekleov, David Yen, Pradeep Sindhu, Jean-Marc Frailong, and Jean Gastinel *et. al.* SPARCcenter 2000: Multiprocessing for the 90's! In *COMPCON Spring 1993*, pages 345–353, 1993.

[Dar88] Gregory A. Darmohray. Gaussian techniques on shared-memory multiprocessors. Master's thesis, University of California, Davis, April 1988.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proc. of Int. Conference on Computer Design*, pages 522–525, 1992.

[DL92] Czarek Dubnicki and Thomas LeBlanc. Adjustable block size coherent caches. In *Proc. of 19th Int. Symp. on Computer Architecture*, pages 170–180, 1992.

[DN87] Srinivas Devadas and A. Richard Newton. Topological optimization of multiple level array logic. *IEEE Transactions on Computer-Aided Design*, November 1987.

[Egg89] Susan Eggers. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessor*. PhD thesis, University of California, Berkeley, 1989.

[EJ91] Susan Eggers and Tor Jeremiassen. Eliminating false-sharing. In *Proc. of Int. Conf. on Parallel Processing*, pages I–377–381, 1991.

[EK88] Susan Eggers and Randy Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 373–382, 1988.

[EK89] Susan Eggers and Randy Katz. Evaluating the performance of four snooping cache coherence protocols. In *Proc. of 16th Int. Symp. on Computer Architecture*, pages 2–15, 1989.

[FIR93] S. Frank, H. Burkhardt III, and J. Rothnie. The KSR-1: bridging the gap between shared memory and MPPs. In *Proc. of Spring 1993 COMPCON*, pages 285–294, February 1993.

[Fuj90] Richard Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[Goo87] James Goodman. Coherency for multiprocessor virtual caches. In *Proc. of 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 72–81, 1987.

[Gus92] David Gustavson. The Scalable Coherence Interface and related standards projects. *IEEE Micro*, pages 10–22, February 1992.

[GW88] James Goodman and Phillip Woest. The Wisconsin Multicube: A new large-scale cache coherent multiprocessor. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 422–431, 1988.

[GW92] Anoop Gupta and Wolf-Dietrich Weber. Cache invalidation patterns in shared memory multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, 1992.

[GW94] Mike Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, volume I, pages 134–143, 1994.

[GWM90] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *International Conference on Parallel Processing*, pages (I) 312–321, 1990.

[Her93] Stephen Herrod. *Tango Lite: A Multiprocessor Simulation Environment.* Stanford University Computer Systems Laboratory, November 1993.

[HLH92] Erik Hagersten, Anders Landin, and Seif Haridi. DDM - a cache-only memory architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[HP90] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[Jer95] Tor E. Jeremiassen. *Using Compile-Time Analysis and Transformations to Reduce False Sharing on Shared Memory Multiprocessors.* PhD thesis, University of Washington, 1995.

[Jou93] Norm Jouppi. Cache write policies and performance. In *Proc. of 20th Int. Symp. on Computer Architecture*, pages 191–201, 1993.

[KEL91] Eric Koldinger, Susan Eggers, and Henry Levy. On the validity of trace-driven simulation for mulitprocessors. In *Proc. 18st Int. Symp. on Computer Architecture*, pages 244–253, 1991.

[KEW+85] R. Katz, S. Eggers, D. Wood, C. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proc. 12th Int. Symp. on Computer Architecture*, pages 276–283, 1985.

[KKEL90] David Keppel, Eric Koldinger, Susan Eggers, and Henry Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–47, 1990.

[KLMO91] Anna Karlin, Kai Li, Mark Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 41–55, 1991.

[KMRS86] Anna Karlin, Mark Manasse, Larry Rudolf, and Daniel Sleator. Competitive snoopy caching. In *Proc. of 27th Symposium on Foundations of Computer Science*, pages 244–254, 1986.

[Lil93] David Lilja. Cache coherence in large-scale multiprocessors: Issues and comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance. In *Proc. 19th Annual Symposium on Computer Architecture*, pages 92–103, June 1992.

[LT88] Tom Lovett and Shreekant Thakkar. The Symmetry multiprocessor system. In *Proc. of Int. Conf. on Parallel Processing*, pages 303–310, 1988.

[LZGS84] Edward Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevick. *Quanititative System Performance.* Prentice-Hall, Inc., 1984.

[McC84] E. McCreight. The Dragon computer system: An early overview. Technical report, Xerox Corp., 1984.

[MDWS87] H-K. T. Ma, S. Devadas, R. Wei, and A. Sangiovanni-Vincentelli. Logic verification algorithms and their parallel implementation. In *Proceedings of the 24th Design Automation Conference*, pages 283–290, 1987.

[NO94] Basem A. Nayfeh and Kunle Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proc. 21st Int. Symp. on Computer Architecture*, pages 166–175, 1994.

[PP84] Mark Papamarcos and Janak Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proc. of 11th Int. Symp. on Computer Architecture*, pages 348–354, 1984.

[RHL+93] Steven Reinhardt, Mark Hill, James Larus, Alvin Lebeck, James Lewis, and David Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, 1993.

[Rid] Ridge Computers. *Ridge 32 User's Guide.*

[SBS93] Per Stenstrom, Mats Brorsson, and Lars Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proc. of 20th Int. Symp. on Computer Architecture*, pages 109–118, 1993.

[SD87] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proc. of 45th Int. Symp. on Computer Architecture*, pages 234–233, 1987.

[Sez94] André Seznec. Decoupled sector caches: Conciliating low tag implementation cost and low miss ratio. In *Proc. of 21th Int. Symp. on Computer Architecture*, pages 384–393, 1994.

[SFG+93] Pradeep Sindhu, Jean-Marc Frailong, Jean Gastinel, Michel Cekleov, Leo Yuan, Bill Gunning, and Don Curry. XDBus: A high-performance, consistent, packet-switched VLSI bus. In *COMPCON Spring 1993*, pages 338–344, 1993.

[SJF91] Craig Stunkel, Bob Janssens, and W. Kent Fuchs. Address tracing for parallel machines. *IEEE Computer*, 24(1):31–8, January 1991.

[SJG92] Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc. 19th Int. Symp. on Computer Architecture*, pages 80–91, June 1992.

[SJGH93] Jaswinder Pal Singh, Truman Joe, Anoop Gupta, and John L. Hennessy. An emprical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors. In *Supercomputing*, pages 214–225, 1993.

[SR84] Z. Segall and L. Rudolph. Dynamic decentralized cache schemes for an MIMD parallel processor. In *Proc. of 11th Int. Symp. on Computer Architecture*, pages 340–347, 1984.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, pages 5–44, March 1992.

[TCS92] Charles Thacker, David Conroy, and Lawrence Stewart. The Alpha demonstration unit: A high-performance multiprocessor for software and chip development. *Digital Technical Journal*, 4(4):51–65, 1992.

[TM91] Milo Tomašević and Veljko Milutinović. A simulation study of snoopy cache coherence protocols. In *Proc. of the 25th Hawaii Int. Conference on System Sciences*, pages 427–436, 1991.

[VF94a] Jack Veenstra and Robert Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 201–207, January 1994.

[VF94b] Jack Veenstra and Robert Fowler. The prospects for on-line hybrid coherency protocols on bus-based multiprocessors. Technical Report 490, University of Rochester, 1994.

[VSLW91] Zvonko Vranesic, Micahel Stumm, David Lewis, and Ron White. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Computer*, 24(1):72–79, January 1991.

[WB72] W. A. Wulf and C. G. Bell. C.mmp - a multi-mini processor. In *Proc. Fall Joint Computer Conference*, pages 765–777, December 1972.

[Wil87] Andrew Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proc. of 14th Int. Symp. on Computer Architecture*, pages 244–252, 1987.

[WTP+92] Andrew Wilson, Marc Teller, Thoams Probert, Dyung Le, and Richard LaRowe. Lynx/Galactica Net: A distributed, cache coherent multiprocessor system. In *Proc. of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 416–426, 1992.

[YTB92] Q. Yang, G. Thangadurai, and L. Bhuyan. Design of an adaptive cache coherence protocol for large scale multiprocessors. *IEEE TPDS*, 3(3):281–293, May 1992.

[ZB92] Richard Zucker and Jean-Loup Baer. A performance study of memory consistency models. In *Proc. of 19th Int. Symp. on Computer Architecture*, pages 2–12, 1992.

Vita

Craig S. Anderson was born in Walnut Creek, California on May 25, 1965. After graduating from Washington High School in 1983, he attended the University of California at Berkeley. He received his Bachelor of Science in Electrical Engineering and Computer Science with High Honors in May, 1988. Craig received the Master of Science degree in Computer Science and Engineering in March of 1993, and his Doctor of Philosophy in Computer Science in November, 1995, both at the University of Washington.