Limits to the Performance of Software Shared Memory: A Layered Approach

Angelos Bilas[1], Dongming Jiang[2], Yuanyuan Zhou[2], and Jaswinder Pal Singh[2]

[1]Department of Electrical and Computer Engineering
10 King's College Road
University of Toronto
Toronto, ON M5S 3G4, Canada
bilas@eecg.toronto.edu

[2]Department of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08544, USA
{dj, yzhou, jps}@cs.princeton.edu

Abstract

Much research has been done in fast communication on clusters and in protocols for supporting software shared memory across them. However, the end performance of applications that were written for the more proven hardware–coherent shared memory is still not very good on these systems. Three major layers of software (and hardware) stand between the end user and parallel performance, each with its own functionality and performance characteristics. They include the communication layer, the software protocol layer that supports the programming model, and the application layer. These layers provide a useful framework to identify the key remaining limitations and bottlenecks in software shared memory systems, as well as the areas where optimization efforts might yield the greatest performance improvements. This paper performs such an integrated study, using this layered framework, for two types of software distributed shared memory systems: page-based shared virtual memory (*SVM*) and fine-grained software systems (*FG*).

For the two system layers (communication and protocol), we focus on the performance costs of basic operations in the layers rather than on their functionalities. This is possible because their functionalities are now fairly mature. The less mature applications layer is treated through application restructuring. We examine the layers individually and in combination, understanding their implications for the two types of protocols and exposing the synergies among layers.

1 Introduction

As clusters of workstations, PCs or symmetric multiprocessors (SMPs) become important platforms for parallel computing, there is increasing interest in supporting the attractive, shared address space (SAS) programming model across them in software. The traditional reason is that it may provide successful low–cost alternatives to tightly–coupled, hardware–coherent distributed shared memory (DSM) machines. A more important reason, however, is that clusters and hardware DSMs are emerging as the two major types of platforms available to users of multiprocessing. Users would like to write parallel programs once and run them efficiently on both types of platforms, and programming models that do not allow this may be at a disadvantage. Thus, despite (and in fact, because of) the success of hardware–coherent DSM, software shared memory on clusters remains an important topic of research.

Supporting a programming model gives rise to a layered communication architecture that is shown in Figure 1. The lowest layer is the *communication layer,* which consists of the communication hardware and the low level communication library that provide basic messaging facilities. Next is the *protocol* layer that provides the programming model to the parallel application programmer. We assume all–software DSM protocols in this paper, and focus on two well-studied approaches: page–based shared virtual memory (*SVM*) and fine–grained DSM (*FG*). Finally, above the programming model or protocol layer runs the *application* itself.

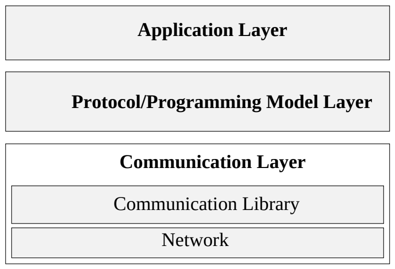

Figure 1: The layers that affect the end application performance in software shared memory.

The last decade has seen a lot of excellent research in the individual layers, especially the lower two system layers [4, 3, 5, 14, 12, 13, 21, 7, 18]. Still, software shared memory systems currently yield performance that is, for several classes of applications, far behind that of hardware–coherent systems even at quite small scale.

This paper uses a layered framework to examine where the major gains (or losses) in the parallel performance of applications can or cannot come from in the future, both in the individual layers and through combinations of them, and to help cluster software and hardware designers determine where best to spend their energy for the goal of supporting software shared memory.

In the application layer, the main variable is how an application is structured or orchestrated for parallelism. The two system layers (communication and protocol) however, have both functionality and performance characteristics, all of which contribute to end application performance. It is this space that we need to navigate. This paper treats the functionality of the system layers as fixed (since it is now quite mature), and varies only their basic performance costs. Results obtained by varying costs can suggest modifications in functionality that can reduce or avoid the critical costs. The less mature, application layer is treated by examining the impact of application restructuring, as performed for

SVM in [11], starting from programs that are optimized for moderate-scale hardware coherent systems. We examine the impact of the layers individually and in combination, and also isolate the impact of different costs in each layer. Solving problems or reducing costs at each layer has its own advantages, disadvantages, and potential for being realizable, and is under the control of different forces. Knowledge of actual trends in the layers (application or system) can be used to draw implications for the future of different approaches.

While some individual aspects of this work have been studied in the past (see Section 4), the contributions of this paper are: (i) the layered approach in which it investigates performance issues, which we believe is generally useful, (ii) studying the effects of each layer for both *FG* and *SVM*, (iii) studying the synergies among layers, and (iv) providing detailed results and analysis for a wide range of applications.

Our highest level conclusions are: (i) With currently achievable system parameters the *FG* and *SVM* approaches are competitive in performance, at least at the scale we examine, with the tradeoffs depending on particular application characteristics. (ii) For *FG*, other than access control costs the most important layer to improve is the communication layer (especially overhead and occupancy), and the impact of access control costs are relatively independent. (iii) *SVM* exhibits a much richer story: All three layers are important and exhibit a lot of synergy, and no one or even two of them will suffice for a wide range of applications; in general, the order of importance is application, communication, protocol. While the set of demands is broader, this may be an advantage since there is less dependence on particular aspects of one layer that may be very difficult to control. (iv) Overall, if only one system layer can be improved, it should be the communication layer, though different aspects of it matter for different protocols.

Section 2 describes the protocols and the methodology we use. Section 3 presents our main results for each layer individually as well as for the synergy among them. Section 4 briefly discusses some closely related work, and Section 5 provides a discussion and concludes the paper.

2 Methodology

Although real *SVM* systems are available, a study that varies (and especially reduces) costs in the manner done here can only be done through simulation. We therefore use a detailed execution–driven simulator which we validate carefully against real systems as discussed later. While computational clusters are moving toward using SMP rather than uniprocessor nodes, we perform this study assuming uniprocessor nodes, primarily because protocols for SMP nodes are not so mature and there are many more interactions in the nodes that can affect performance in subtle ways.

Protocols: Since we use well known, state-of-the-art protocols, we describe them only briefly. Page–based coherence protocols use the virtual memory mechanism of microprocessors for access control at page granularity [14]. Fine–grained access control can be provided by either hardware support or by code instrumentation in software [7]. In both cases, we assume the coherence protocol runs in software handlers rather than in hardware, and on the main processor rather than on co–processors.

For *SVM*, we use the Home–based Lazy Release Consistency (HLRC) protocol [21], which implements the lazy release consistency (LRC) model [12] to reduce the impact of false sharing. Both HLRC and older LRC protocols use software *twinning* and *diffing* to solve the multiple–writer problem, but with different schemes for propagating and merging diffs (updated data). Traditional LRC schemes maintain distributed diffs at the writers, from where they must be fetched on a page fault [12]. In HLRC, the writer sends the diffs eagerly to a designated home node for the page, and the diffs are applied there to the home copy which is always kept up to date according to the consistency model. On a page fault, instead of fetching diffs from previous writers, the whole page is fetched from the home.

Fine–grained protocols are able to reduce the occurrence of false sharing by virtue of their fine granularity of coherence, so they do not rely heavily on relaxed consistency models. However, they require support for fine–grained access control. Our fine–grained implementation (*FG*) uses sequential consistency, is based on the Stache protocol [16], and is similar in protocol structure to many directory–based hardware implementations [1]. Access control is assumed to be provided at any fixed power of two granularity for a given application. The fact that we use the best–performing granularity for each application in the *FG* case requires some programmer intervention. The granularities used are 64 bytes in all other cases than the regular applications: FFT (4 KBytes), LU (4 KBytes) and Ocean (1 KByte).

Simulation environment: Our simulation environment is built on top of `augmint` [19], an execution driven simulator using the *x86* instruction set, and runs on *x86* systems. It models a cluster of 16 uniprocessor nodes connected with a commodity Myrinet-like interconnect [4] (Figure 2). Contention is modeled in great detail at all levels, including the network end–points, except in the network links and switches themselves. Thus, when we change protocol or communication layer costs, the impact on contention is included as well. The processor has a P6–like instruction set, and is assumed to be a 1 IPC processor. Details about the simulator can be found in [1].

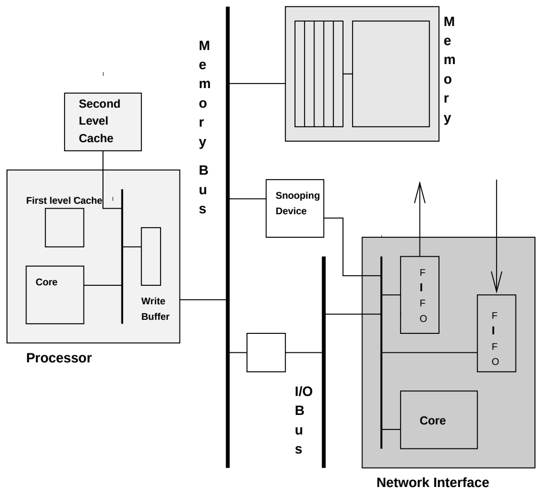

Figure 2: Simulated node architecture.

The fine-grained access control needed for *FG* can be provided via either code instrumentation [7, 18] or hard-

[1]Protocols using more complex, delayed consistency or single-writer eager release consistency were found to perform only a little better in [22].

ware support [17]. Code instrumentation is also used for polling to handle asynchronous incoming messages, which would otherwise cause expensive and frequent interrupts (interrupts are much less frequent in the coarser-grained *SVM*, so the tradeoffs between interrupts and polling are less clear there). Since we do not have access to high-performance instrumentation for the x86 instruction set, and since it is unclear what cost to ascribe to hardware access control which can be implemented in various ways, we assume access control and polling are free in the *FG* protocol (access control is already built in to the processor for *SVM*). In fact, due to our lack of instrumentation, the simulator uses an interrupt-like mechanism for asynchronous message handling, but charges costs based on those of invoking a handler in polling. Actual instrumentation costs can likely be simply added to execution time to first order, since they are just additional compute instructions. We quote some instrumentation overhead data from the Shasta system, which are probably optimistic for x86 machines and hence are provided only to give a very rough idea. Directly, the results we obtain can only be used to compare *SVM* with software *FG* assuming very efficient hardware access control. The cost of each protocol handler is computed according to the protocol task it performs.

The simulator has been validated against real system implementations for both *FG* (by setting parameters close to those of the Typhoon–zero system [17] and comparing with it) and *SVM* for our real cluster [1]. The results, omitted for space reasons, are surprisingly accurate.

Applications: Table 1 shows the applications and the problem sizes we use in this work. These applications are written for hardware DSM and they are known to deliver excellent parallel speedups for hardware cache coherent systems at the 16–processor scale we assume in this paper. They are taken from the SPLASH–2 suite (and from the restructured versions in [11]), and will not be described further here.

Application	Problem Size	Instrum. cost
Barnes	8K particles	8%
FFT	1M points	29%
LU	512x512 matrix	29%
Ocean	514x514 grid	12%
Radix	1M keys	33%
Raytrace	car	29%
Volrend	256x256x256 CT head	5%
Water-Nsquared	512 mols	14%
Water-Spatial	512 mols	18%

Table 1: Applications, problem sizes and instrumentation costs. The last column is the instrumentation cost in the aggressive, Alpha-based Shasta system, which we quote from [18] .

Protocol layer parameters: The protocol costs we vary and their achievable values that are assumed for the base system are shown in Table 2. The page protection cost is the cost of `mprotect`. A single `mprotect` call may be made to the kernel for a range of contiguous pages; it incurs a startup processing overhead, that is simulated, plus this per–page cost. This cost is aggressive for current operating systems but achievable with existing technology (we verified that using a somewhat larger cost does not change the results much). The effects of twinning and diffing on the cache (and the misses incurred) are simulated. Protocol handlers themselves cost a variable number of cycles. The basic handler cost parameter is set to 100 cycles for both *SVM* and *FG*; additional costs for diffing or traversing lists (e.g. write notice lists) are added to this cost. The cost for traversing lists depends on processor speed and is set to 20 cycles per list element. The cost for diffing is varied to allow for studying the importance of diffing in *SVM*. All these costs closely approximate those of our real implementation. The set of values we consider for the protocol layer costs are: the *achievable* set, the *best* (idealized) set, where all costs are set to zero, and the *halfway* set, where all costs are halfway between achievable and best. We do not examine a worse set of costs than the achievable set since these costs are quite closely linked to processor speed.

Parameter	Achievable	Best	Halfway	Units
Page Protection	50	0	25	cycles/page
Diff creation	10,10	0,0	5,5	cycles/word
Diff Application	10	0	5	cycles/word
Twin Creation	10000	0	5000	cycles/word
Handler Cost	100+x	0+x	50+x	cycles

Table 2: Protocol layer parameter values. The diff creation cost incurs a cost per word that is compared (first number) and an additional cost per word that is put in the diff (second number). The handler cost is composed of a basic cost and any additional cost for operations that may be executed by the handler.

Communication layer parameters: The parameters of the communication layer that are of most interest are the following *Host overhead* is the time the host processor itself is busy sending a message, i.e. placing it in a buffer for the NI (we assume asynchronous send operations). *NI occupancy* is the time spent in the NI processor or state machine to prepare each packet of the message and place it in an output queue for transmission. Packets are of variable size, depending on how much of the message data is available to the NI at the time, and are up to 4 KBytes long. *I/O bus bandwidth* is the main determinant of the host to network bandwidth, since network links and memory buses tend to be much faster. Finally, *message handling cost* models the time from the moment a message reaches the head of the NI incoming queue to the moment the handler for the message starts to execute on the main processor; time and contention before reaching the head is simulated in detail and is not included in this parameterized cost. Incoming data messages (as opposed to requests) do not invoke a handler, and are deposited directly in host memory by the NI without causing an interrupt or requiring a receive operation on the processor [3, 5]. Hardware link latency is kept fixed at 20 processor cycles since it is usually small compared to the other costs.

All costs we discuss for the communication architecture and the protocol are normalized to processor cycles. Table 3 shows the parameter values we use in the base system. The sets of values we consider for the communication layer are: The *achievable* set, which is modeled after a cluster of Intel PentiumPro nodes connected by a Myrinet interconnect. Occupancy per packet is high because the processor in the Myrinet and most other commercial network interfaces is slow. However, since packets can be as large as 4 KBytes, occupancy can often be amortized. Recall that all contention in the system is modeled in detail, so when costs are changed the effects on contention are simulated as well. The *best* set, where all protocol costs are set to zero. The *halfway* set, where all values are halfway between the first two sets (more realistic than the best set). The *worst* set, where all values are doubled compared to the achievable set (i.e. worse, relative to processor speed).

Parameter	Achievable	Best	Worse	Real
Host Overhead	600	0	1200	600–800
I/O Bus Bandwidth	0.5	2	0.25	0.45
NI Occupancy	1000	0	2000	1000
Message Handling Cost	200	0	400	N/A

Table 3: Communication parameter values. The first three columns describe the values we use in our study as the Achievable, Best and Worse sets of values. Units are in cycles (and bytes/cycle for bandwidth). If we assume a 1 IPC, 200 MHz processor the achievable values are from top to bottom 3μs, 100 MBytes/s, 5μs and 1μs. The fifth column contains the values measured using a real communication library (VMMC [5] on a network of Intel Pentium nodes connected with Myrinet). The message handling cost is obtained from the polling–based Blizzard–S and Shasta systems (Personal communications.).

Presentation of results: We use the following nomenclature to refer to the parameter configurations in the two system layers. C and P refer to the communication and protocol layers, respectively. The set of values used for each layer is denoted as a subscript. The achievable set is denoted with the digit 0, the best set with the symbol $+$, the halfway set between achievable and best with $\frac{1}{2}$, and the "worse" set with $-$. Thus the configuration corresponding to what is achievable today is C_0P_0, and the zero–cost configuration for both the communication and protocol parameters discussed earlier is C_+P_+.

Figure 3 presents all the main speedup results in this paper. Breakdowns of execution time are presented in Figure 4. As we go through the next few sections, we shall focus on portions of the data in these figures.

For every application, there is a graph in Figure 3 with bars depicting speedups for different combinations of settings for the three layers. Each application graph has two major sets of bars, on the left for the SVM protocol and on the right for the FG protocol. When an application has two versions, there are two sets of bars for each protocol, separated by a space: original and restructured. For ease of comparison, speedups are always measured with respect to the same best sequential version, and the order of arrangement of the bars for the restructured version is a mirror image of the order for the original version.

The color of the bars in the speedup graphs encodes some information as well. The bars with the same color represent configurations with the same communication parameter values, with only the protocol layer parameters being modified across them. Thus, black bars represent varying protocol layer parameter sets while keeping communication parameters fixed at the achievable set, uniformly shaded bars with communication layer parameters fixed at the worse set, and striped bars with those parameters fixed at the best set. Unshaded bars show "ideal", 16-fold speedup as a reference point, which is close to the algorithmic (PRAM) speedup for these applications.

The breakdown graphs (Figure 4) divide the time into components, such as busy time, local cache stall time, data wait time or time spent waiting for communicated data, lock wait time, barrier wait time, and protocol overhead. Breakdowns are shown averaged over all processors for a given execution, to save space. However, in analyzing the results we refer to per–processor breakdowns and more detailed per–processor statistics gathered in the simulator to understand imbalances as well. The averaging may sometimes lead to discrepancies between the heights of the breakdown bars and the speedup bars; however, the alternative of showing the worst or last–to–finish processor instead is often unrepresentative in terms of the breakdowns.

3 Results

Let us first look at the results on only the base or achievable system C_0P_0, to make some basic comparisons and place the results in context with previous work. This includes comparing SVM with FG with our system assumptions, and validating the application restructuring results for SVM from [11]. Then, we will examine new application restructuring results for FG, the impact of the system layers, and the synergy between layers.

3.1 Results for Base Architecture

Consider only the base C_0P_0 bars (the black bars to which the arrows point in Figure 3). For both the original and the restructured applications, we see that FG either equals or outperforms SVM. FG is especially better in applications that use locks frequently, like Barnes-Original and to a lesser extent Water-Spatial and Volrend, and those like Radix or Ocean-Contiguous in which coarse granularity causes a lot of false sharing or fragmentation (false sharing is also a problem with the image and task queues in Volrend due to intervening synchronization). The only case in which SVM does better is the restructured version of Barnes (Barnes-Spatial). Note, however, that this comparison does not account for access control costs in FG. Even with Shasta's aggressive software instrumentation costs (see Table 1 with all caveats of Section 2), performance would be much closer, with each protocol outperforming the other in different applications. Applications with more complex pointer references may incur higher instrumentation overheads. These results for FG versus SVM would also be similar to the results obtained on the Typhoon–zero platform, which provides commodity-oriented hardware support [22]. (The communication parameters here are different than on Typhoon–zero: the C_0P_0 platform has somewhat more bandwidth relative to processor speed, helping SVM, but much more efficient access control, helping FG.) If we disallow application-specific granularities, using 128 bytes or 256 bytes (the best overall granularity found in [22]) in all cases, we find (not shown) that several of regular applications perform better under SVM.

Qualitatively, it appears that the protocols are similar at this scale with realistic access control, differing based on application. For both protocols however, the speedups on C_0P_0 are clearly far from ideal, so there is a lot that can be gained from optimizing the layers.

3.2 Impact of the Application Layer

The original versions of the applications we use (Table 1) are described in [20]. If evolutionary communication trends hold relative to processor speed, and hardware support to reduce protocol costs is not forthcoming, application restructuring may be the only way to improve performance. For SVM, the result of comparing the two black C_0P_0 bars for each application agrees with that of [11]. The sources of the often large and sometimes dramatic improvements, described in [11] include: (i) making actual access granularities to remote data larger in the application (e.g. writing to a local buffer first in Radix or using rowwise instead of square partitioning in Ocean), (ii) reducing locking and fine–grained synchronization at perhaps some cost in load balance (e.g. in the tree building phase of Barnes), and (iii) in some cases improving the initial assignments of tasks so there is less need for task stealing (which is now very expensive due to synchronization and protocol activity, e.g. in Volrend; note that in Volrend restructuring also greatly improves false sharing and fragmentation in the image at page granularity, and hence data wait time). Many characteristics of the applications relevant to *SVM*, including sharing patterns, message frequencies and message sizes, are described in [10, 22, 2].

For *FG*, not examined in [11], restructuring helps significantly in cases where application access granularity is made larger (e.g. Ocean), since it allows a larger granularity to be

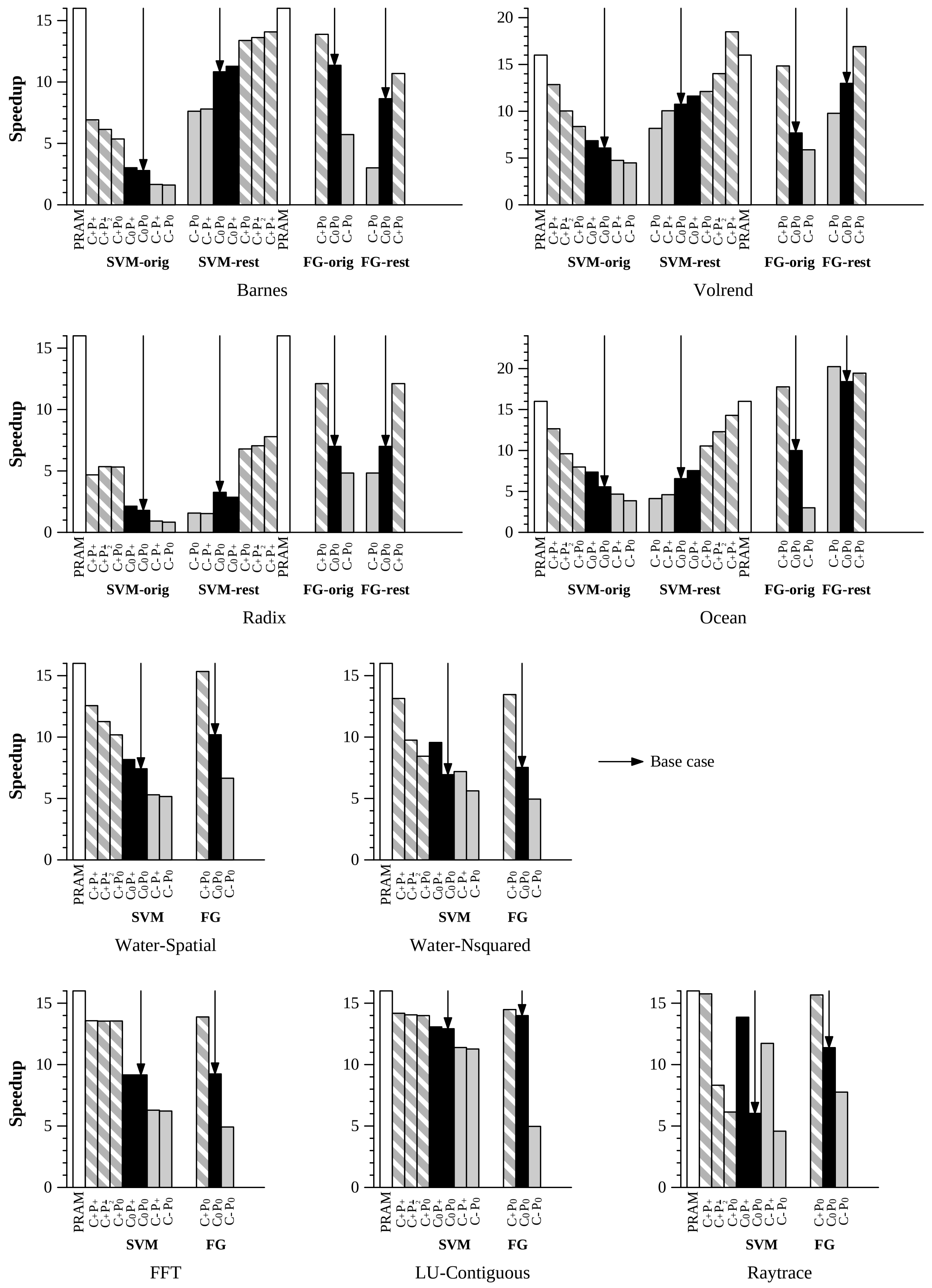

Figure 3: Application speedups. Please see text in **"Presentation of results"** in Section 2 for how to interpret the bars and focus attention on some of them.

used successfully in the system, and when the need for task stealing is reduced (Volrend). However, restructuring that helps *SVM* can sometimes hurt *FG*, such as in cases where load balance is compromised to reduce lock wait time (e.g. Barnes). Locks are not so problematic in *FG* as in *SVM* protocols: protocol activity is not postponed till synchronization points, and satisfying finer–grained block faults within critical sections is less costly than satisfying page faults, so there is less serialization.

Instead, load imbalance matters more. Overall, restructuring is not beneficial in as many situations as in *SVM* because the *FG* protocol and granularities look a lot more like the hardware–coherent protocols for which the original applications are optimized.

3.3 Impact of the Protocol Layer

Even with restructuring, performance of software DSM is still far from hardware DSM for many classes of applications and improvements in the lower layers are clearly needed. This section first examines the impact of idealizing protocol costs (Table 2), i.e. moving from C_0P_0 to C_0P_+. Protocol costs may be improved by selectively adding hardware support or by altering the protocol.

Impact for FG: Focusing only on the black C_0P_0 bar and the C_0P_+ bar just next to it for only the original versions of the applications for *FG* in Figures 3, we see that improving protocol costs does not have a significant impact on *FG*. *FG* does not have diffs or complicated protocol operations at synchronization points. There are only two protocol–related costs for *FG*: access control and protocol handlers. A rough idea of the impact of instrumentation costs for software access control (which we do not simulate, as discussed earlier) can be gleaned from Table 1. Since protocol handlers are very simple in *FG*, the cost of running the handlers is very small compared to the communication layer costs associated with handling the messages and invoking the handlers. Instrumentation of the original runs for *FG* shows that changing the cost of handlers (within a reasonable range) will not really affect performance, so we do not simulate different costs for *FG* protocol handlers.

Impact for SVM: In *SVM*, protocol costs have greater impact. However, even the complete elimination of protocol costs usually does not lead to dramatic improvements in performance if the underlying communication architecture is kept the same (with the exception of Raytrace). For coarse–grained–access, single–writer applications like FFT and LU, there is very little expensive protocol activity to begin with. Several of the other applications that use a lot of locks (especially) or have irregular access and invalidation patterns benefit to varying extents: Barnes-original, Water-Spatial and Volrend-original about 10%, and Radix 20%. Greater improvements are seen in Ocean-Contiguous, Water-Nsquared and especially Raytrace.

Table 4 shows the percentage of the protocol time spent in diff computation and application. The rest of the protocol time is mostly spent in executing protocol handlers. The other protocol cost components are very small. Regular, single–writer applications spent very little time in handlers and there is almost no diffing at all. Water-Nsquared is a regular application but not single-writer; it computes many diffs for a lot of migratory data when it is updating forces. For many irregular applications, diff–related computation at synchronization points, such as frequent locks, usually dominates. Raytrace does not have much synchronization (and hence diff activity) but has a very large number of fine–grained messages due to irregular access, so protocol handler cost is a large fraction of data wait time and processors spend a lot of time in the handlers themselves. Ocean-contiguous behaves similarly, although it is a regular application, due to fine-grained (one-element) remote accesses at column-oriented partition boundaries in the near-neighbor calculations. When diff cost is a problem, hardware support for automatic write propagation [3] can eliminate diffs [13, 9, 10], at the potential cost of contention and/or code instrumentation; we might expect it to help substantially in Water-nsquared and Radix. Finally, improving protocol costs halfway ($C_0P_{1/2}$, not shown in the figures) usually provides about half or less of the benefit of eliminating them (more on this in Section 3.5).

Application	Protocol (%)	Diff Compute (%)
Barnes-Original	7.2	68
FFT	0.5	0
LU-Contiguous	0.6	20
Ocean-Contiguous	3.0	10
Radix-Original	7.0	85
Raytrace	58	31
Volrend-Original	13.2	85
Water-Nsquared	10.7	80
Water-Spatial	3.2	94

Table 4: Percentage of total execution time spent by processors in protocol activity, and percentage of this protocol activity time that is spent in diff computation and application.

We see from the restructured applications (black bars in Figures 3), that even idealizing protocol costs together with application restructuring is insufficient to approach the desired levels of performance that these applications can achieve on hardware-coherent machines. The large page granularity usually does not compensate for a less aggressive communication architecture even if protocol costs are magically made zero.

3.4 Impact of the Communication Layer

In recent times, overheads and bandwidths have scaled just a little bit slower than effective processor speed (about 50% per year versus about 70%) and NI occupancy can be assumed to scale with processor speed. It is useful to examine both what might be gained if thresholds or breakthroughs occur in communication performance and what might be lost in parallel speedups if it degrades relative to increasing processor speeds. Let us now return to the original protocol and examine the impact of improving the communication architecture only. In the best set of parameters, only the I/O bus bandwidth is finite or non-zero, being set to the same value as the memory bus bandwidth. Note that contention is still modeled in all parts of the system.

Impact for FG: Figure 3 shows that the best communication layer makes a dramatic difference to speedups (C_+P_0 versus C_0P_0), bringing them quite close to the ideal for the original applications. Except for the (small) protocol handler cost, the *FG* system becomes like hardware cache coherence with a very fast communication architecture. Of course, adding in instrumentation costs for software access control would make the speedup worse, but the impact of the communication architecture is clear. The cases where C_+P_0

speedups are substantially less than ideal (Barnes cases) are due to load imbalance and the still significant synchronization cost.

Impact for SVM: For *SVM*, the effects of the best communication architecture alone are substantial but less dramatic than for *FG*. Applications that have little protocol activity, coarse–grained remote access and few or no locks, like FFT and LU, are constrained only by communication layer costs, so they now achieve close to ideal performance just as with *FG*. The irregular applications are helped substantially, but speedups with C_+P_0 are still as low as 5–7 on 16 processors for some of them. The breakdowns (Figure 4) show that the data wait time is mostly eliminated, but time waiting at locks is often still high. Imbalances in this time cause time waiting at barriers to be high too. Overall, starting from C_0P_0, communication costs generally seem to have greater impact than protocol costs even for *SVM* (with exceptions like Water-spatial, Raytrace, and some restructured versions); but as we shall see in Section 3.5, once communication costs are improved even to halfway (or restructuring is performed), protocol costs begin to gain in importance.
Effects of Individual Parameters: Figure 5 shows the impact of varying only one communication parameter at a time for some applications. These data for *SVM* were discussed in detail in [2], but the data for *FG* and the comparison is new. *FG* clearly depends mostly on overhead and occupancy, whereas *SVM* depends mostly on bandwidth (overhead and occupancy do not matter much, and interrupts which dominated in [2] are not used). These data also show the points where crossovers in protocol performance might happen (ignoring access control). Together with access control costs and technology trends (e.g. bandwidth is increasing more rapidly than overhead is dropping), they can help draw conclusions about choices among the protocols for specific architectures or in the future.

Overall, we conclude that dramatic improvements in communication relative to processor performance will dramatically improve data wait time and *FG* performance; for *SVM*, performance improves substantially (usually more than for just improving protocol costs, at least starting from C_0P_0) but is not enough since part of the problem in irregular applications is due to fine–grain lock synchronization. Since I/O bandwidth remained finite in C_+P_0, we experimented with doubling the I/O bandwidth compared to the C_+P_0 configuration. Applications that exhibit contention in the network interface (e.g. FFT) benefit substantially and the performance improves to halfway between C_+P_0 and the PRAM speedup for the application.

3.5 Synergy between Layers

Finally, let us examine both how improvements in the system layers affect the benefits of application restructuring and are affected by them, as well as the synergy between pairs of layers.
Synergy in FG: Application restructuring for *FG* helps primarily due to the ability to use coarser granularity. Directly improving the communication architecture therefore reduces the impact of restructuring. When instrumentation is used for access control, its cost may also improve with restructuring (due to coarser access granularity and spatial locality), but this effect is much smaller.

For example, in Barnes, where the restructured version hurts rather than helps due to increased load imbalance, per–process execution–time breakdowns reveal that the key load imbalance is that among data wait times. A faster communication architecture reduces this imbalance, leading to less damage from restructuring. Improving instrumentation costs does not interact much with communication layer performance since the two are quite orthogonal. Thus, in *FG* there is not much to be gained from the synergy between layers, especially if the communication layer is aggressive.
Synergy in SVM: In *SVM* we find a lot of interesting interactions and a lot more synergy among layers. First, how application restructuring affects, and is affected by the two system layers depends on the application. In Barnes-original, restructuring reduces the amount of locking and serialization, and while its importance diminishes as system layers improve, it remains important all the way to C_+P_+ and even beyond. This is because some costs within the node (that are kept fixed) remain, and bandwidth on the I/O bus is still not infinite. Although protocol processing at synchronization points is free, some cost for communication (whether all needed or due to fragmentation) remains and hence so does some serialization at locks. This is especially true for Barnes-Original because every critical section incurs more than one page miss and a lot of fragmentation within it. The $C_{++}P_+$ (Table 5) case does much better. Because reducing communication costs had a dramatic impact on lock serialization in Barnes-original, the communication layer affects the impact of restructuring more than the protocol layer. As for protocol costs, they take on greater relative significance after restructuring than before it because a major other bottleneck has been removed; however, even in the restructured version communication costs are more important than protocol costs. In Ocean, on the other hand, protocol costs impact the benefits of restructuring more than communication costs; also, the relative importance of the two system layers is reversed in the restructured version compared to the original. This is because the restructured version greatly reduces the number of messages but not the amount of useful data transferred, so the impact of message handling (protocol) cost is reduced relative to that of communication. Volrend and Radix behave similarly to Barnes in this respect (only less so): restructuring remains important as lower layers improve, and the impact of restructuring is reduced more by communication costs than by protocol costs (increasing the importance of protocol costs in the restructured version). In Volrend, C_+P_+ finally allows the critical sections used for task stealing to be efficient enough for stealing to be successful at load balancing. While Radix continues to have low performance due to high communication and contention, restructuring makes a substantial difference (66%) even at C_+P_+. It takes $C_{++}P_+$ to deliver good speedups, as in Barnes-Original but for a different reason.

In *SVM*, the two system layers themselves show substantial synergy as well. While protocol costs are limited in the improvement they can provide in the original C_0 case in many applications, once communication costs are dramatically reduced (even halfway) the impact of reducing protocol costs becomes much greater (Figure 3), and vice versa when protocol costs matter (e.g. Ocean-Contiguous, Volrend).

Overall, in *SVM* no two layers are enough to improve performance to rival hardware coherence for all applications, and all three layers are important to improve. The application layer and either of the lower layers combine to give a much greater improvement than any system layer does individually. In most cases, the order of impact is application followed by communication followed by protocol. For both *SVM* and *FG*, the communication layer appears to be more important to making the programmer's job easier (i.e., making application restructuring less important). Restructuring changes the balance between the relative importance of protocol and communication costs, usually making the

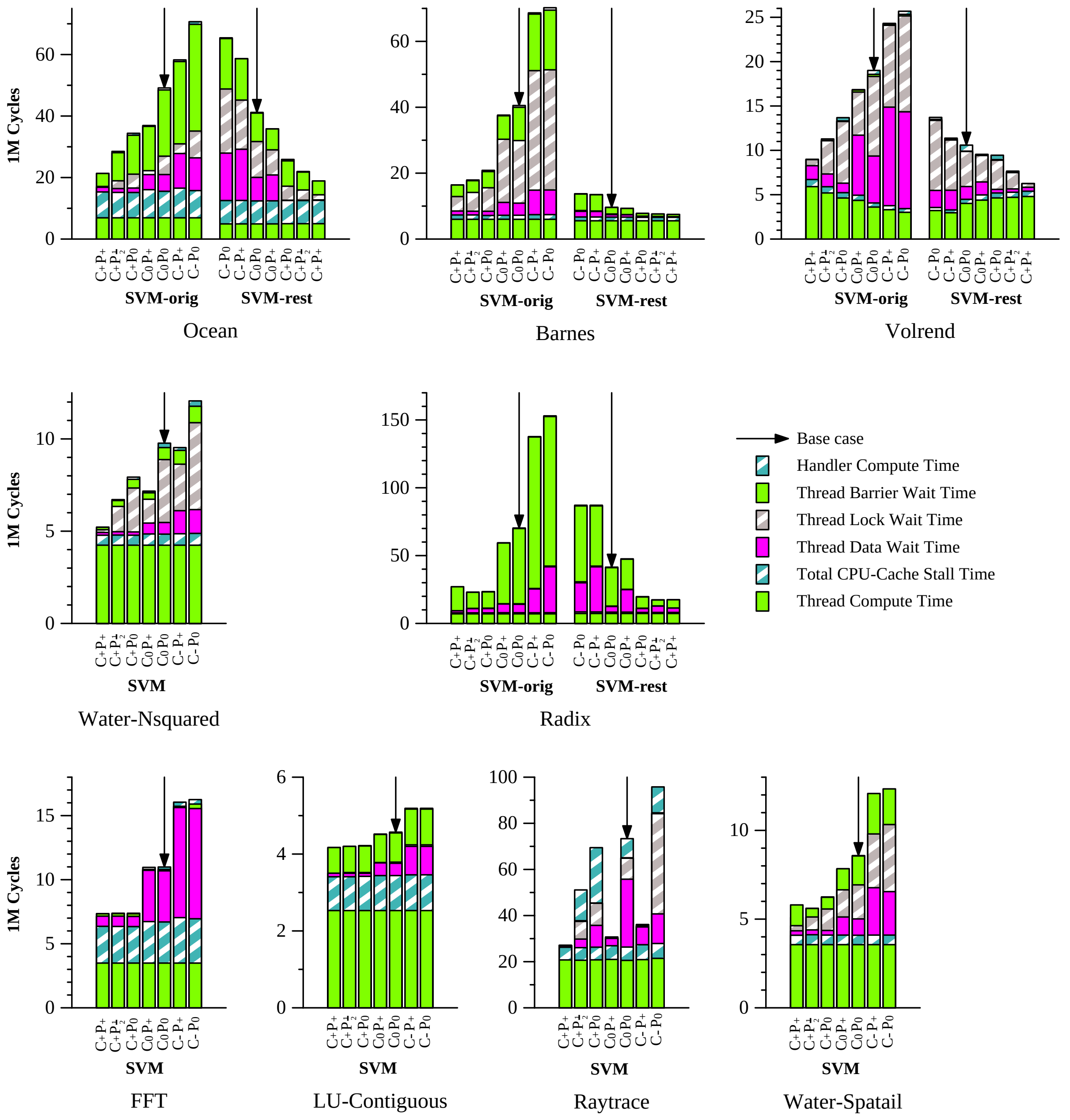

Figure 4: Application execution time breakdowns.

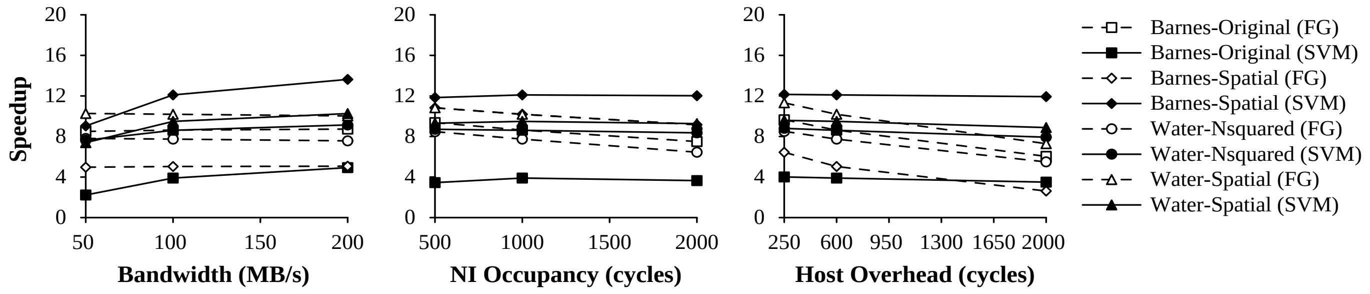

Figure 5: Speedups with communication architecture parameters. Each figure is a different parameter for various applications. Only four applications are shown; for some of the more regular ones *FG* uses a coarse granularity too and the behavior of the two protocols is very similar. The curves with dashed lines are for *FG*, and with solid lines are for *SVM*.

protocol costs more important than they were before. Protocol costs also become more important once communication costs are improved (to say halfway), and vice versa. Table 5 summarizes for *SVM*, for each application, whether communication cost or protocol cost is initially more important, whether $C_{1/2}P_{+}$ is better than $C_{+}P_{1/2}$, and what combinations of parameter values does it take to achieve about 10–fold speedup on 16 processors, which tells us both what is more important and what is needed. The need for improvement in all layers, whether through basic performance characteristics as examined here or through altered functionality to avoid some of these costs [1] is apparent from the last category. More detailed trends can be obtained from the graphs.

4 Related Work

The impact of individual communication architecture parameters on performance has been studied for *SVM* [2], hardware coherence [8], and a non–coherent SAS [15]. However, it has not been studied for *FG*, or by varying parameters simultaneously, or by examining the protocol layer as well. Various hardware supports to accelerate protocols have been examined for SVM in [9] and [13], and for *FG* in the Typhoon–zero prototype [17].

A previous study [6] compared the performance of several non-home-based *SVM* protocols; it examined two network bandwidths and varied a unified software overhead cost. The study included only three small kernels and one application. Both *FG* and *SVM* were compared on a particular hardware platform with a fixed set of cost parameters in [22], with a focus on the impact of consistency models and coherence granularity. Its results are qualitatively similar to the ones we observe here for *FG* versus *SVM*. This is the first study we know of that examines the impact of all these protocol and communication cost issues in an integrated framework for a wide range of applications (as well as including the impact of the application layer too).

5 Discussion and Conclusions

We have presented a framework for studying the limitations on the performance of software shared memory systems, in terms of the layers of software and hardware that can be improved and how they impact the end performance of a wide range of applications. We have studied the limitations and performance effects through detailed simulation, treating the system layers (protocol and communication layer) through only their cost parameters. The main limitations of the present work include the fact that it does not simulate true instrumentation–based fine–grained software shared memory, but rather a system with very efficient hardware access control but software protocols. Our main conclusions can be summarized as follows.

First, for currently achievable architectural values the variable–grained *FG* and page–grained *SVM* protocols appear to be quite comparable overall, especially if we factor in aggressive but realistic access control costs. This assumes that *FG* is allowed to choose the best granularity of communication and coherence for each application; for cases like FFT where *FG* benefits from coarse granularity, we have found using a finer granularity to perform substantially worse. In general, *FG* will be worse when access control (e.g. instrumentation) costs are high or when it causes too many small messages to be transferred; *SVM* is worse when applications have a lot of fine-grained synchronization or a lot of fragmentation in page-grained communication (or especially both, as in Barnes-Original).

Second, for *FG* the communication layer is key; protocol layer costs are not very significant, except for instrumentation cost when software access control is used. Application restructuring is also not so widely important, at least when starting from applications that are well tuned for hardware–coherent DSMs. Exceptions are when restructuring enables a coarser granularity of communication to be used effectively (as in Ocean), and when it reduces the frequency of task stealing (Volrend). As might be expected, the communication parameters that matter most are overhead and NI occupancy, while bandwidth matters only when a coarser granularity is used.

Third, for *SVM*, contrary to our results for *FG*, we find a much richer set of interactions. (i) Improvements in all three layers are often necessary to achieve performance comparable to efficient hardware–coherent systems. No one layer or even pair of layers is enough across the range of applications. (ii) There is synergy among the layers in that improving one system layer (even halfway) allows the other to have greater impact, so that realistic improvements to both system layers relative to processor speed may go a good way, and improving the applications often does not substantially diminish the further impact of system layers. (iii) Application restructuring is used both to make access patterns to communicated data more coarse grained and to reduce the frequency of synchronization; when applicable, it usually outperforms the gains from idealizing any one other layer. Thus, if useful guidelines can be developed for programming *SVM* systems, they can be extremely helpful. (iv) If only one system layer is targeted for improvement, it should be the communication layer; among these parameters, the greatest dependence of performance is on bandwidth, and it is quite insensitive to improvements in the other parameters (when interrupts are used instead of polling, interrupt cost is a key bottleneck [2]). (v) Among protocol costs, the sensitivity is usually greatest to the costs associated with diffs, primarily diff creation. Handler cost can matter substantially in applications that are more constrained by data message count than by synchronization. Table 5 summarizes the key results for *SVM* in terms of relative importance and what is needed for good performance.

At a higher level, many interesting tradeoffs remain between *FG* and *SVM*. For *SVM*, the fact that it needs synergistic improvements in multiple layers rather than just the communication layer may appear intimidating, and restructuring applications in fact is difficult (although recent results show that similar restructurings are often needed for hardware-coherence at larger scale). But it is in some ways a promising sign, especially since the communication layer may be difficult to improve relative to processor speed, and is not so much under the control of the *SVM* architect. Also, the communication parameter it relies on most, bandwidth, is the most likely to improve. *FG* requires overhead and occupancy to improve further, which may be more challenging under current memory and network speed trends. For *SVM*, while the heretofore major challenge of false sharing appears relatively easy to overcome in applications with today's protocols, the large page granularity of communication seems to get in the way of applications that suffer fragmentation in communication or fine-grained locking (often associated with true sharing as well).

Application	Original					Restructured				
	$C_+P_0 >$ C_0P_+	$C_+P_{1/2} >$ $C_{1/2}P_+$	Min Levels			$C_+P_0 >$ C_0P_+	$C_+P_{1/2} >$ $C_{1/2}P_+$	Min Levels		
			Com.	Prot.	Comb.			Com.	Prot.	Comb.
FFT	**Yes**	**Yes**	$\frac{1}{2}$	N/P	$C_{1/2}P_0$	N/A	N/A	N/A	N/A	N/A
LU	Yes	Yes	0	0	C_0P_0	N/A	N/A	N/A	N/A	N/A
Ocean	Yes	No	N/P	N/P	$C_+P_{1/2}/C_{1/2}P_+$	**Yes**	Yes	$\frac{1}{2}$	N/P	$C_{1/2}P_+/C_+P_{1/2}$
Barnes	**Yes**	**Yes**	N/P	N/P	$C_{++}P_+$	**Yes**	Yes	0	0	C_0P_0
Radix	**Yes**	**Yes**	N/P	N/P	$C_{++}P_+$	**Yes**	**Yes**	N/P	N/P	$C_{++}P_+$
Volrend	Yes	Tie	N/P	N/P	$C_+P_{1/2}/C_{1/2}P_+$	Yes	Tied	0	0	C_0P_0
Raytrace	**No**	**No**	N/P	+	C_0P_+	N/A	N/A	N/A	N/A	N/A
Water-Nsquared	No	No	N/P	+	$C_0P_+/C_+P_{1/2}$	N/A	N/A	N/A	N/A	N/A
Water-Spatial	**Yes**	Yes	+	N/P	$C_+P_0/C_{1/2}P_+$	N/A	N/A	N/A	N/A	N/A

Table 5: Summary of system layer impact for *SVM*. $C_{++}P_+$ means that the communication architecture is even more aggressive than in C_+P_+ by further doubling the I/O bandwidth. *N/P* stands for *not possible,* and *N/A* for *not applicable*. Bold font is used when the difference is large. The columns for each version of each application are: (i) Is C_+P_0 better than C_0P_+, i.e. is communication more important than protocol. (ii) Is $C_+P_{1/2}$ better than $C_{1/2}P_+$. (iii) What minimum set of parameter values for each system layer is necessary, with the base level of the other layer, to get about 10–fold or greater speedup on 16 processors? What combination of levels does it take to achieve the same performance? For example, an $\frac{1}{2}$ under the entry called "Com." says that the halfway or better communication costs, together with any protocol costs starting from the original, is enough to obtain 10–fold speedup. ($C_{++}P_+$ is excluded since it is too unrealistic).

Acknowledgments: We thank the members of the PRISM group at Princeton for useful discussions, and Rudrajit Samanta additionally for help with graphs. We gratefully acknowledge the support of NSF and DARPA.

References

[1] A. Bilas. *Improving the Performance of Shared Virtual Memory on System Area Networks*. PhD thesis, Dept. of Computer Science, Princeton University, August 1998. Available as technical report, Princeton University TR-586-98.

[2] A. Bilas and J. P. Singh. The effects of communication parameters on end performance of shared virtual memory clusters. In *Proceedings of Supercomputing 97, San Jose, CA*, November 1997.

[3] 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 International Symposium on Computer Architecture (ISCA)*, pages 142–153, Apr. 1994.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[5] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, pages 388–396, April 1997.

[6] S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel. An evaluation of software distributed shared memory for next-generation processors and networks. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993. To Get.

[7] I. S. et al. Fine-grain access control for distributed shared memory. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–307, Oct 1994.

[8] C. Holt, M. Heinrich, J. P. Singh, , and J. L. Hennessy. The effects of latency and occupancy on the performance of dsm multiprocessors. Technical Report CSL-TR-95-xxx, Stanford University, 1995.

[9] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[10] L. Iftode, J. P. Singh, and K. Li. Understanding application performance on shared virtual memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[11] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.

[12] P. Keleher, A. 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, Jan. 1994.

[13] 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*, Feb. 1996.

[14] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[15] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effect of communication latency, overhead, and bandwidth on a cluster architecture. Technical Report CSD-96-925, Berkeley, Nov. 1996.

[16] S. Reinhardt, J. Larus, and D. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 325–336, Apr. 1994.

[17] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 34–43, New York, May22–24 1006. ACM Press.

[18] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *The 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[19] A. Sharma, A. T. Nguyen, J. Torellas, M. Michael, and J. Carbajal. Augmint: a multiprocessor simulation environment for Intel x86 architectures. Technical report, University of Illinois at Urbana-Champaign, March 1996.

[20] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, May 1995.

[21] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, Oct. 1996.

[22] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. Toonen, I. Schoinas, M. Hill, and D. Wood. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.