# Understanding Application Performance on Shared Virtual Memory Systems

Liviu Iftode, Jaswinder Pal Singh and Kai Li
Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

## Abstract

Many researchers have proposed interesting protocols for shared virtual memory (SVM) systems, and demonstrated performance improvements on parallel programs. However, there is still no clear understanding of the performance potential of SVM systems for different classes of applications. This paper begins to fill this gap, by studying the performance of a range of applications in detail and understanding it in light of application characteristics.

We first develop a brief classification of the inherent data sharing patterns in the applications, and how they interact with system granularities to yield the communication patterns relevant to SVM systems. We then use detailed simulation to compare the performance of two SVM approaches—Lazy Released Consistency (LRC) and Automatic Update Release Consistency (AURC)—with each other and with an all-hardware CC-NUMA approach. We examine how performance is affected by problem size, machine size, key system parameters, and the use of less optimized program implementations. We find that SVM can indeed perform quite well for systems of at least up to 32 processors for several nontrivial applications. However, performance is much more variable across applications than on CC-NUMA systems, and the problem sizes needed to obtain good parallel performance are substantially larger. The hardware-assisted AURC system tends to perform significantly better than the all-software LRC under our system assumptions, particularly when realistic cache hierarchies are used.

## 1 Introduction

Since shared virtual memory (SVM) was proposed [15], much research has concentrated on improving its performance through both relaxed consistency models [5, 13, 3] and architectural support. The goal has been to deliver performance close to that of a hardware cache-coherent shared address space machine at the same scale. While many SVM systems have been proposed, their performance on real applications is not well understood so far. Papers that have proposed protocols or extensions have evaluated them with only a small set of usually quite simple programs. The programs, problem sizes and machine parameters used have also been limited and different across studies, making comparisons difficult. Despite a few comparative studies [6, 1] it is not well understood whether the SVM approach works well for important classes of applications, whether the approaches that use a small amount of hardware support have substantial advantages over all-software implementations, and how the answers to these questions are affected by architectural and application parameters. This paper begins to answer these questions, using two different SVM systems as examples.

To obtain good coverage, we use ten applications from the SPLASH-2 suite [17] that display a range of different communication patterns and granularities. We first classify their sharing patterns in a manner relevant for SVM systems. We then use detailed simulation to compare performance under two SVM approaches—Lazy Release Consistency (LRC) and Automatic Update Release Consistency (AURC)—with that under the hardware-coherent CC-NUMA (cache-coherent non-uniform memory access) approach. LRC is an all-software scheme, while AURC relies on a little hardware assistance to automatically propagate local writes to a remote memory using memory-mapped communication. There are clear performance tradeoffs between the two. In addition to sharing patterns, we examine how problem size, machine size, key system parameters, and the degree of optimization of the parallel programs affect the performance of the two systems.

We find that SVM can indeed perform quite well for systems with 16-32 processors for several nontrivial applications at reasonable problem sizes. However, performance is much more variable across applications than with CC-NUMA, and the problem sizes needed to obtain good parallel performance are substantially larger. Smaller problems often yield very poor performance, even though they perform well with CC-NUMA. We also found that the hardware-assisted AURC performs significantly better than LRC, at least under our next-generation architectural assumptions, and the performance gap between these two protocols increases when realistic cache hierarchies are used.

Section 2 introduces the two SVM systems we use in our study. The simulated architectures are presented in Section 3, and the applications and the SVM-oriented classification of their sharing patterns are described in Section 4. Section 5 provides an overview of the performance results, assuming an optimistic local cache hierarchy that tends to benefit the all-software LRC system, and Section 6 analyzes performance in detail. The effects of using a more realistic

In Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996.

cache hierarchy are examined in Section 7. Section 8 briefly examines robustness with respect to program optimizations (particularly data structuring and placement), as well as sensitivity to some detailed machine parameters. Finally, Section 9 summarizes our results and concludes the paper.

## 2 Shared Virtual Memory

The basic idea in SVM [15] is to exploit the virtual memory mechanism to emulate a simple invalidation-based cache coherence protocol at the granularity of pages instead of cache lines. The large granularity of communication and coherence (an entire page) can affect the performance of any SVM by causing false sharing and fragmentation. False sharing occurs when two processors access logically unrelated data that happen to fall on the same page and at least one of them writes the data, causing the coherence mechanism to ping-pong the page between the processors. Fragmentation causes unnecessary traffic when a processor needs only a part of the page but has to fetch the entire page. Relaxed consistency models help to alleviate these problems by allowing the system to delay the coherence activity from when the modifications happen until the next synchronization point. For example, release consistency [7] is a relaxed model which guarantees memory consistency only after explicit synchronization points, distinguished as either *acquire* or *release* operations. The two SVM systems we examine, LRC and AURC, both use the same lazy form of release consistency, in which shared pages are made coherent at a processing node only when that processor performs an acquire synchronization operation. Both also use a software protocol, implemented in page-fault handlers and through synchronization operations, to fetch pages and make them coherent.

To fully exploit release consistency in reducing the ping-ponging due to false sharing, an SVM protocol must allow different copies of a page to be written locally by multiple writers at a time (in the same synchronization interval). The key problem is to provide a mechanism by which the writes performed on the different copies of the page will be merged when a consistent copy of the page is needed (after the next synchronization point). Our two SVM systems differ in the mechanisms they use for detecting, propagating and merging these writes to a page. While LRC uses all-software mechanisms, AURC relies on a small amount of hardware support, available in some commodity and research network interfaces, to do this automatically. Let us examine the two protocols.

### Lazy Release Consistency

LRC is an all-software multiple writer protocol. Every writer locally records the changes it makes to every shared page between two synchronization events. To detect these changes, a processor saves a copy of the page (called the *twin*) before writing to it. Later, by comparing the current (changed) copy with the twin, the processor detects the changed locations for the page and records them in a structure called a *diff*. At an acquire event, a processor's pages that have been modified by other processors before the previous release event are invalidated. At the first access to a page following an invalidation, the page fault handler must collect the diffs for the page from the processors that have produced them, and apply these diffs locally in the proper causal order to reconstitute the coherent page. Further details can be found in [13] and [12].

Such a delayed, multiple-writer protocol has several benefits. By delaying coherence until synchronization, it reduces the number of times data and coherence messages are communicated and hence the number of times expensive software protocol overhead is incurred. Fragmentation in communication is also reduced, not only by allowing multiple modifications of a page to be combined but also by sending diffs (i.e. only the modified words plus the diff information overhead) instead of whole pages. The disadvantages of the all-software scheme are the costs associated with the diff processing. First, the creation of a twin, the creation of a diff and the application (merging) of diffs to a page all have substantial software overhead. These operations also pollute the processor's caches. Second, the acquiring processor may have to visit more than one other processor to obtain diffs. Third, even when consecutive diffs of the same page (from multiple synchronization intervals) are obtained from one place, they have to be obtained as separate diffs and applied individually by the faulting processor.

### Automatic Update Release Consistency

The AURC protocol [9] is also based on a lazy release consistency model, but it takes advantage of the automatic update hardware feature of virtual memory-mapped network interfaces [4, 14] to detect, propagate and merge writes by different processors to a page. By snooping on the memory bus, the automatic update mechanism can propagate updates to shared data at a word granularity with no software overhead, as long as a mapping has been established between the local page and a remote page to which the writes are propagated. The mappings are established as soon as more than one processor accesses a shared page.

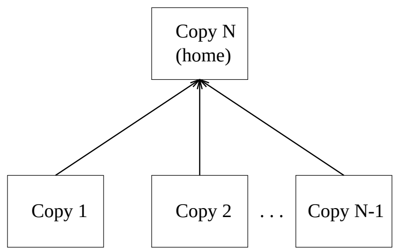


Figure 1: AU-based Write Collection.

The basic approach is to have a *home* memory for every shared page (as in hardware coherent machines) and to set up mappings such that writes to other copies of the page are automatically propagated to the home (see Figure 1). The home copy is thus always kept up-to-date by this mechanism, and serves to merge the writes to the page. The AURC software protocol then keeps other copies coherent through page invalidations at acquire points, in a manner similar to the LRC protocol. Only now, upon a page fault a coherent version of the page can be obtained directly from the home, which is guaranteed to have an up-to-date copy. There is also the potential for reduced page faults, since the home processor does not fault on the page. In fact, if only two processors share a page, then a specialized version of the protocol kicks in for that page: The two copies are mapped to each other, all writes are propagated in hardware between them in either direction, and no page faults are incurred. The regular protocol kicks in once more than two processors share a page.

The tradeoffs between AURC and LRC are simple. The automatic propagation of writes in AURC eliminates the use of diffs and hence the software overhead of diff processing at both the producer and the consumer. On the other hand, the automatic propagation of all writes to a mapped page by AURC will generate higher communication traffic than the purely invalidation-based LRC, just as update protocols generate more traffic in hardware-coherent machines. The AURC protocol also fetches the whole page on a miss (not just the modified data). This usually increases traffic, unless diffs are large in the LRC case. The major tradeoff, then, is between bandwidth and end-point overhead.

## 3 Simulation Environment

We use execution-driven simulation driven by the TangoLite reference generator. We couple TangoLite to three different types of memory system simulators: one for LRC, one for AURC, and one for hardware-coherent CC-NUMA. Since the focus of our study is to understand the interactions of protocols with applications, not to study a particular platform, we assumed a common, generic but aggressive architectural model.

The basic model assumes a 300 MHz 1 CPI processor, with 200 ns memory latency, 1 MB of single-level two-way set associative cache with a 64 byte line size, and an 8 cache-line deep merging write-buffer. The write buffer merges writes and retires them in FIFO order. Since it is potentially important, particularly in AURC, its size is varied in Section 8. All instructions in our simulated processors as well as cache hits cost 1 CPU cycle.

The use of a large, one-cycle access cache is of course unrealistic. We choose it for the bulk of our discussion because it removes the cache size as a parameter for sensitivity. In fact, it tends to slightly underestimate the speedups for AURC and hardware CC-NUMA, and to benefit the all-software LRC. The reason is that the diffs performed in LRC not only cause misses themselves but also pollute the cache. A large cache allow the diffs to not perturb the application's working set in the first-level cache too much. However, to see the effects of a more realistic cache hierarchy, in Section 7 we present results for a two-level hierarchy with a 16KB direct-mapped first-level data cache and a 1MB two-way set-associative second-level cache (with 10-cycle hit cost).

The cache policy can be programmed on a per-page basis to be either write-back with write-allocate or write-through with write-around. The latter is used only for shared pages and only in the AURC case. In the AURC simulator, we assume a memory-mapped network interface that snoops the memory bus and is connected to the I/O bus. The overhead of initiating a deliberate transfer (e.g. a page transfer) is 1 microsecond, and the overhead of initiating an automatic update is a single CPU cycle when the write buffer is not full. The effective bandwidth of a page transfer is limited by the speed of the I/O bus, which is 32 bits wide. The page size is 4KB, so a page transfer takes 1K I/O bus cycles. The network interface has a mechanism that combines consecutive automatic updates to a single network packet, whose size cannot exceed 40 words. It is similar to the network interface of the SHRIMP multicomputer [4]. The network topology is a two-dimensional mesh. The simulators model contention everywhere (buses, memory, network interface) in great detail, except for contention in the network topology itself.

We have tried to charge both the AURC and LRC protocols fairly for software operations. We model a fast interrupt handling mechanism which takes 150 cycles, a reasonable cost of 4 cycles per word for twin creation in LRC, and 8 cycles per word for diff computation or processing. These and all protocol operations are actually simulated by running the protocol handlers, and the simulator also charges for the cache misses incurred during the operations.

Since both the memory and I/O buses are involved in AURC and LRC, we use two combinations of speeds for them to examine sensitivity. We call these the *fast bus* and *slow bus* combinations. The fast bus setup is an 800MB/sec memory bus and a 256 MB/sec I/O bus, while the slow bus setup is a 400 MB/sec memory bus and a 128 MB/sec I/O bus.

The hardware CC-NUMA simulator we use is a modified version of the detailed FlashLite simulator built for the Stanford FLASH machine. It is the same as the simulator used in [8], and models a hardware rather than flexible communication controller. It uses almost exactly the same parameters as described above (with an 800MB/s memory bus; the I/O bus is irrelevant), and is used only as a reference point for comparison. The cost of a remote miss to a clean line at the home is about 180 cycles, and that of a miss to a dirty remote line is about 300 cycles.

## 4 Applications and Classifications

In this section, we first provide a general classification of data sharing and communication patterns that is relevant to understanding performance on SVM systems. We then briefly describe the applications and classify their patterns accordingly, focusing on the phases of their execution that contribute most to performance differences among LRC, AURC, and hardware CC-NUMA schemes.

### 4.1 Classification of Sharing Patterns

We start with the dominant *inherent sharing patterns* in the applications; that is, the sharing patterns at word granularity. These patterns interact with the large granularity of coherence in an SVM (a page) to construct the *induced sharing patterns* of pages, which may be very different than the inherent patterns. It is these induced sharing patterns that determine the communication actually seen in an SVM execution, exhibit false sharing and fragmentation, and hence impact performance. The induced pattern for an application is a function of its inherent sharing pattern and the spatial granularity in the address space of the read and write accesses made by processors (i.e. whether a processor tends to read or write entire pages of data or have more scattered or fine-grained access patterns). For AURC (like CC-NUMA), the communication incurred also depends on how data are allocated and distributed among processor memories, but this is irrelevant in LRC (see Section 8).

Several previous studies have proposed classifications of data objects based on access patterns, although in different contexts. Gupta and Weber [16] proposed a classification to characterize cache invalidation patterns in shared memory multiprocessors. Bennett et al [2] expanded this classification and used it to perform adaptive software cache management on distributed shared memory machines. Our classification has a different purpose, to understand the observed performance of SVM systems, and is hence different from these. Let us look at our classification of inherent patterns, access granularity and induced patterns and effects.

### 4.1.1 Inherent Sharing Patterns

The inherent sharing patterns are determined by the parallel algorithms and partitioning schemes used. There are three types of inherent sharing patterns relevant to our purposes that we usually encounter: *1P-1C* (one producer with one consumer), *1P-MC* (one producer with multiple consumers), and *Migratory*. *Migratory* sharing is a frequent *1P-1C* pattern in irregular applications where data are consecutively accessed in read-write sequences by different processors (it is also encountered in accumulations and the use of global counters in regular applications). The access patterns to task queues when enqueuing and dequeuing tasks are common examples. There is also read-only data (once initialized), which can be thought of as 0P-1C or 0P-MC.

### 4.1.2 Access Granularity

An important characteristic that helps determine how an inherent sharing pattern is converted to an induced pattern is the granularity at which the read and write accesses to shared data from different processes are interleaved in the address space (for reads, particularly to nonlocal or communicated data). In particular, does a processor tend to read (write) a whole contiguous page at a time (coarse-grained accesses), or are its reads and writes scattered at a much finer grain so it reads (writes) only parts of a page. We distinguish between *coarse*, *medium*, and *fine* grained read and write accesses (by medium we mean not very fine-grained but the contiguity is still smaller than a page). We will see applications with many different combinations of read and write access granularity, and that in many cases the access granularity depends on the relationship between the problem size and the number of processors.

Finally, some applications `malloc` lots of data objects that are each much smaller than a page. If these mallocs are performed on separate pages, then SVM protocol operates and communicates at the granularity of the `malloc`s rather than whole pages. In these cases, accesses are considered coarse-grained or fine-grained at the granularity of these malloced objects rather than whole pages.

### 4.1.3 Induced Sharing Patterns and Effects

When both read and write access patterns are coarse grained in terms of pages (or the size of mallocs), the induced communication pattern is very similar to the inherent pattern. In general, fine or medium grained read accesses to nonlocal data cause *fragmentation* (often, but not always, inducing a multiple-consumer or MC pattern). Fine- or medium-grained writes cause *false sharing* of a page (often inducing a multiple-producer or MP pattern). We shall see examples of fine-grained reads causing fragmentation in the FFT (induced 1P-MC) and Ocean (preserves 1P-1C) applications, and of fine-grained writes causing false-sharing but not fragmentation in Radix (induced MP-1C). If both the read and write access patterns of processors are fine-grained, as in many irregular applications, the induced pattern is MP-MC end exhibits both false sharing and fragmentation regardless of the inherent sharing pattern. Nonetheless, the inherent pattern is still useful in serving as a contrast to the induced pattern.

Different induced sharing are handled differently by the two protocols. With proper data placement (homes), AURC can take advantage of 1C induced patterns by making the consumer the home of that page. This avoids page misses completely in 1P-1C cases like Ocean due to the specialized protocol discussed earlier. A 1P-MC pattern allows LRC to amortize the cost of diff computation at the producer over the page misses at multiple consumers. Finally, in LRC, an induced MP pattern for a page is particularly bad because it means that means that diffs for the page will have to be obtained from multiple processors upon a page fault (especially after a barrier synchronization).

## 4.2 Applications

The applications we use include a range of scientific, engineering and graphics programs [17]. We use them as written for hardware cache-coherent systems with cache line granularity, not attempting to reprogram them for SVM other than to change the padding amount for some obvious data structures. We describe the applications themselves only very briefly, focusing mostly on their dominant sharing patterns relevant to SVM systems. More detailed descriptions can be found in the references cited in [17].

**Barnes** simulates the interactions among a system of particles over a number of time steps, using the Barnes-Hut hierarchical N-body method. The particle update and force calculation phases in the application are inherently 1P-MC sharing patterns. While the force calculation is by far the dominant component of execution time in hardware CC-NUMA systems, is not the key distinguishing phase for SVM since it affords a lot of temporal locality and data reuse. Rather, the interesting phase is the one in which the tree is built every time-step, by loading particles into it and synchronizing on space cells as needed. This phase exhibits both 1P-MC and migratory inherent sharing patterns between synchronization events. However, both read and write access patterns to the global particle and cell arrays are fine-grained. The induced pattern at page granularity is therefore MP-MC, which causes substantial fragmentation and false sharing to occur.

**Cholesky** is a sparse matrix factorization kernel with inherent sharing pattern 1P-MC (the pivot row blocks once produced are consumed by several other processors). However, the sparse structure of the matrix can lead to multiple processors' assigned blocks falling on the same page. This medium granularity of block accesses induces an MP-MC pattern, causing false sharing and fragmentation. More significant problems occur in the accesses to distributed task queues, which are inherently migratory but are fine- rather than medium-grained in both reads and writes.

**FFT** is a high-performance FFT kernel. The communication in an $n$-point FFT is essentially a matrix transposition of a $\sqrt{n}$ by $\sqrt{n}$ matrix. Matrices are distributed so that every processor is assigned a contiguous set of $n/p$ rows,

| Application | Inherent Pattern | Read Access Granularity | Write Access Granularity | Induced Pattern | Induced Effect |
|---|---|---|---|---|---|
| Barnes | migratory & 1P-MC | fine | fine | MP-MC | false sharing & fragmentation |
| Cholesky | migratory & 1P-MC | fine-medium | fine-medium | MP-MC | false sharing & fragmentation |
| FFT | 1P-1C | medium | coarse | 1P-MC | fragmentation |
| LU | 1P-MC | coarse | coarse | 1P-MC | none |
| Ocean | 1P-1C | fine | coarse | 1P-1C | fragmentation |
| Radix | 1P-1C | coarse | fine-medium | MP-1C | false sharing |
| RayTrace | migratory | fine | fine | MP-MC | false sharing & fragmentation |
| VolRend | migratory | fine | fine | MP-MC | false sharing & fragmentation |
| Water $N^2$ | migratory & 1P-MC | coarse | coarse | MP-MC | none |
| Water Spatial | 1P-MC | coarse | coarse | 1P-MC | none |

Table 1: Classifications of Sharing Patterns in the Applications.

and the source and destination matrices are reversed for every transpose. In this version of FFT, each processor in a transpose reads an $\frac{\sqrt{n}}{p}$ by $\frac{\sqrt{n}}{p}$ submatrix from every other processor and writes it into its local partition or set of rows. Consequently the write access granularity is coarse while the read access granularity depends on $n$ and $p$ (a processor reads subrows of size $\frac{\sqrt{n}}{p}$ from other processors). For typical values of these parameters, the read granularity is often medium, thus generating fragmentation but not false sharing (an induced 1P-MC pattern). However, for very large problems it may be coarse (1P-1C induced, if $\sqrt{n}/p$ is a multiple of the page size), generating neither false sharing nor fragmentation.

**LU** performs the blocked LU factorization of a dense matrix. It follows the standard cookie cutter-scatter decomposition of blocks to processors. For the bulk of the paper, we used a version with an optimized data structure for the matrix (a 4-d array) in which each block is allocated contiguously in virtual memory and the blocks assigned to a processor are allocated contiguously as well. The inherent sharing pattern of blocks is 1P-MC as in Cholesky. However each 32-by-32 block is exactly 8KB (2 pages) large, so both the read and write granularities are coarse. The induced pattern remains 1P-MC, and there is no false sharing or fragmentation. In Section 8, we will examine using a version with a more natural but less optimized data structure (a 2-d array).

**Ocean** simulates eddy currents in an ocean basin. The dominant sharing pattern is an iterative near-neighbor one, complicated a little by the use of uses a multigrid solver. Here too, we focus first on the "contiguous" version in Splash-2 in which the data in each subgrid are allocated contiguously in virtual memory using a 4-d array. The inherent pattern is mostly 1P-1C, except at the corners of partitions. Write accesses are local and coarse-grained, since a processor writes only to its own partition which is allocated contiguously; however, the nonlocal read accesses that generate communication (of border elements from neighbor partitions) are medium- or coarse-grained along row-oriented partition borders but very fine-grained (a single element) along column-oriented borders. Thus, there is little false sharing but significant fragmentation in the induced patterns. The inherent 1P-1C pattern is preserved if $n/p$ (one subrow of a partition for $n$-by-$n$ grids) is a multiple of a page; otherwise it can becomes 1P-2C as the partitions on either side of a processor's partition access elements on the same page. While in FFT the fragmentation can be eliminated by running large enough problems, in Ocean the fragmentation remains (a processors still needs only its immediate neighbor elements).

**Radix** sorts a series of integer keys in ascending order. The most dominant phase, and also the most distinctive for SVM, is they key permutation phase. The inherent pattern in this phase is 1P-1C: a processor reads its locally allocated $n/p$ contiguous keys from a source array and writes them to a destination array in a highly scattered and irregular permutation. For a uniform distribution of key values, a processor writes contiguous sets of $\frac{n}{r*p}$ keys in the destination array (where $r$ is the radix used); the $r$ sets a processor writes are themselves separated by $p-1$ other such sets, and a processor's writes to its different sets are temporally interleaved in an unpredictable way. Unless the ratio $\frac{n}{r*p}$ is larger than a page (which requires extremely large values of $n$), writes are fine- to medium-grained while reads are coarse-grained. This induces substantial false-sharing at page granularity.

**Raytrace** renders complex scenes in computer graphics using an optimized ray tracing method. The accesses to the scene data, into which rays are shot in this program, are read only and relatively uninteresting other than the fact that they cause fragmentation (0P-MC). The interesting communication occurs in task stealing using distributed task queues (a migratory sharing pattern), and in updating pixels in the image plane as part of a task. Both types of access patterns are fine grained and cause a lot of false sharing and fragmentation at page level in an induced MP-MC pattern, although in hardware CC-NUMA machines their impact is quite insignificant compared to scene accesses.

**Volrend** renders three-dimensional volume data into an image using a ray casting method. Volrend's sharing patterns are very similar to those of Raytrace, both in task queues and in the image plane. Only, it's working set of read-only scene data is much smaller, and the accesses to scene data are less scattered.

**Water-Nsquared** simulates a system of water molecules in liquid state, using an $O(n^2)$ brute force method with a cutoff radius. The water molecules are allocated contiguously in an array of $n$ molecules, and partitioned among processors into contiguous pieces of $n/p$ molecules each. Again, the most time-consuming phase in a CC-NUMA machine (the force calculation phase with its inherent 1P-MC pattern) is relatively uninteresting since accesses are coarse-grained and there is no synchronization within the phase. The challenging phase for SVM is a small one immediately following force calculation, which exhibits a migratory pattern. In this phase, a processor updates its own $n/p$ molecules and the following $(n/2 - n/p)$ molecules of other processors in the array, using per-partition locks to protect these updates. This leads to a coarse-grained 1P-1C migratory induced pattern, and incurs no false sharing or fragmentation if the array partitions are multiples of and align well to pages.

**Water-Spatial** solves the same problem as Water-Nsquared, but uses a spatial directory rather than a brute-force method, beneficial for large problems. The 3-d physical space is broken up into cells, and every processor is assigned a contiguous cubical partition of cells together with the linked lists of molecules in them. A processor reads data from those processors that own cells on the boundary of its partition. For large enough problems, this turns out to be mostly a 1P-1C inherent pattern with multiple consumers for some cells. For smaller problems it is mostly 1P-MC. The interesting fact is that each water molecule is allocated in a separate `malloc` (hence separate page) thus preventing granularity effects from inducing a different pattern than the inherent one. The infrequent migration of molecules between cells causes locking of linked lists but doesn't make a substantial contribution to performance.

Table 1 summarizes the inherent and induced sharing patterns for the different applications.

## 5 Overall Performance

Table 2 presents the speedups for the ten applications using the basic architectural model with a large one-level cache. The first ten columns show the speedups for each of the CC-NUMA, AURC and LRC schemes for 16-processor executions, with two problem sizes and two sets of bus speeds for AURC and LRC. The last two columns present results for 32 processors for the larger problem size and the fast bus only.

| Application | Small Problem | Large Problem |
|---|---|---|
| Barnes | 8K particles | 16K particles |
| Cholesky | 0.3 M (bcsttk15) | 1.7 M (bcsttk29) |
| FFT | 64K points | 1M points |
| LU | 512-by-512 | 1024-by-1024 |
| Ocean | 128-by-128 | 512-by-512 |
| Radix | 1M keys | 4M keys |
| Raytrace | — | balls4 |
| Volrend | — | $256^3$ head |
| Water-$N^2$ | 343 molecules | 512 molecules |
| Water-Spatial | — | 512 molecules |

Table 3: Problem Sizes Used.

Table 3 shows the problem sizes corresponding to our small/large terminology for each application. Due to simulation resource constraints we simulated only one problem size for Volrend and Raytrace. Also, for Water-Spatial we do not report the results for the small problem size used in Water-Nsquared (343 molecules), since it is too small a problem for that algorithm and causes severe load imbalance.

We see that the hardware-coherent speedups in Table 2 are excellent even for the small problem sizes on almost all applications. It is also clear that the performance of both SVM schemes is more variable across applications than that of CC-NUMA. For the smaller problems, speedups are quite poor except for the regular applications that have no adverse induced communication effect, like LU or Water-Nsquared. Barnes and Ocean also deliver reasonable performance for AURC with the fast bus combination. As we expected, the fast memory bus tends to improve performance for the SVM schemes, more so for AURC in which the automatic updates stress bus performance more than in LRC.

For the applications that perform poorly for the small problems under SVM, the natural question is whether this is due to the problem sizes being too small, or whether the applications are inherently not well-suited to SVM. To explore the questions, we also look at the results for large problem sizes. Using large problems helps many of the applications tremendously. For example, FFT gets a huge boost, and the performance of Water-Nsquared, Water-Spatial, and LU, as well as Barnes and Ocean for AURC, get closer to the performance of hardware coherence on 16 processors. Cholesky achieves somewhat reasonable performance for the first time with the large problem in the fast bus case. Radix improves but continues to perform poorly.

Now that we have problem sizes for which many applications perform quite well for 16 processors, let us look at what happens when we move to 32 processor systems for these problem sizes (the last two columns of Table 2). In some cases such as Raytrace, Cholesky and Radix, the speedup doesn't improve much or even drops, although hardware-coherent machines continue to scale very well to 32 processors and beyond. In other cases, the speedups continue to increase, although the parallel efficiency (speedup divided by number of processors) drops substantially. The speedup increase for LU, Water-spatial and Ocean is substantial, while for others it is significant but much smaller (FFT, Barnes, Water-nsquared and Volrend).

It therefore appears that several of the applications have the potential to run well even on 32-processor SVMs with larger problem sizes.[1] To study this further, and to examine if the loss in parallel efficiency relative to hardware coherence in going from 16 to 32 processors can also be overcome with some increase in problem size, we took a few applications with different performance behaviors and ran them on 32 processors with even larger problems (see Table 4). For some (Water-Spatial, FFT, Cholesky), the speedup improved substantially. However, Radix does not seem to be improving, and some irregular applications like Barnes improve very slowly with problem size. Ocean too improves only a little. The detailed discussion of application performance in the next section will explain these effects. So far, it seems that for several applications we can indeed get quite good performance from SVM systems (particularly AURC), but usually by using significantly larger problems than needed to get good performance from hardware cache-coherent systems. Other applications seem to be more difficult.

As for a comparison of AURC and LRC at this level, although the use of a very large one-level cache biases the

[1] Note that even the problems we call "large", while realistic, are not anywhere near the high end of the sizes people would run in practice. We have simply taken problem sizes that perform well on hardware cache-coherent machines of this scale—our "small" problems—and increased them somewhat to see the effect of running larger problems.

| Application | Small Problem (P=16) | | | | | Large Problem (P=16) | | | | | Large Problem (P=32) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Fast Bus* | | | *Slow Bus* | | *Fast Bus* | | | *Slow Bus* | | | |
| | CC-NUMA | AURC | LRC | AURC | LRC | CC-NUMA | AURC | LRC | AURC | LRC | AURC | LRC |
| Barnes | 13 | 9.1 | 6.2 | 6.7 | 5.7 | 14.2 | 10.3 | 6.4 | 8 | 6 | 13.4 | 8.2 |
| Cholesky | 14.8 | 4.5 | 2.9 | 2.7 | 2.4 | 15.1 | 6 | 4.4 | 3.7 | 3.8 | 4.3 | 3.5 |
| FFT | 14.3 | 4.6 | 4.6 | 3 | 2.9 | 14.8 | 11 | 10.2 | 8.3 | 7.6 | 14 | 11 |
| LU | 10.4 | 10.3 | 9.3 | 10.2 | 9 | 14.3 | 13.3 | 11.8 | 13.2 | 11.7 | 22 | 19 |
| Ocean | 5.6 | 5.2 | 3.1 | 3.4 | 2.2 | 13.2 | 11.4 | 5.3 | 8.2 | 4.1 | 17 | 8.7 |
| Radix | 20 | 1.4 | 1 | 1 | 0.8 | 16 | 2.9 | 2.7 | 2.4 | 2.1 | 3.6 | 2.8 |
| Raytrace | — | — | — | — | — | 14.7 | 6.6 | 7 | 6 | 6 | 0.37 | 0.28 |
| Volrend | — | — | — | — | — | 14.1 | 7.1 | 7.1 | 5.3 | 5.5 | 9.5 | 9 |
| Water-$N^2$ | 11.6 | 8.9 | 7.6 | 7.8 | 6.8 | 14.4 | 11.5 | 10.2 | 10.2 | 9 | 14 | 11 |
| Water-Spatial | — | — | — | — | — | 15.1 | 13 | 12 | 11.6 | 10.1 | 19 | 16 |

Table 2: Speedup Table.

| Application | Larger Problem (P=32) | | |
|---|---|---|---|
| | Size | Fast Bus | |
| | | AURC | LRC |
| Barnes | 32K particules | 19 | 11 |
| Cholesky | 4 M (bcsttk31) | 10 | 10 |
| FFT | 4M points | 20 | 19 |
| Water-Spatial | 4096 molecules | 26 | 24 |

Table 4: Run with Larger Problems

results toward LRC (by not letting its diffs pollute the cache much), the hardware-supported AURC seems to perform better than LRC (Ocean is a dramatic example). AURC is also helped more by the use of faster buses, improves more quickly as larger problems are used, and preserves its speedups better to larger processor counts. The difference among the schemes of course varies across applications. In Section 7, we shall see that the impact of using a more realistic two-level cache hierarchy can be substantial, hurting speedups significantly for LRC but not affecting (in fact, usually improving a little) the speedups for AURC.

Finally, to capture the effect of realistic situations in which the working set of an application does not fit in the cache, we ran Ocean and FFT with smaller cache sizes as recommended in [17] (we did not run the other two recommended applications, Radix and Raytrace, this way, the former because it performs so poorly and the latter because of simulation time). The speedups for FFT were not much different than with the large cache, while those for Ocean exhibited a superlinear effect due to the partitions fitting in the cache as more processors were used.

In addition to the above performance summary, an important point to take away from the high-level speedup comparisons is that the performance of SVM in general, and even the relative performance of the two SVM schemes, varies much more widely across the applications than the performance of the hardware-coherent machine. It is also much more sensitive to issues like problem and machine size. Studies that evaluate SVM with a small set of applications and without looking at the effects of problem size can therefore be misleading. To truly understand the high-level results described above, the potential for SVM systems, and the value of the hardware support in AURC for different applications, it is very important to interpret and explain the results in light of individual application characteristics. We do this in the next section.

## 6 Performance Analysis

In this section, we analyze the performance of the two SVM schemes in the context of the application classification described earlier, identify the key performance bottlenecks, and explain the high-level results and trends we observed in the previous section. We divide the applications into four classes for this discussion. All the figures we use show the components of execution time in the SVM cases normalized to the execution time of the hardware CC-NUMA system. In all figures **H** stands for hardware CC-NUMA, **A** for AURC and **L** for LRC. The legend for the components is the same for all graphs, and is shown only in Figure 4. (*Computation* is the time spent executing useful instructions; *Communication*, *Synchronization* and *Stalling* are the times spent waiting at page faults, synchronization operations and local accesses, respectively; and *Protocol* is the time spent executing software protocol handlers, including their memory accesses.)

### 6.1 Applications without Adverse Granularity Effects

Three of our applications are not affected by the large granularity of sharing in SVM, and retain their inherent sharing patters. These are LU (Figure 2), Water-Nsquared (Figure 3), and Water-Spatial (Figure 4). In all three, the performance for both SVM protocols is fairly close to those of CC-NUMA, with AURC being at most 20% worse than

CC-NUMA for 16 processors.

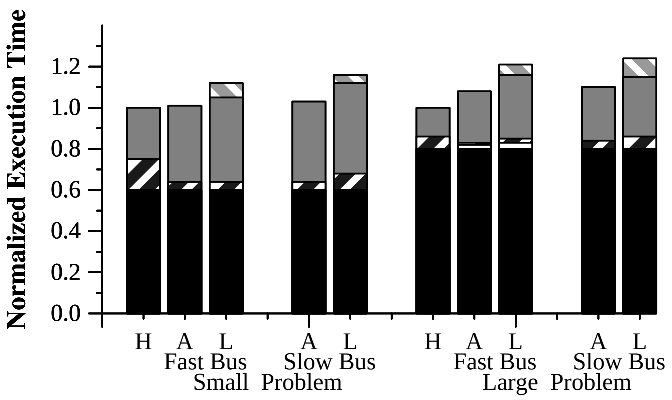


Figure 2: Results for LU.

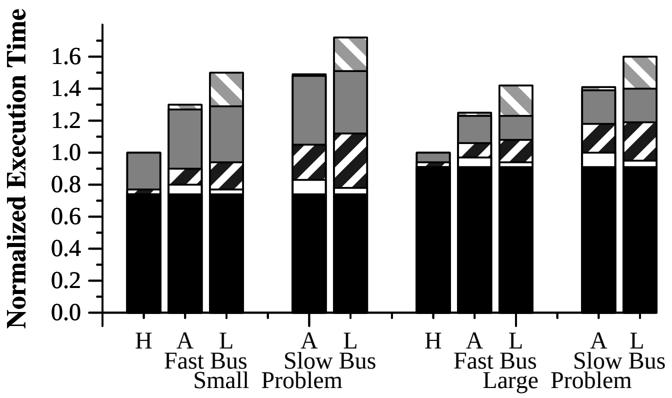


Figure 3: Results for Water-Nsquared.

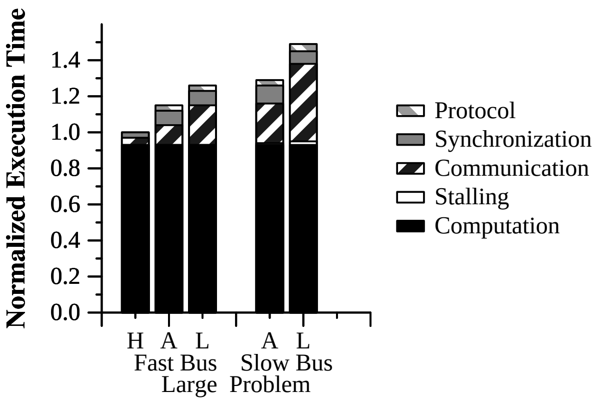


Figure 4: Results for Water-Spatial.

For LU, AURC delivers the same performance as CC-NUMA for 16 processors. This is because the use of pages provides a beneficial prefetching effect that compensates for the software protocol overheads. The main performance bottleneck in LU is load imbalance. While this persists for all systems, LRC is slightly more affected because the computation of diffs increases the imbalance. For 32 processors the speedup for the SVM systems is still good although the efficiency drops due to worse load imbalance.

In Water-Nsquared, the remote accesses are regular and to contiguous data, which makes communication at page grain very efficient. The inherent communication to computation ratio is low. The frequency of lock-based synchronization in the interesting force accumulation phase is kept low by using temporary data structures as partial sums, but the remaining locks protect large critical sections and are quite highly contended. Locks are generally much more expensive in SVM systems than in hardware CC-NUMA, which hurts the performance of many of our next few applications: Software protocol activity occurs at the synchronization points, and the communication that occurs within the critical sections is more expensive, resulting in a large dilation of the time spent in the critical section and hence increased contention for the lock. This exaggerated contention for lock-based synchronization accounts for most of the difference from hardware CC-NUMA (which has a negligible synchronization component). The synchronization component increases at higher processor counts, limiting SVM speedup even more, but it reduces relative to computation when the problem size is increased. In Water-Nsquared, diffs are large because the migratory data are updated at medium to coarse granularity, so this is in fact a case where the communication cost actually ends up higher in LRC. While bus contention due to write-through has a visible effect for AURC, especially with the slow bus, the effect is negligible compared to other costs.

Although Water-Spatial makes accesses at the granularity of molecules, as discussed earlier it behaves like a coarse-grained program so it also performs well at 16 processors. The speedup is quite good even at 32 processors, but the parallel efficiency drops mainly because the contention for locks increases. Increasing the problem size much further helps performance greatly (see Table 4).

For both LU and Water-Spatial, the differences between AURC and LRC are quite small. This is because communication to computation ratio is low, and the main bottleneck is either load balance (LU) or contention for synchronization, which affects both protocols similarly. Also, in both cases the induced 1P-MC pattern allows LRC to amortize the cost of computing the diffs over multiple consumer requests.

### 6.2 Regular Applications with Induced Fragmentation

As we know, both FFT (Figure 5) and Ocean (Figure 6) fall in this category since they have fine- or medium-grained remote read accesses. Thus, the efficiency of communication is low in both cases if performed at page granularity.

In FFT, the amount of fragmentation depends on the problem size, so increasing the problem size improves performance dramatically. The high communication volume in FFT causes the bus speed to have a large impact too. Fragmentation increases when using more processors, so the speedup at 32 processors is not much higher than at 16. However, continuing to increase the problem size (Table 4) improves the speedups greatly even at 32 processors as expected. A large enough problem would eliminate the fragmentation, at which point speedup should be comparable with CC-NUMA. Comparing AURC with LRC, we see similar performance. While there is additional protocol overhead due to diff computation in LRC, bandwidth needs are high and this overhead is compensated by larger memory bus contention due to write-through traffic in AURC.

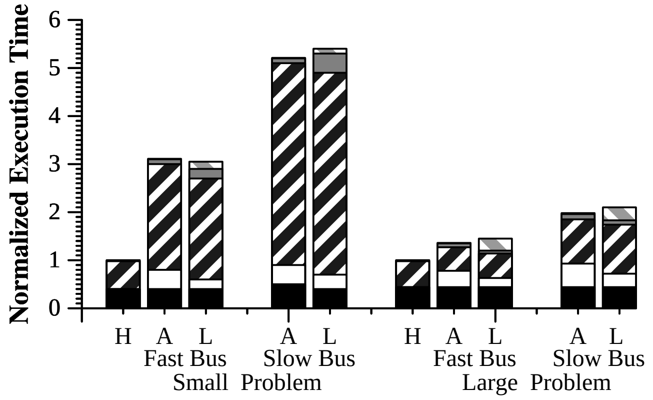


Figure 5: Results for FFT.

Ocean should have similar behavior, with the difference that the fragmentation doesn't disappear by increasing the problem size. This is exactly what we observe for LRC (although communication to computation ratio decreases substantially). However, Ocean turns out to be a performance showcase for AURC. This is because the 1P-1C pattern allows the specialized AURC protocol to kick in (see Section 2), and thus not incur any page faults at all.

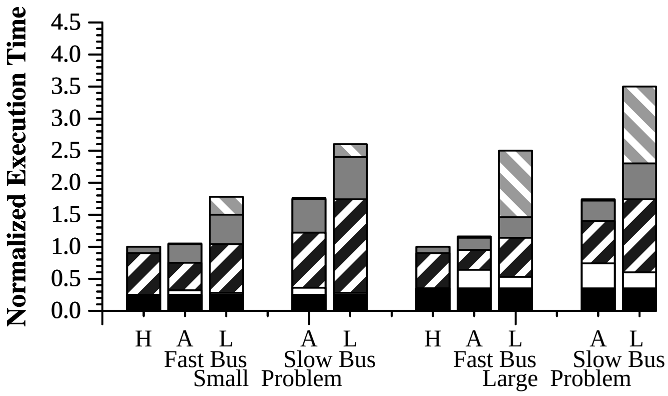


Figure 6: Results for Ocean.

This specialized protocol also allows AURC to perform dramatically better than LRC, and causes performance to scale well even to 32 processors. The problem for LRC in Ocean is that although only the boundary elements need to be communicated, the diffs constitute the whole page that they fall on since the whole page (not just the boundary element) has indeed been written by the producing processor. Thus, not only does the whole page have to be diff-ed unnecessarily, it ends up being communicated as well. The extra bus bandwidth needs of AURC are substantial for Ocean, but are overshadowed by these costs in LRC.

### 6.3 Irregular Applications with False Sharing and Fragmentation

Barnes, Cholesky, Raytrace and Volrend (Figures 7 to 10) are all irregular applications that have the undesirable MP-MC sharing pattern induced by their fine-grained read and write accesses. All therefore suffer substantially from false sharing as well as fragmentation. Synchronization overhead

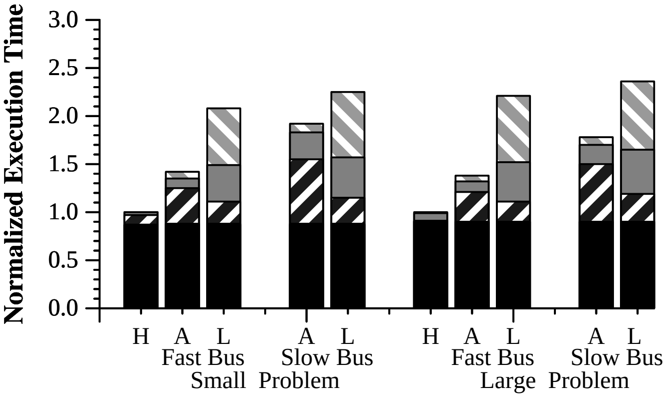


Figure 7: Results for Barnes.

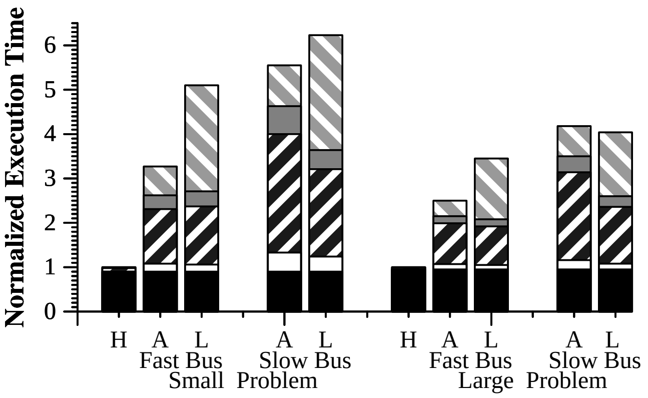


Figure 8: Results for Cholesky.

is also high, because most of the synchronization is through a large number of locks. The dilating effect that SVM systems have on critical sections, discussed earlier, causes the cost and contention for these locks to be very high, even though the same locks incur very low overhead in hardware CC-NUMA. Imbalances in communication, contention and synchronization cost also amplify the overall load imbalance in these applications for SVM systems. The result is that performance can be quite limited for these irregular applications.

Speedups can be increased in some cases by increasing the problem size, but not as effectively as for regular applications. This is because in the irregular applications the accesses do not tend to get less scattered when the problem size increases [17, 10], and no thresholds are crossed that reduce fragmentation or false sharing. The inherent communication to computation ratio does not change so quickly either. Given the MP-MC patterns, let us see how the tradeoffs between the multiple-write mechanisms of LRC and AURC play out in the different applications in this category.

In Barnes, the tree building phase exhibits the big difference between CC-NUMA and the SVM protocols. Unlike in hardware CC-NUMA where it takes a small fraction of the execution time, it becomes an expensive phase under software shared memory (this characteristic holds for many of the irregular applications). The tree building involves a lot of very fine-grained communication, protected by locks, and produces large communication and protocol overhead for very little computation. The synchronization wait time in Barnes

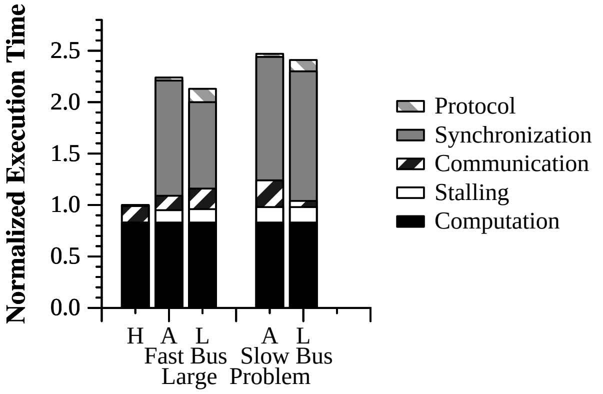


Figure 9: Results for Raytrace.

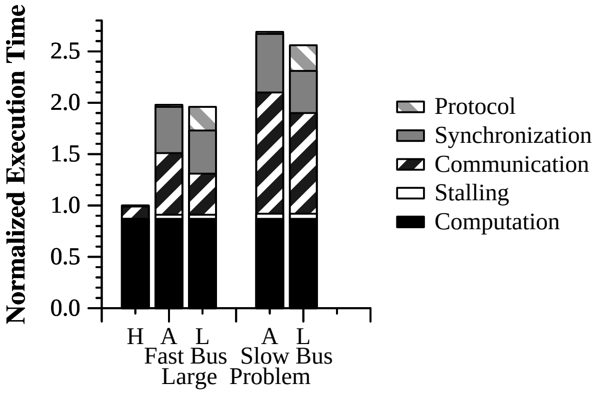


Figure 10: Results for Volrend.

is large mainly because of imbalances in communication and contention (not in computation) caused by the accesses to the tree. This too is a much larger effect than in hardware CC-NUMA. Increasing the number of processors doesn't increase the speedup much because the relative frequency and cost of the expensive synchronization increases. Larger problem sizes help, but only slowly.

In Barnes, AURC transfers twice as much data as LRC due to the effectiveness of diffs for fine-grained sharing. However, the diff computation is much more expensive than the savings in bandwidth. Hence, Barnes performs much better with AURC particularly due to the tree-build phase.

In Cholesky, because accesses are largely medium-grained (half a page) rather than fine-grained, diffs are less effective in reducing traffic. The data communicated is about the same for AURC and LRC. On the other hand, the cost of diff computation is amortized better in Cholesky since the inherent pattern for the blocks is 1P-MC (similar with LU). This is why the performance gap between AURC and LRC is significant only for the fast bus case. Overall, performance is quite poor for Cholesky compared to CC-NUMA for small problems, and even degrades at 32 processors. Performance is made reasonable at 32 processors only by using much larger problems (Table 4), but is still much worse than that of hardware CC-NUMA.

For the graphics applications (Volrend and Raytrace), SVM performance is modest for two reasons: write-sharing is very fine-grained, and there is intensive lock synchronization separating these writes. The performance of the two SVM protocols is similar. In Volrend, it is the standard tradeoff between protocol overhead in LRC and extra communication in AURC. In Raytrace, synchronization and contention dominate for both systems. This is why while in Volrend the speedup increases from about 7 to about 9 in going from 16 to 32 processors, in Raytrace there is in fact an absolute slowdown at 32 processors. We were unable to run larger problems for these applications due to simulation constraints.

### 6.4 Applications Dominated by False Sharing

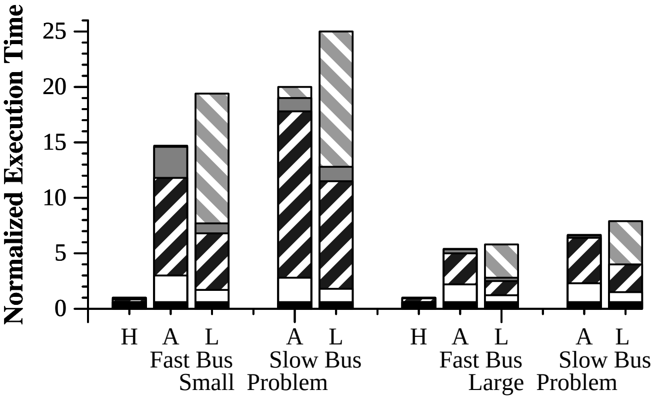


Figure 11: Results for Radix.

The Radix program (Figure 11) is interesting in that its induced pattern appears to be MP-1C at first blush (different processors write keys to portions of the array that only one processor will read). However, this pattern is not really preserved across synchronizations in an SVM system even under release consistency, as long as the producers reference the data again sometime later (as they do in Radix to write the data again in the next iteration). The next time they try to write the data after a synchronization (e.g. in the next iteration), they become involuntary consumers (hence MP-MC) since they have to fetch the updates from the other writers even though they don't need them. This false sharing, together with an inherently large communication to computation ratio and a lot of uneven contention due to page granularity (see the synchronization component0, hurts performance dramatically in Radix leading to almost no speedup.

We might think that AURC should perform much better than LRC, since it supports fine-grained remote updates and automatically propagates the writes to the single consumer site. However, the above involuntary consumer effect comes into play here as well, causing a lot of page faults. Using larger problem sizes doesn't help much in Radix, since it only tends to increase contention due to larger traffic volume (the inherent communication to computation ratio does not improve with problem size in Radix). Performance is only likely to improve when the problem is large enough that the number of keys per radix value per processor ($\frac{n}{r*p}$) is much larger than a page, so the writes by a processor can be considered contiguous at page granularity and there is no false-sharing. However, the bandwidth requirements and contention are high enough in Radix that we do not expect to see good performance from SVM.

One solution for the AURC protocol in a Radix-like involuntary consumer situation is to support *write-only* pages: A writer of such a page does not need to see previous updates, and does not need to be invalidated after a synchronization. This is perfect for Radix: AURC would not incur any page faults in the permutation phase, and would potentially perform quite well given enough bandwidth. But this requires a further relaxation in the consistency model. In fact, FFT can also benefit from this protocol extension, if the transpose is changed from being read-remote, write local to read-local, write-remote.

## 7 A More Realistic Cache Hierarchy

All the performance results presented so far are obtained from simulating a large one level cache. As discussed in Section 3, this is likely to bias the results in favor of LRC. To study the behavior of the SVM protocols under more realistic assumptions, we also simulated a two level cache hierarchy with a 16K direct mapped first level cache (L1) and a 1MB two-way set associative second level cache (L2) with a 10 cycle hit cost. We perform all these experiments with the fast bus setup and the large problems from Table 2. Table 5 presents the speedups for both cache hierarchies. We can see that LRC speedups are reduced quite substantially (there are no diffs in the uniprocessor case), while the AURC speedups either don't change or increase slightly (the damage to uniprocessor performance is greater than to the multiprocessor case). AURC now outperforms LRC substantially in all cases. In four out of 10 cases (Barnes, Choleksy, Ocean, Raytrace) the speedup with AURC is double that with LRC.

| Application | Large Problem (P=16) *Fast Bus* | | | |
|---|---|---|---|---|
| | L1 Cache | | L1 & L2 Caches | |
| | AURC | LRC | AURC | LRC |
| Barnes | 10.3 | 6.4 | 10.3 | 4.1 |
| Cholesky | 6 | 4.4 | 7.3 | 3.2 |
| FFT | 11 | 10.2 | 11 | 9.8 |
| LU | 13.3 | 11.8 | 13.5 | 11.3 |
| Ocean | 11.4 | 5.3 | 12.9 | 5.4 |
| Radix | 2.9 | 2.7 | 3.4 | 2.7 |
| Raytrace | 6.6 | 7 | 8.7 | 4.4 |
| Volrend | 7.1 | 7.1 | 7.4 | 5.5 |
| Water-$N^2$ | 11.5 | 10.2 | 11.6 | 8.6 |
| Water-Spatial | 13 | 12 | 13.5 | 12 |

Table 5: Effect of Two-Level Caches.

## 8 Sensitivity and Robustness

Finally, before we conclude we briefly examine some interesting variations in the architecture and programs that can affect our results.

**Write Buffer Size** The write buffer in our simulations so far was 8 cache lines deep. Since deep cache buffers favor a write-through scheme like AURC, we also examine some of the applications that have a significant write stall time (Barnes, Cholesky, FFT and Ocean) using a 4 cache line deep write buffer both under LRC and AURC. We do not show the results for space reasons, but we find that while the write buffer size does affect the number of processor stalls which become 2-3x larger, the impact on total elapsed time is negligible, ranging from 2-4%.

**Contiguous vs. Non-Contiguous Block Allocation** Two of the applications, Ocean and LU, have less optimized versions provided for them in the Splash-2 suite. These "non-contiguous" versions use a more natural 2-d array representation of the 2-d grids and matrices, which does not allow us the benefit of contiguous access patterns or effective data placement across main memories that we have seen so far, and results in substantial false sharing. These programs thus provide a good test of one form of performance robustness of the systems. The results are catastrophic for both schemes due to false sharing, particularly for the LU program. In AURC, the poor placement of homes also has a deleterious effect, particularly for LU where the heuristics used by default in AURC (which assigns the page to the first process that touches it, using a heuristic to avoid initialization effects) behaves poorly and creates hot spots for the automatic updates. Neither scheme is very robust to the lack of the data structuring optimization, but AURC is less robust than LRC.

**Data Placement** LRC doesn't benefit from the placement of pages in home memories, since updates are collected in a distributed fashion everywhere. In AURC good home placement may save page faults, and may also reduce data traffic if a page is assigned to a heavy writer. This is why programmers' hints about page placement may help, particularly when they can be conveyed implicitly. Because home placement with our heuristic was a problem for the non-contiguous version of LU, we experimented with round-robin home allocation in this case. This worked much better since it avoided the hot spots, but the communication traffic due to false sharing remained high. To test the importance of data placement further, we also tried round-robin home assignment for the FFT program. Since the writes now became nonlocal, the automatic update communication traffic in AURC became huge and halved the speedup from 11 to 5. LRC was not affected much. A sound home placement is therefore very useful for AURC, in reducing communication and contention, but since it is also important for hardware CC-NUMA in the same situations shared address space programs often have it already.

## 9 Conclusions

There are several lessons we learned from this study. Overall, we found SVM to perform surprisingly well for several class of applications, both regular and irregular, even up to 32-processor systems. However, SVM clearly needs significantly larger problem sizes to obtain good parallel efficiency than hardware CC-NUMA. Besides improving communication to computation ratio, in regular applications large problems often overcome thresholds in relationship with the coherence and communication granularity. The problem sizes we have used are by no means large. The performance potential of SVM may increase for large practical problems.

Because of its high software protocol overheads, SVM is extremely sensitive to how data structures and access patterns interact with the page granularity. The sharing patterns induced by this granularity can be very different than the inherent sharing patterns in the application, which can have a substantial impact on performance. However, certain techniques like contiguous allocation of related data (as in Ocean) or separate memory allocations for logical objects (as in Water-Spatial) can be used to preserve simple sharing patterns like 1P-MC which are very effective. We also found lock-based synchronization to be much more expensive and contentious in SVM than in CC-NUMA, particularly in the irregular applications, since the software protocol operations at synchronization points greatly dilate the critical sections. In general, contention is a much more substantial issue, and induces high communication costs and load imbalances in many applications.

Our results suggest that the simple hardware support needed for AURC is effective, and enables good performance for SVM for a much larger class of applications. For certain regular applications, AURC performs quite comparably with CC-NUMA for these small systems. Particularly with a realistic two-level cache hierarchy, AURC outperforms LRC significantly in most cases. The basic tradeoff between the two schemes is that AURC potentially consumes more bandwidth due to its write-through nature for shared data, while LRC has much greater end-point communication overhead due to diff processing. Our results suggest that the bandwidth needs are likely to be met more easily in modern systems, and will be a less significant bottleneck than the diff processing overhead.

The detailed understanding of performance issues and interactions that we have gained from this study has led us to several interesting directions for SVM systems research that we are now beginning to explore. For example, further relaxing the consistency model in a transparent manner can help reduce false sharing in many situations [11]. It is also interesting to examine how to modify programs to perform better on SVM systems [10].

## Acknowledgments

We are grateful to the referees, whose comments helped improve the paper. Chris Holt helped us with hardware CC-NUMA simulations. We also thank Cezary Dubnicki and Stefanos Damianakis for their help, and Jim Philbin, Henry Cejtin, Ed Rothberg and Andrew Erlichson for generously providing us with simulation cycles. This work is sponsored in part by ARPA under contract under grant N00014-95-1-1144, by NSF under grant MIP-9420653, by Digital Equipment Corporation and by Intel Corporation.

## References

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementation. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[2] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 125–134, May 1990.

[3] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON '93 Conference*, February 1993.

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[6] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 106–117, April 1994.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[8] C. Holt, M. Heinrich, J.P. Singh, E. Rothberg, and J. Hennessy. The Effects of Latency, Occupancy and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Stanford, January 1995.

[9] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[10] L. Iftode, J.P. Singh, and K. Li. Irregular Applications under Software Shared Memory. Technical Report TR-514-96, Princeton, NJ, February 1996.

[11] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: a Bridge between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996. To appear.

[12] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[13] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[14] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve t he Performance of Distributed Shared Memory. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[15] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[16] W. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *The Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[17] Steven Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, June 1995.