Monitoring Shared Virtual Memory Performance on a Myrinet-based PC Cluster

Cheng Liao, Dongming Jiang, Liviu Iftode[†], Margaret Martonosi, and Douglas W. Clark
Princeton University Rutgers University[†]
Princeton, NJ 08544 Piscataway, NJ 08855
{cliao, dj}@cs.princeton.edu iftode@cs.rutgers.edu mrm@ee.princeton.edu doug@cs.princeton.edu

Abstract

Network-connected clusters of PCs or workstations are becoming a widespread parallel computing platform. Performance methodologies that use either simulation or high-level software instrumentation cannot adequately measure the detailed behavior of such systems. The availability of new network technologies based on programmable network interfaces opens a new avenue of research in analyzing and improving the performance of software shared memory protocols. We have developed monitoring firmware embedded in the programmable network interfaces of a Myrinet-based PC cluster. Timestamps on network packets facilitate the collection of low-level statistics on, e.g., network latencies, interrupt handler times and inter-node synchronization.

This paper describes our use of the low-level software performance monitor to measure and understand the performance of a Shared Virtual Memory (SVM) system implemented on a Myrinet-based cluster, running the SPLASH-2 benchmarks. We measured time spent in various communication stages during the main protocol operations: remote page fetch, remote lock synchronization, and barriers. These data show that remote request contention in the network interface and hosts can serialize their handling and artificially increase the page miss time. This increase then dilates the critical section within which it occurs, increasing lock contention and causing lock serialization. Furthermore, lock serialization is reflected in the waiting time at barriers. These results of our study sharpen and deepen similar but higher-level speculations in previous simulation-based SVM performance research. Moreover, the insights about different layers, including communication architecture, SVM protocol, and applications, on real systems provide guidelines for better designs in those layers.

1 Introduction

Network-connected clusters of PCs or workstations are becoming a widespread parallel computing platform. To reduce the hardware cost of these systems, cache coherence hardware is often foregone in favor of software layers implementing Shared Virtual Memory (SVM) [18] as illustrated in Figure 1.

Although SVM implementations have been the focus of several past performance studies [12, 15, 17, 24], data-gathering approaches have varied widely. At the programming and protocol layer, run-time tools and software instrumentation have been used to detect high-level contributors to execution time: e.g., counts of synchronizations, page faults, and messages. Prototype evaluations in these studies indicate that the communication-related costs plus the software overhead are responsible for limiting the performance of the software shared memory approach.

Simulations have been the principal vehicle for much previous research on SVM performance [3, 13, 14, 17]. However, the simulation approach has several limitations which constrain one's trust in its results. Namely, fast simulators usually make simplifying assumptions about certain costs and behaviors, such as protocol costs or bus and network contention.

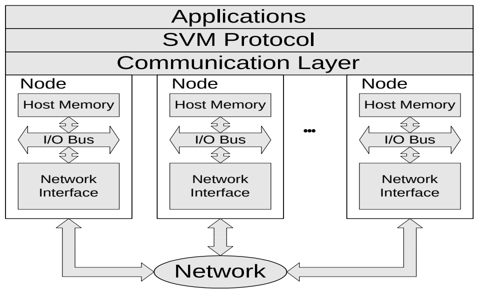

Figure 1: Layers that affect the end performance of SVM applications.

The alternative to simulation is to collect detailed information by monitoring the protocol execution in a real system. In this paper we describe a software performance monitor which we have used to understand the performance of a software shared memory prototype implemented on a Myrinet-based [6] PC cluster. We have embedded additional monitoring and global clock synchronization code into the communication firmware running on the Myrinet network interface card. This allows us to gather low-level statistics on, e.g., network latencies, interrupt handler times and inter-node synchronization. Although we initially designed this monitor to analyze and tune message passing programs, this paper demonstrates the flexibility of the approach by describing our experiments using it to understand and tune a collection of SVM applications from the SPLASH-2 [23] suite.

The performance monitor interacts with the SVM protocol via a simple and efficient interface. By interfacing the performance monitor with the SVM protocol we can track the time spent in various communication stages during main protocol operations: remote page fetch, remote lock synchronization, and barriers (global synchronization). Our detailed results show that the remote request contention in the hosts on interrupts can serialize their handling and artificially increase the page miss time. In addition, contention in the network and network interface can increase page miss time as well. This increase then dilates the critical section within which it occurs, increasing lock contention and thus causing lock serialization. Furthermore, lock serialization is also reflected in the waiting time at barriers. Although some of these effects have been speculated from higher-level statistics and detailed simulations [2, 3, 12, 13, 14, 15, 17, 24], the initial interrupt delay and the exact succession of this cascade of effects could not have been traced exactly without the performance monitor on real systems.

The contributions of this work are two-fold: we illustrate the design of a firmware-based performance monitor and we apply it to further understand application behavior on SVM systems.

The remainder of this paper is structured as follows. Section 2 describes the design of the SVM system and applications being studied, and Section 3 describes the performance monitor we implemented. Section 4 then presents a series of experiments done using the performance monitor on the SVM protocol. Based on the new insights got from these experiments, Section 5 outlines a series of guidelines to SVM system implementers and application programmers. Finally, Section 6 discusses related work and Section 7 offers our conclusions.

2 Implementing SVM On A Myrinet-based PC Cluster

In this section, we describe the SVM prototype we implemented on a Myrinet-based PC cluster [9] built at Princeton. Based on high-level timings of the SVM system, we preview its performance. We also show that some crucial performance characteristics are hard to understand using high-level profiling alone.

2.1 Myrinet-based PC Cluster

The Myrinet-based PC cluster implements the Virtual Memory-Mapped Communication model (VMMC) [4] on a Myrinet network of PCI-based PCs. Myrinet is a high-speed local/system area network for computer systems [6]. A Myrinet network is composed of point-to-point links that connect hosts and switches. The Myrinet-based PC cluster used in this study consists of 8 PCI PCs connected to a Myrinet switch via Myrinet PCI network interfaces. Each PC is a Gateway P5-166 with a 166MHz Pentium CPU, and has 512KB L2 cache and 64MB main memory. Each network interface is controlled by a 33MHz LANai processor running communication firmware. We embed our performance monitor in this code.

2.2 SVM Implementation

The SVM system implements an all-software, home-based lazy release consistency (HLRC) protocol [24]. The HLRC protocol assigns each page to a *home* node. To alleviate the false-sharing problem at page granularity, HLRC implements a multiple-writer protocol based on using “twins” and “diffs”. After an acquire operation, each writer of a page is allowed to write into its local copy once a clean version of the page (twin) has been created. Changes are detected by comparing the current (dirty) copy with the clean version (twin) and are recorded in a structure called a *diff*. At a release operation, diffs are propagated to the designated home of the page, not to other sharers. The home copy is thus kept up to date. Upon a page fault following a causally-related acquire operation, the entire page is fetched from the home.

2.3 Application Overview

2.3.1 Applications

Benchmarks	Problem Size	Speedup (8procs)
Barnes	8192particles	2.36
Cholesky	tk14	1.31
FFT	256K points	4.25
Ocean	130×130 grids	4.12
Radix	2M keys	1.88
Volrend	128^3 head	3.06
Water-Nsquared	4096 molecules, 3 steps	7.06
Water-Spatial	4096 molecules, 5 steps	7.27

Table 1: Applications, problem sizes, and speedups on the SVM system on the Myrinet-based PC cluster with 8 processors.

This study uses 8 SPLASH-2 [23] applications with a range of characteristics. Table 1 shows the problem sizes of the 8 benchmarks and their speedups on the SVM system with 8 processors. We use important computational kernels as well as real applications, both regular and irregular. We also explore applications with a range of behaviors: different inherent communication and data referencing patterns, and different access granularities to data that interact with SVM page granularity. We are especially interested in applications that are challenging in performance on the SVM system in this study, to demonstrate the need for performance debugging tools in SVM system design and understanding.

2.3.2 Application Performance on the SVM System

Table 1 indicates that most of the benchmarks cannot reach more than 50% of the parallel efficiency on the SVM system (i.e., a speedup of 4 on 8 processors), while all of them are known to deliver very good scalability on hardware CC-NUMA machines of 8 processors. As a preview of some of the performance bottlenecks, Figure 2 shows a breakdown of program execution times, obtained from high-level profile timings of our SVM protocol.

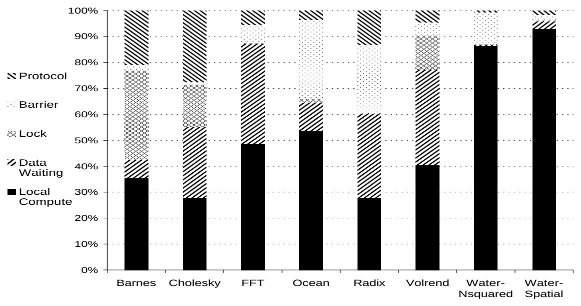

Figure 2: Runtime breakdowns of applications. Local Compute time mainly contains time spent on the local host, including instruction execution and local memory time. Data Waiting time is the time spent waiting for data to arrive at remote page faults. Lock and Barrier times include both the wait time and the communication time of the synchronization events. Protocol time is the time the processor spends in protocol processing on incoming or outgoing transactions, including the time spent computing and applying diffs and services to remote requests.

Figure 2 reveals that the bottlenecks of application performance have much to do with the communication architecture of our SVM system, including data waiting time, synchronization and protocol overhead. The question is why and how they happen. While it is relatively easy to get high-level breakdowns like Figure 2 from instrumenting the SVM code itself, detailed low-level data is impossible to get with this approach. Since the network-layer performance characteristics are crucial to the design and utilization of SVM systems, they were studied through simulation in previous research [3, 13, 14]. Most current SVM simulators are inadequate in modeling contention effects on buses, network links, and at the network interfaces. They are also slow. Given that many important computational kernels and real applications do not perform well on the SVM system, performance diagnosis tools are badly needed on the real systems on which SVM is built.

3 Performance Monitoring in the Myrinet-based System

To gain insights into the behavior of applications on the Myrinet-based PC cluster, we have built a performance monitoring tool for this system. It is a purely firmware-based performance debugging tool focused on collecting network-level data and tying it to higher-level software events. The core of our monitor is firmware run by the LANai processor on the Myrinet network interface card. Working at the network-firmware level gives us access to crucial determinants of application performance not available in higher-level software. Several pieces of the tool’s functionality would be infeasible or costly to implement in higher-level software, such as: (i) a globally synchronized clock, (ii) the ability to measure network-level latencies, and (iii) access to interrupt-driven handler invocation times. Network-level monitoring also allows us to build monitoring infrastructure that applies to a range of different higher-level programming models. We have used our strategy to performance debug several message passing applications in the past [19]; this paper concentrates on its use for SVM applications.

Copies of the monitoring system run on each node's network interface. The monitor software adds time-stamps and sequencing information to messages outgoing from a particular node, and also processes this extra information as packets arrive at a node. As incoming messages are processed, the firmware monitor measures packet characteristics such as its size, and also computes network latency by subtracting the packet's time-stamp from the current node time. For all programs, the monitor keeps a multidimensional histogram of packet sizes, sending and receiving node IDs, and packet latencies. As particular monitoring needs arise, however, the firmware nature of the tool also allows the statistics gathered to be easily customized to the particular experiment and protocol. This allows us to gather some of the detailed lock and page acquire statistics presented in Section 4. A globally-synchronized clock is needed to keep the time-stamps generated by the sending node consistent with the clock at the receiving node; this is described in Section 3.1. Statistics data are kept in LANai data structures until after the experiment has completed; at that point, the data can be post-processed to be presented in a number of ways, including 3D plots of latency statistics versus packet size, sending node, etc.

3.1 Global Clock Synchronization

To keep time consistent across the loosely coupled nodes, the monitor employs a global clock synchronization algorithm based on Christian's algorithm [7]. In our current system, Node 0 is always responsible for collecting and distributing global clock information. Periodically, it contacts other nodes querying them for their current time; when the answer is returned, Node 0 computes the time difference between the pair. When Node 0 finishes the collection, it broadcasts the time difference table to all the other nodes. Other nodes then update their clocks based on the table received from Node 0. The centralized role of Node 0 reduces clock synchronization messages dramatically, but if this were our sole means of aligning clocks, we would need to perform a global resynchronization roughly once per second. This frequent global resynchronization, unfortunately, would impose a rather heavy overhead on the system.

Instead, to decrease global clock synchronization communication, we re-synchronize globally much less frequently, every 5 minutes in these experiments, rather than once per second. To maintain acceptable accuracy, we note that clock drifts are roughly constant within short time periods. With this assumption, we allow each node to "re-synchronize" *without* global communication by interpolating current time differences based on measured drift rates and the time difference table from the most recent global resynchronization.

3.2 Coordinating Performance Monitor and SVM System

The communication layer often induces high synchronization and data waiting overheads in SVM systems. Figure 2 illustrates that these overheads are quite significant in these applications. Thus, in order to understand the performance issues, we have to study the interactions between applications and underlying communication layer. In the SVM system, these protocol bottlenecks can be a complex series of network messages, each following different paths within the communication architecture. Spotting performance problems in this complicated flow is particularly difficult for the programmer and the SVM system designer. Since the performance monitor is at the network level, it can be difficult to tie low-level network statistics up to the particular program or SVM events that cause them. This section will first consider how the performance-intensive SVM operations interact with the communication layer; an understanding of this software structure will help us describe, in the next subsection, the performance monitor interface for SVM systems.

3.2.1 Background on SVM Operations

While there are several performance-critical SVM operations that interact with the communication layer, we will summarize them using two examples: remote page fetch and remote lock acquire. These are described below.

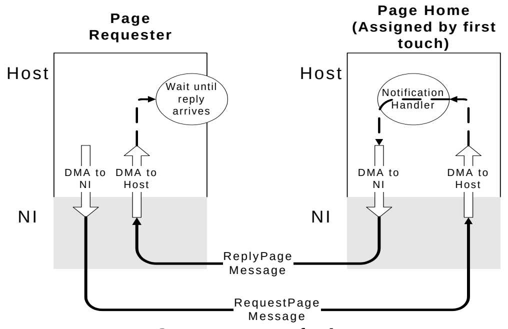

Figure 3: Remote page fetch operation.

Remote page fetch: In the HLRC SVM system considered, each memory page is assigned a home node, which is the one that first touches it. There are two messages involved in a remote page fetch. When a page fault occurs on a node, it sends a request message to the page's home. The home node then deals with this request and sends the page back. Meanwhile, the page requester waits on the page fault. These two messages are drawn in Figure 3 as thick arrows between the two hosts. The hollow arrows inside each host show data flow between the host memory and the network interface memory on the Myrinet card. The page requesting message is sent with "notification". When such a message arrives in the host memory at its destination node, an interrupt will be generated. If this interrupt, or notification, is not blocked, it can get through immediately and a high-level handler is activated to deal with the message. If notifications are blocked—either because the host is activated in a notification handler already, or because notification is intentionally blocked for mutual exclusion in protocol operations—the handler execution is postponed until notifications are unblocked.

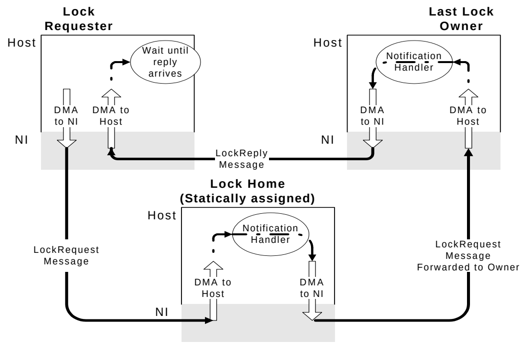

Figure 4: Remote lock acquire operation.

Remote lock acquire: Each lock has a home node statically assigned. In the most general case, a remote lock acquire is composed of 3 messages: (i) a message sent from the lock requester to the lock home, (ii) a message in which the lock home forwards the request to the last lock owner, and (iii) one in which the last lock owner grants the lock to the requester after it releases the lock. These are shown as thick arrows in Figure 4. The data flows inside each node are identical to those in Figure 3. The "last lock owner" represents the last node that requested the lock right before the current requester. It is not guaranteed that the last lock owner is holding the lock when the current lock request message arrives at it—it may be waiting for the lock to be granted from its last lock owner at the moment. We will discuss this situation in detail in Section 4.2. The first two messages, the initial lock request and the home's request forwarding, are both sent with notifications. Meanwhile, the lock requester waits for arrival of the granting message. Sometimes when the requester and the lock home are identical, or the lock home is also the last lock owner, this 3-hop process turns into 2 hops.

3.2.2 Our SVM Monitoring Approach

With the preceding section as background, we introduce how the performance monitor is used to diagnose performance issues for the SVM system in this section.

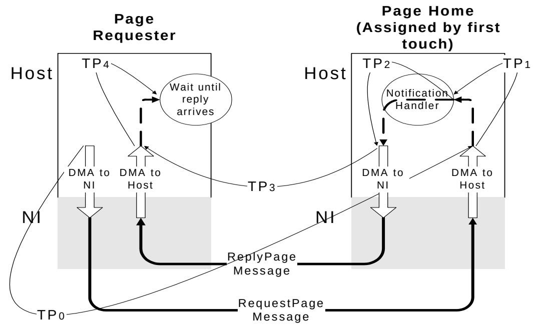

Figure 5: A remote page fetch operation traveling across the communication layer. The cycle is broken into 5 time segments.

Figure 5 breaks the whole cycle of remote page fetch—2 node-to-node hops—into 5 time segments, from the performance monitor's point of view. In Figure 5 each thin arrow labeled by TP defines a time segment. TP_0 defines the period from the time the requesting host issues the request message until the time when the request message reaches the page home's host memory. TP_1 is the time from when the request message gets to the page home's host memory until the notification handler starts to execute. TP_2 is from the point the notification handler is activated to the point the page home sends the page reply message back to the requester. TP_3 is like TP_0 except the starting and ending nodes are reversed. TP_4 is from the point the page reply message reaches the requester's host memory to the point the requester finally sees the page.

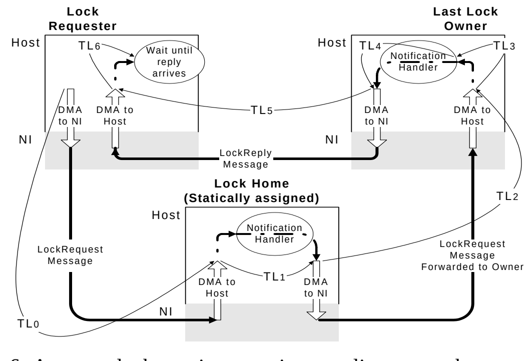

Figure 6: A remote lock acquire operation traveling across the communication layer. The cycle is broken into 7 time segments.

Similar to remote page fetch, remote lock acquire can be illustrated from TL_0 to TL_6, as shown in Figure 6. By breaking the time SVM operations spent at the communication layer into detailed pieces, we are able to separate performance characteristics related to different paths, and we can thus see whether the network latency, or the time blocked at notification (interrupt), or the period spent in the notification handler, dominates the overhead. Furthermore, some of the performance issues that are very hard to detect elsewhere are now possible to illustrate using the rich set of statistics from the performance monitor. Examples discussed in Section 4 include the serialization effect on some lock-based synchronization and contention problems in the system or application.

The performance monitor and the SVM system both hold some information that is not visible to the other. The performance monitor has no idea which messages belong to the same operation, nor when the notification handler will start to execute. The SVM system does not know how long the messages spend in crossing the network or when the message arrives at the host memory. A key issue in coordinating the performance monitor and the SVM system is designing a clean interface for them to exchange information. Our solution is very simple, consisting of a unique identifier and two functions.

To inform the performance monitor of the time pieces in one SVM operation described above, we assign a system-wide unique sequence identifier to each operation, which is carried by all messages within the operation when traveling through the communication layer. In this way, in MCP firmware residing in the Myrinet card, the performance monitor checks this identifier and recognizes messages of the same operation. For example, consider a remote lock acquire. When a message gets into host memory, the performance monitor on that node records the message arrival time associated with its identifier. When the node is able to send out a message which carries its identifier, the performance monitor recognizes the identifier and retrieves the arrival time associated with it. With arrival and departure time at hand, now we can get TL_1 and $TL_3 + TL_4$. Time pieces spent on the network between hosts are easy to get and only involve the performance monitor. As the monitor uses the global synchronized clock previously described, each outgoing message can carry a departure time-stamp so that the time this message spent traveling through network can be used for latency calculations at the receiving side. These include TL_0, TL_2 and TL_5. TL_4 and TL_6 are more difficult because they involve activities that are only visible to the SVM system, which reflect the time the notification handler starts to execute and the time the host finds out the reply has arrived. So we provide a simple and efficient interface from the performance monitor for the SVM software. GetCurrentRTC() returns the current time and GetReqFinishTime(identifier) returns the arrival time of a message into the host memory. It therefore becomes easy for the SVM software to calculate and store TL_4 and TL_6.

In addition to the unique operation identifier, we also record the time-stamps when each notification happens. In this way, we can build an operation log to show not only the relation between messages within the same operation, but also the mutual effects among different operation messages.

Up to this point, we have argued that the performance monitor on the Myrinet-based PC cluster can be used efficiently and easily for the SVM system to debug performance. Meanwhile a general issue to consider is to what extent the performance monitor perturbs the system it is trying to measure. The good news is that perturbation from our performance monitor is quite small: less than 2.5% for each of the SVM applications. The perturbations of the TPs and TLs are also small, only several microseconds, as there are only a few lines of code inserted into the sending and receiving path.

4 Monitoring SVM Performance

In this section, we present analyses of application performance on the SVM system. In particular, we use the performance monitor to consider the interactions of SVM with the communication architecture of the Myrinet-based PC cluster. Interesting operations that provoke these interactions include remote page fetch for coherence and synchronization. This performance study will particularly touch on the bottlenecks illustrated in Figure 2: data waiting time, lock time and barrier time.

4.1 Remote Page Fetch

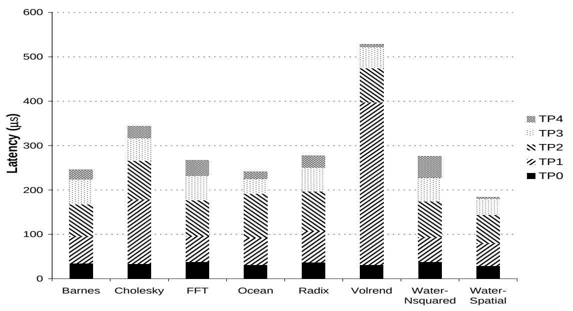

Figure 7: Breakdown of average page acquire latencies across applications. Average is total time spent on remote page fetch on all nodes, over total page-fetch acquires.

Figure 2, profiled by the SVM system, shows that the data waiting time for remote page fetch operations is quite high for some applications, such as Cholesky, FFT, Radix and Volrend. As shown in Figure 5, the performance monitor in the network interface can break the overhead of page fetch operations into more detailed components as shown in Figure 7. It illustrates that time spent on the network (TP_0, TP_3) is quite consistent and small. Time spent in the notification handler to deal with a page fetch request on the home node and send out this page (TP_2) is also quite consistent at about 70μs. However, TP_1, time spent waiting for a request to be handled (by an interrupt) is very high in Cholesky and Volrend.

Event Arrival Time (μs)	Time Waiting For Interrupt (μs)	Handler Execution Time (μs)
x	168.0	63.0
$x + 16$	287.5	71.5
$x + 40$	361.0	64.0
$x + 56$	425.0	64.5
$x + 80$	489.5	64.0
$x + 104$	553.0	60.0

Table 2: Queuing effect of interrupts for remote page fetches in Volrend.

By sorting the arrival time of each request on certain nodes (i.e., the starting point of TP_1) we found that the high TP_1 in Cholesky and Volrend is caused by a queuing effect in waiting for notifications attached to the page acquires to be activated. In our current system, a notification cannot be delivered when the node is already active in a notification handler. The upcoming notifications have to be queued and served in a FIFO manner. Put another way, it is the situation that many requests rush into the same home at almost the same time and have to wait for an interrupt, which may take a long time due to the serialization. Using data from the performance monitor, Table 2 shows a snapshot of this queuing effect on interrupts in Volrend's page fetch operations. We can clearly see that the handler execution time is short and almost constant. In contrast, the interrupt waiting time grows from 168.0μs to 553.0μs from one request to the next in a queue.

Unlike Cholesky and Volrend, FFT and Radix, whose data waiting times are also very high (see Figure 2), do not show high notification blocking time in Figure 7. To explore the possible amortization effect of averaging in Figure 7, Figure 8 gives the breakdown across nodes for FFT, which illustrates that the distribution of different time pieces is balanced across nodes. Thus, there seems to be no particular bottleneck. However, compared to a microbenchmark measurement of "ideal" remote page fetch overhead, the over-

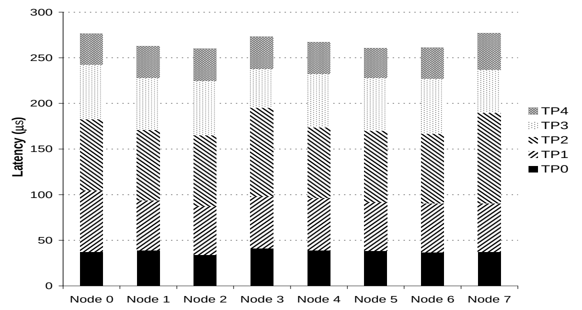

Figure 8: Per-node overhead breakdown of average page fetches in FFT.

head of such an operation in FFT is 50% higher. In particular, time spent on the network to transfer a page from the home to the requester (TP_3) in FFT is much higher: more than 2.5 times higher than that for the ideal case. This fact tells us that contention in the network and network interface is a problem for FFT, and this contention arising from all-to-all communication is quite uniform across links between different nodes. Experiments for Radix indicates that contention in the network and network interface is also a severe problem. However, we find that this contention is quite imbalanced across links for different nodes. In particular, deeper monitoring shows that Radix's permutation phase, in which scattered writes are carried out within the second and third loops, is the substantial bottleneck in the application.

To summarize, the performance monitor reveals that interrupt overhead in remote page fetch is high, and even dominates in some cases, such as Volrend. The queuing effect on interrupt waits relates to the strategy used to assign page homes. If the home distribution is not fair enough, a popular home becomes a bottleneck that causes contention in the host. For applications with all-to-all communication patterns, FFT and Radix for example, contention in network and network interface is clearly a problem. Node-to-network bandwidth is the bottleneck.

4.2 Lock-based Synchronization

In SVM systems, lock-based synchronization is expensive due to the inherent cost of the messages and the fact that coherence protocol activity is required at synchronization points. For applications like Barnes, Cholesky and Volrend (Figure 2) in which locks are extensively used, time spent on locks is substantial. But is it due to the overhead of synchronization or due to serialization of lock acquire requests because of dilated critical sections?

4.2.1 Components of Lock Overhead

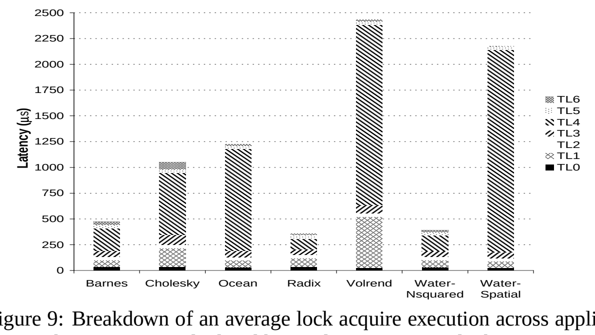

Figure 9: Breakdown of an average lock acquire execution across applications. The average is calculated by total time spent on lock acquire operations on all nodes over the total number of lock acquire operations.

By breaking the time spent on lock acquire into the 7 segments shown in Figure 6, Figure 9 exhibits the breakdowns of an average lock acquire execution across applications. From Figure 9 we can see that the time spent for messages traveling through the network for a lock acquire (TL_0, TL_2, TL_5) is quite consistent across different applications and relatively small (about 35μs). In contrast, time spent for handling the SVM protocol inside the host is much higher, including TL_1 and TL_3 + TL_4.

TL_1 is especially high for Cholesky and Volrend. Like TP_1 in a remote page fetch execution, TL_1 is mainly the overhead in waiting for an interrupt. This contention is particularly a problem for Cholesky and Volrend because they use task queues with task redistribution and task stealing. When task stealing or redistribution happens, it is likely that multiple processors try to enter a single critical section (a task queue) at the same time. On the other hand, TL_3 is less a problem, because every time an acquire of the same lock is forwarded, the last owner changes to the current requesting node, with the result that the burst at the lock home node is distributed into different nodes.

We also find that TL_4 is extremely high for most of our benchmarks. TL_4 represents the time that the last lock owner starts processing the lock request until it is finally able to release the lock to the next requesting node. Within TL_4, the last lock owner sends out the lock immediately if it is already released. It may have to postpone the send until the release operation and related coherence activities are done. In the worst case, it may not get the lock yet so it has to wait for the lock to arrive before finishing work within the critical section, and finally be able to release the lock to the requester. Therefore, TL_4 spans a large range of time for different situations described above. Over the different applications, this overhead ranges from less than 100μs per average operation in Radix to about 1800μs in Volrend. Though lock cost is significant in the total execution time of Barnes, Cholesky and Volrend and TL_4 is the highest segment in the whole lock acquire time for all of them, Figure 9 indicates that TL_4 plays different roles among them. In Barnes, TL_4 occupies less than half of the total latency, while TL_4 significantly dominates in Cholesky and Volrend. Since Barnes uses many locks (i.e., thousands), Figure 9 implies that it is the high operation overhead in SVM that makes lock time a performance bottleneck in it. In Cholesky and Volrend, however, the issue is serializations on lock synchronization.

4.2.2 Lock Serialization

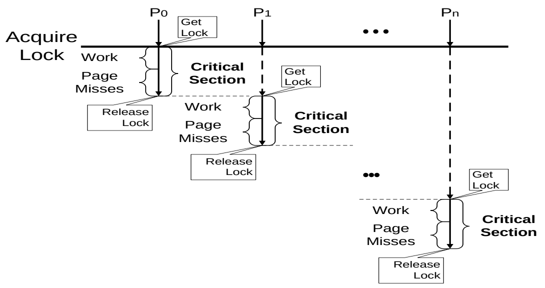

Figure 10: Lock serialization effects.

Before introducing our experiments, Figure 10 briefly describes how lock serializations occur. Because locks are expensive in SVM systems, this serialization effect hurts performance. Furthermore, page misses within a critical section can greatly exaggerate the critical section due to the overhead of remote page fetch operations, especially when contention happens during the remote requests. The dilated critical section thus makes the lock serialization even worse.

The performance monitor makes it possible to detect this lock serialization effect on real problems. To demonstrate what the performance monitor can show us in this effort, let us look at the time distribution the performance monitor generates for Ocean. Since Ocean has only a few coarse-grained locks, Figure 11 illustrates the lock serialization phenomenon in a particularly obvious way.

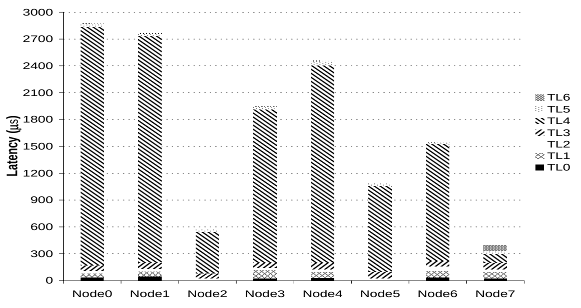

Figure 11: A lock serialization effect in Ocean.

TL_4 in this case ranges from about 100μs on node 7 (the first requester) to about 2700μs on node 1 (the last requester) in this chain for one lock acquisition.

Even though the lock-based synchronization is more complicated in Cholesky and Volrend, statistics generated by the performance monitor can still demonstrate that lock serialization is the main reason for high lock overheads. Table 3 gives one piece of the statistics data that exhibits the serialization on a lock in Volrend. Let us first focus on the first three rows in Table 3. When Node 1 is interrupted by Node 4 for a lock acquire at time $x + 9.0$, Node 1 has to wait for another 99μs ($108.0 - 9.0$) to catch the lock. Therefore Node 4 has to wait a long time (TL_4 = 915μs) for Node 1 to first get the lock, then enter the critical section, and finally release the lock. At time $x + 1461.0$, Node 4 is interrupted by Node 2 while Node 4 is likely still holding the lock (granted at time $x + 954.0$) and working in the critical section. Node 2 then has to wait until Node 4 releases the lock. As a result, TL_4 in Node 2's request time is still relatively long compared to that in the situation in which the lock is immediately served. The rest of the table can be explained in the same way. Then the question is why nodes spend so long a time in critical sections even though it is known that critical sections in Volrend are very small. Volrend uses a task queue and task stealing that features a migratory data referencing pattern. Analyzing statistics data provided by the performance monitor, we find that page misses occur quite often within critical sections. As we have mentioned earlier, page misses artificially dilate the critical sections, which increases serialization at the locks dramatically. Cholesky shows similar effects.

Requesting Node	Time Last Owner's Notification Handler Start to Execute (μs)	Time Lock Granted (μs)	TL_4 (μs)
Node 1	x	$x + 108.0$	75.5
Node 4	$x + 9.0$	$x + 954.0$	915.0
Node 2	$x + 1461.0$	$x + 1898.0$	389.0
Node 0	$x + 1628.0$	$x + 2810.0$	1155.0
Node 3	$x + 2665.5$	$x + 3209.0$	518.0
Node 1	$x + 2688.5$	$x + 3899.0$	1203.0

Table 3: A lock serialization effect in Volrend.

To summarize, the performance monitor demonstrates two major causes for high synchronization overhead on locks: waiting for interrupts, and serializations for lock acquires. The former is more communication architecture oriented while the latter is related to both the communication architecture and the SVM protocol. These two lock performance issues were previously known, but no prior efforts have successfully illustrated them with real numbers.

4.3 Barrier-based Synchronization

Like lock operations, barrier synchronization incurs coherence protocol activities. Unlike locks, however, barriers are a global synchronization. This global characteristic makes it an even more expensive operation in the SVM system. Our SVM protocol implements a simple n^2 barrier algorithm, in which each node has to communicate with all the others to keep coherent. In our benchmark suite, Ocean and Radix are clearly sensitive to barrier overhead, as shown in Figure 2. Barrier overhead is high in Ocean simply because Ocean uses a large number of barriers (98 in our experiment) between series of time steps. Radix is a more interesting example, however, because the reason for its barrier overhead is not obvious, and performance monitoring greatly helps.

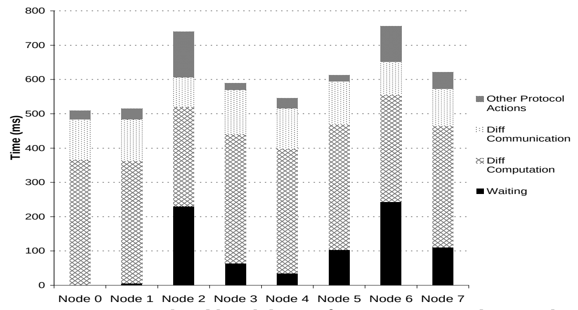

Figure 12: Barrier overhead breakdown after permutation phase in the second loop in Radix. Waiting time: time spent in waiting for coherence information from other nodes. Diff Computation time: time spent on diff computation for pages that have been written on this node before the barrier. Diff Communication time: time spent to send the diffs to the homes of the corresponding pages. Other Protocol Action contains protocol overhead to send and process data of bins, time-stamps, write-notices, etc.

Even though the total barrier overhead appears to be almost identical across different nodes in Radix, deeper performance monitoring finds that it is actually quite imbalanced for particular bottleneck phases in different loops. One example is shown in Figure 12. We have several important observations from this figure. First, waiting time is high and extremely imbalanced. However, the number of pages fetched by each node is quite balanced before this barrier. This indicates that the contention problem due to high data and control traffic arises from scattered writes to remote data in the permutation phase. This demonstrates how expenses from other operations can accrue to barriers. Second, time spent in *diff* computation is surprisingly high because the number of pages that have been updated is large due to scattered writes. This raises questions regarding the *diff* operation cost. Third, *diff* communication overhead is high compared to microbenchmark numbers, indicating that contention is also a problem, and that the home node is a bottleneck. Fourth, time spent on other protocol actions, mostly communication time, is also high and imbalanced. Given that the number of control messages from each node are almost the same, contention of control messages is a problem too. This points out the need for a better barrier algorithm that decreases control messages. In response to these measurements, we have implemented a centralized barrier algorithm and are now working to characterize its performance.

5 Guidelines

Incorporating our firmware-based performance monitor with the SVM system plays an essential role in debugging SVM performance on the Myrinet-based PC cluster. Going into successively deeper details of performance characteristics, from the SVM protocol layer to the base Myrinet communication layer, points us to insights on a range of SVM performance issues.

5.1 Communication Layer

As processor speeds keep increasing, it is important to provide high bandwidth both within compute nodes and across the network. This is demonstrated by the network and network interface contention seen when running applications with bursty and all-to-all communication patterns on SVM (barrier and remote page fetch in Radix, remote page fetch in FFT). Although fast system interconnects are available, low level communication libraries often fail to approach raw hardware performance. Therefore more efficient low-level communication interfaces may be helpful in providing low-cost, high-performance SVM systems.

The performance monitor also demonstrates that interrupt overheads usually stand in the way of good performance on the SVM system. Reducing interrupt overhead at the system level would help to improve SVM performance. Another solution is to remove interrupts by using a "smart" network interface—programmable by customers on Myrinet—instead of the processor in the host, to deal with corresponding SVM protocol operations.

5.2 SVM Protocol Layer

In studying bottlenecks in synchronization and remote page fetch operations in the SVM system, the performance monitor reveals that system contention can be high. The problem of contention in remote page fetch stems from the home assignment strategy the SVM protocol applies. While it is difficult to adopt a general and fair scheme to distribute the homes for pages, the performance monitor makes a better strategy possible on real systems. Combining the SVM system with the performance monitor, it is now possible for the SVM system to adjust the home distribution in real-time whenever contention is detected by the performance monitor. We are currently working on such an approach.

Lock serialization effects are usually substantial in applications with migratory data referencing patterns, which may cause frequent page misses within the critical section. Thus critical sections are artificially dilated, which causes subsequent lock acquires to be serialized. The key to eliminating this effect is then to decrease data migration in critical sections. One solution is to choose homes of locks dynamically rather than assigning them statically. Another solution is to return the lock to its home each time the lock is released (cheap with a "smart" network interface on Myrinet).

5.3 Application Layer

A previous simulation study [14] shows that understanding how an application behaves and restructuring it properly can dramatically improve performance on SVM systems. This however is not always easy, especially on real systems. Insufficient tools are available in parallel systems to help discover the causes of bottlenecks and obtain insight about application restructuring needs, especially when contention is a major problem, as it often is in commodity-based communication architectures. Our firmware-based performance monitor on the Myrinet-based PC cluster eases the task of understanding and restructuring applications for better SVM performance. In particular, by successively deeper detections, the monitor helps the programmers spot the sources of the performance bottlenecks.

6 Related Work

This research builds on prior work on performance tools and SVM systems. Our performance monitoring approach is similar to techniques from the FlashPoint system [20]. That performance monitoring system embedded extra monitoring code into the cache coherence protocol handlers running on the FLASH multiprocessor's MAGIC chip. Our Myrinet-based PC cluster is expected to be used on a much wider variety of programs than FLASH: both shared-memory programs running via SVM and also message passing programs.

Our SVM system implements an HLRC protocol on a Myrinet-based PC cluster. The same HLRC protocol was previously implemented on Intel-Paragon [24], Wisconsin Typhoon-Zero [10] and SHRIMP [5] (a PC-based cluster interconnected with a custom designed network interface). The HLRC implementation on Typhoon-zero used active messages for communication support. Our HLRC implementation uses the VMMC-2 library [9] and in this sense is similar to the HLRC implementation on SHRIMP [11] which also uses a memory-mapped communication library.

SVM performance has been studied, both in simulations [3, 13, 14, 17] and on real systems [1, 15, 16]. But our work is the first, to our knowledge, in incorporating the SVM system and a performance debugging tool embedded into Myrinet firmware for performance study on a real system. Also, our work is different from other performance evaluations of real (not simulated) Myrinet-based systems, which have relied on microbenchmarks and higher-level system measurements [8, 21, 22]. This limits their ability to make lower-level determinations of how different components of communication impact performance.

7 Conclusions

This paper has presented an empirical discussion of an HLRC-based SVM system and a software performance monitor residing in low-level Myrinet firmware. Our hierarchical approach for performance analysis on a real system allows us to diagnose SVM performance characteristics in deeper detail than previously available.

In studying applications with a range of behaviors, especially those with poor SVM performance, the performance monitor helps discover the causes of bottlenecks particularly when contention is a major problem, as it often is in commodity-based communication architectures. The performance monitor was designed to work well with a wide range of software, including both message passing and shared-memory programs. The simple and clean interface eases the process of interfacing the SVM system (or other high-level software) with the performance monitor.

In measuring the real SVM system, we detected a number of interesting performance effects. We observed lock serializations arising from dilated critical sections and contention on interrupts waiting to be activated in the hosts. These observations reinforce previous thoughts on the importance of properly assigning homes, either for data pages or for locks, in home-based SVM protocols. Fair distributions of homes for data pages is the key to solve the contention problem in hosts. To eliminate lock serializations, it is crucial to allocate lock homes dynamically, so that the home of a lock is also the home of the data the lock protects.

Overall, our research provides a combination of real system measurements and detailed monitoring methodology that should effectively complement the body of high-level measurements and simulation data currently available for SVM researchers.

Acknowledgments

This work was supported in part by DARPA contract N000014-95-1-1144, NSF grant MIP-9420653, NSF Career Award CCR-95-02516 (Martonosi), and Intel Corporation. We thank the referees for their helpful suggestions.

References

[1] R. Bianchini, L. Kontothanassis, R. Pinto, et al. Hiding communication latency and coherence overhead in software DSMs. In *The 7th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[2] A. Bilas, L. Iftode, R. Samanta, and J. P. Singh. Supporting a coherent shared address space across SMP nodes: An application-driven investigation. *IMA Volumes in Mathematics and its Applications*, 1998.

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

[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 *Proc. of the 21st Annual Symposium on Computer Architecture*, Apr. 1994.

[5] M. A. Blumrich, R. D. Alpert, Y. Chen, et al. Design choices in the SHRIMP system: An empirical study. In *Proc. of the 25th Annual Symposium on Computer Architecture*, June 1998.

[6] N. J. Boden, D. Cohen, R. E. Felderman, et al. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[7] F. Christian. Probabilistic Clock Synchronization. *Distributed Computing*, vol 3:146–158, 1989.

[8] D. Culler et al. Assessing fast network interfaces. *IEEE Micro*, Feb. 1996.

[9] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, Aug. 1997.

[10] M. D. Hill, Y. Zhou, I. Schoinas, et al. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. Technical Report TR-535-96, Department of Computer Science, Princeton University, Dec. 1996.

[11] L. Iftode, M. Blumrich, C. Dubnicki, et al. Shared Virtual Memory with Automatic Update Support. Technical Report TR-575-98, Department of Computer Science, Princeton University, 1998.

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

[13] L. Iftode, J. P. Singh, and K. Li. Understanding the performance of shared virtual memory from an applications perspective. In *Proc. of the 23rd Annual Symposium on Computer Architecture*, May 1996.

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

[15] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter USENIX Conference*, Jan. 1994.

[16] L. Kontothanassis, G. Hunt, R. Stets, et al. VM-based shared memory on low-latency, remote-memory-access networks. In *Proc. of the 24th Annual Symposium on Computer Architecture*, June 1997.

[17] L. Kontothanassis and M. Scott. Using memory-mapped network interfaces to improve the performance of distributed shared memory. In *The 2nd Symposium on High-Performance Computer Architecture*, Feb. 1996.

[18] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proc. of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, Aug. 1986.

[19] C. Liao, M. Martonosi, and D. W. Clark. Performance monitoring in a Myrinet-connected SHRIMP cluster. In *Proc. of 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, Aug. 1998. To appear.

[20] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers. In *Proc. ACM SIGMETRICS Conf. on Meas. and Modeling of Computer Systems*, May 1996.

[21] S. Pakin et al. Fast messages: efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, Apr. 1997.

[22] T. von Eicken, D. Culler, et al. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Annual Intl. Symposium on Computer Architecture*, May 1992.

[23] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proc. of the 23rd Annual Symposium on Computer Architecture*, May 1996.

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