Home-based SVM Protocols for SMP Clusters: Design, Implementation and Performance

Rudrajit Samanta, Liviu Iftode and Jaswinder Pal Singh
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

Shared virtual memory has been used as a cost-effective means of providing the attractive shared-memory model to programmers. In the past, most SVM implementations have been focussed on clusters of uniprocessor systems. Due to the increasing popularity and declining costs of small scale shared memory multiprocessor systems (SMPs), they have become attractive candidates for replacing the uniprocessor nodes when building a shared virtual memory layer. At present, there is no definite answer on what is the best approach to implementing an SVM layer for a cluster of SMPs, whether such a protocol is necessarily different from one designed for uniprocessor nodes, and what accepts of this difference actually results in increases in performance.

To answer these questions and explore the design space of SVM on SMPs, we have implemented a home-based lazy release consistency (HLRC) protocol on a cluster of 4-way Intel PentiumPro SMPs interconnected with Myrinet. This paper presents numerous protocol design options we were faced with and their implications. We are able to achieve very promising performance on some applications investigated so far. We have investigated (quantitatively) the tradeoffs in our system to validate some of the choices we have made. The tradeoff investigation is ongoing, and we will complete it for the final paper.

1 Introduction

In the last two years, following the rapid growth of the market, clusters of symmetric multiprocessor (SMP) systems have been emerged as the most attractive solution to produce affordable parallel systems. To provide a uniform programming interface on a cluster of SMPs, one approach is to develop a software shared memory layer across SMPs as an extension to the existing intra-node hardware cache-coherent shared memory.

Building an SVM across SMPs may look simple given the experience accumulated in developing SVMs on cluster of uniprocessors, and in fact this is true if we do not wish to take much advantage of the hardware coherence and synchronization within an SMP or to provide a high performance lazy protocol. A few researchers have developed particular implementations starting from the uniprocessor-node based protocols, and identified significant performance limitations such as TLB synchronization or limited network bandwidth [3].

In fact, there are several new issues to consider in designing a SVM protocol across SMPs, which are not present with uniprocessor nodes. These include:

- Whether to use a shared physical or separate physical address spaces for processes within the SMP
- The degree of laziness in the propagation of coherence within an SMP
- How synchronization should be performed within the SMP
- Should a processor(s) be reserved for protocol execution or should this be performed by the compute processors themselves
- The TLB synchronization problem

We discuss each of these issues and their implications systematically, as well as our choices and their relative merits. Our choices are different than those of previous implementations. The goals of in our protocol design has been to exploit the high performance intra-node shared-memory to the maximum possible extent. To make intra-node operations as cheap without affecting the inter-node coherence performance and to make coherence propagation within the SMP very lazy. We quantitatively compare the extreme points in this spectrum of choices and identify some important (easier) intermediate protocols along the way, to understand which choices and degrees of sophistication are most critical.

1.1 Related work

A few recent systems also implement SVM across multiprocessor nodes. The softFLASH system [3] which has goals similar to us, but uses a single-writer protocol (modeled after the FLASH protocol). The protocol is not very aggressive compared to some typical multiple-writer lazy SVM protocols and the system suffers from the problems of high latency and expensive TLB synchronization. The use of 16KB pages in the SoftFLASH system increases false-sharing and causes fragmentation resulting in a waste of bandwidth. Our protocol is rather different since we use a multiple-writer home-based protocol (HLRC) based on lazy release consistency. In our implementation the Myrinet network interface and network provide low latency. Also, due to nature of the architecture we use, the TLB synchronization problem is avoided. Our 4KB page size (as in Linux) is smaller and the bandwidth of the network interconnect (160MB/s) is also helpful in avoiding bottlenecks.

The Multigrain Shared Memory System (MGS) [8] built on top of the MIT Alewife machine uses a protocol very similar to the SoftFLASH system. The system is implemented by portioning the Alewife machine, so each node is a distributed rather than centralized shared memory multiprocessor, and the number of external network connections from the node (and hence bandwidth) scales with the number of processors in it. The processors in Alewife are are also very slow by todays standards.

The Shasta system [6] performs the coherence in software at cache-line rather than page-size granularity by instrumenting the executable to insert access control operations before shared loads and stores. This idea was first implemented in the Blizzard-S system [7]. There is currently an implementation of Shasta that runs on a cluster of Alpha SMP nodes. This implementation does show promising performance, but it is not page-based shared virtual memory. Also, it requires a tool for code instrumentation, which may not be readily available on all architectures. We have designed our control data structures so that their size depends on the number of nodes in the system, not the number of processors. Therefore we can scale the number of processors by increasing processors within a node without an increased memory requirement.

Simulation studies have been performed for extending SVM protocols (including LRC and AURC) to SMP nodes. The first of these [2] shows promising performance. both LRC and ARC protocols to SMP nodes. The LRC simulations study [4] has found that a reserved protocol processor in not required. The ARC simulation study [1] showed that memory contention due to write-though caches can prove to be a bottleneck. Suggestions are made for advanced hardware support to alleviate this problem. The implementation presented does not require any custom hardware support.

2 Uniprocessor Protocols

The basic protocol we have prototyped extends the home-based lazy release consistency (HLRC) protocol described in [9]. In this section we briefly review lazy release consistency (LRC) and then explain the HLRC protocol.

2.1 Lazy Release Consistency

Subsection omitted for extended abstract version. There are many descriptions in the literature such as [5]. The main idea in the protocols is to fully exploit the release consistency model, and propagate coherence operations to (uniprocessor) nodes only when those nodes perform an acquire synchronization operation. Multiple processes may have writeable copies of a page and these are reconciled at an acquire.

2.2 Home-based LRC Protocol

Subsection omitted for extend abstract. Please refer to [9]. Here the reconciliation of multiple writeable copies is done by having them all propagate their changes (detected in software by means of diffs) to the home for that page. The home is kept up to date at release operations and a process can obtain a current version of the page from the home after and acquire operation.

3 Design Issues

In this section we discuss the design space for a shared virtual memory protocol on a cluster of SMPs, starting from a uniprocessor-node protocol such as HLRC. Implementing the HLRC protocol on SMPs requires several non-trivial changes due to the interactions of hardware-coherent intra-node shared memory with the software-coherent inter-node sharing. For instance, even the operation of producing a diff is more complicated than in uniprocessor case. Since processes in an SMP share the same physical memory, a process may perform a diff on a page while other processes are writing into that page. To detect all the updates made to the page, the dirty copy which is being diffed becomes the twin (clean copy) of that page as we perform the diff. Also when fetching a full page one must careful to ensure there are no processes on this node that are currently writing to this page, otherwise the incoming page may overwrite those writes, which are then lost forever. To achieve this, no process is allowed to write on a page while a page fetch is in progress. In the next section we discuss some of the critical issues related to the efficiency of an SVM implementation for SMPs.

3.1 Application data sharing within the SMP

A shared-memory multiprocessor provides an efficient hardware-based shared memory at a cache line granularity. The software-based shared virtual memory protocol can utilize the efficient intra-node sharing in two ways. One approach is to use the shared-memory within the node only to emulate message passing between the processors. In this case the hardware shared memory is hidden in the communication layer and as a result, the protocol views the processors of the same SMP as separate, share-nothing nodes. Using this approach, an uniprocessor-based protocol can be ported to a cluster of SMP nodes with minimum modification. This approach obviously does not leverage the cache-coherent shared memory provided within the SMP except as an acceleration for message passing.

The alternative approach is to expose the intra-node sharing at the protocol level. This requires the protocol to provide a hybrid sharing scheme and distinguish inter-node from intra-node sharing. For inter-node sharing the software protocol is responsible for ensuring coherence using the virtual memory mechanism. For intra-node sharing the coherence is transparently provided by the hardware at cache-line granularity. Since the protocol is aware of the intra-node sharing, the inter-node coherence control can take advantage of it. For instance, all processes can benefit from a page fetch performed by one processor. In order to make this possible, processes must share the data structures describing the current status of the shared pages in the SMP memory. WE assume this approach in the rest of the discussion.

3.2 Coherence propagation within an SMP

On a cluster of SMPs, an all-software protocol is responsible for keeping shared memory coherent across nodes. Like most SVM implementations, we use the page protection mechanism provided by the virtual memory system. Within SMP nodes, coherence actions triggered by one processor in the node affects the others. We need to optimize these interactions. For example, if a process acquires a remote lock (which was released on a different node), when should we propagate the invalidations we receive to the other processes ? We can perform an *eager invalidation* scheme where we perform invalidation for all the processors at once (see Figure 1). This eager invalidation approach is easy to implement, but it may invalidate threads too early and expose irregular applications to greater false sharing than necessary. The eager invalidation scheme is exactly what is achieved when we use the concurrently running copies of the application on a node are treated as threads rather than processes.

On the other hand we can explore the possibility of propagating the invalidations lazily. When a process acquires a lock held by a remote node, only this process performs the invalidations necessary. When other processes on this node subsequently request the same lock only then are they required to perform the invalidation (see Figure 2). The lazy approach requires a more elaborate protocol which has to distinguish the coherence state of each process within the same SMP which require separate page tables etc. However, this scheme may save a large number of page invalidations (which are expensive operations). This lazy inter-node coherence scheme delays some invalidations which may reduce the false sharing effect that occurs with the eager invalidation scheme.

3.3 Intra-Node Synchronization

Another protocol design issue which can impact performance substantially is the implementation of synchronization primitives. As in any shared virtual memory implementation, inter-node synchronization is implemented using explicit message passing. Giving the hardware intra-node shared memory support, a desirable approach is to implement the intra-node synchronization as a local operation without interrupting the process which has released the lock.

With an eager invalidation scheme for the inter-node lock acquire, the intra-node acquire and release operations are totally relieved from any coherence action. This is possible because at all times, all processes (or threads) share the same state (pages that are invalid, etc.), i.e. the protocol actions needed were already performed at the last *inter*-node synchronization.

When lazy invalidations are used for inter-node coherence, invalidations are performed at all (both intra-node and inter-node) acquire operations as required to maintain the causal order imposed by the release consistency model. Since the inter-node coherence control data is kept in shared memory, an intra-node acquire is performed exclusively by the process that requested the lock. Therefore, it is a local operation which does not involve interrupting other processors, although it does invalidate local pages (unlike the previous case). It is not clear which approach is better in practice. While eager invalidations increase false sharing and make inter-node synchronization more expensive, lazy invalidations increase intra-node synchronization time.

3.4 TLB Coherence

Previous studies have discussed TLB synchronization as a major obstacle for an SVM implementation to achieve good performance on a cluster of SMPs. TLB synchronization or TLB shootdown are terms used for the global operation (within an SMP) of flushing the TLB of all the processors of the same SMP.

During the executing of an SVM system, since we use the virtual memory mechanism to detect reads and writes, a large number of changes in the protection bits of pages are required. Hence we perform a large number of changes to the page table. Whenever a change in the page table is performed, one is required to flush the TLB to ensure that this change is made visible to the processor. This is a simple operation on a uniprocessor but becomes complex with the introduction of multiprocessor systems.

Many current processors (such as the Sun Sparc, MIPS R10000 and Digital Alpha) allow TLB entries from multiple address spaces to coexist, as a consequence these processors do not perform TLB flushes on context switches. Since processes may migrate from one processor to another, when a page table entry is modified all the TLBs within the node require a TLB flush operation. When attempting to flush all the TLBs in the system we have to signal other processors and wait for their flush operations to complete, this can prove to be quite expensive, especially as the number of processors within a node increases.

For processors in the Intel Pentium family the TLB can hold entries only for a single process, hence TLBs are flushed on every context switch. This means that these processors do not require the TLB synchronization operation described above. This solution (flushing on every context switch) could also be used instead of the TLB synchronization for the other processors but this would hurt their performance significantly.

3.5 Protocol handling

In all SVM implementations remote requests sent over a network need to be serviced. On a uniprocessor node there is really no choice in this regard : the processor must be interrupted (unless it polls periodically). However, with SMP nodes we have a number of choices. The two basic ideas are to either dedicate a processor within the SMP to handle network requests exclusively (by polling) or to handle the requests by interrupting one of the computation processors.

A dedicated processor implementation helps to avoid interrupts, which are known to be a performance bottleneck in most systems. However this choice wastes a valuable compute resource. We do notice that this dedicated processor does spend a large portion of it's time idle, since actual protocol processing overhead still is not very high.

If we use the compute processors to handle requests then we could either statically assign one compute processor for this purpose or we could perform a round-robin assignment

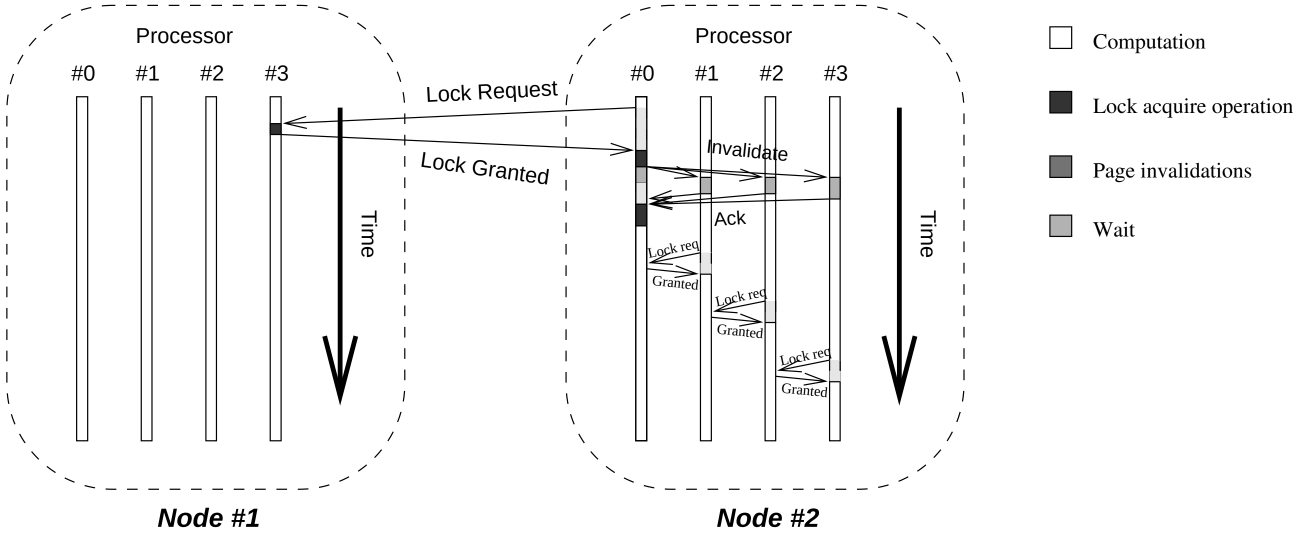

Figure 1: Eager invalidation scheme

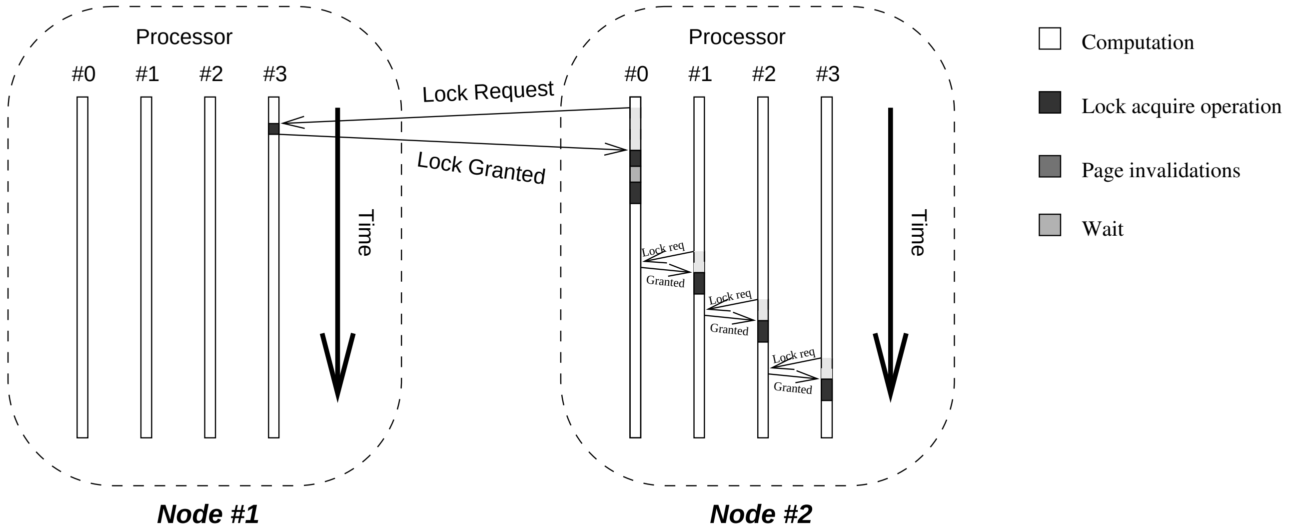

Figure 2: Lazy invalidation scheme

as the requests arrive. To reduce interrupts we can instruct idle compute processors to poll for requests and interrupt a random compute processor only when there are no idle processors.

It is interesting to note that each solution presented for protocol handling is expected to perform better but is more complex than the previous one. Also, some choices may be difficult to perform due to architectural limitations. For instance, Linux 2.0.x sends interrupts only to processor 0, and it is not possible to distribute interrupts among processors within an SMP.

4 Protocol outline

Now that we have presented the various choices available, we will now describe the specific choices made in our implementation. The data structures and schemes used will also be described in some detail.

To utilize the shared memory within the SMP to the maximum extent possible, we implement a protocol where all the application data for processes within a node is kept coherent by the hardware. We investigated the use of a dedicated protocol processor and determined that the protocol processing overhead is rather low and hence dedicating a processor is a waste of resources. Instead we interrupt a compute process for protocol handling. Finally, we use a lazy invalidation propagation scheme to reduce false-sharing to a minimum.

4.1 Operations and Data structures

To illustrate the data structures used we first need to define some key terms. The time during the actual execution of a parallel program is broken into *intervals*. With uniprocessor nodes, intervals are maintained on a per-process basis. An interval is the time between two consecutive releases by the process. These intervals are numbered in a monotonically increasing sequence. Each process maintains a vector called the *update-list*, which records all the pages that have been modified (by this process) in the current interval.

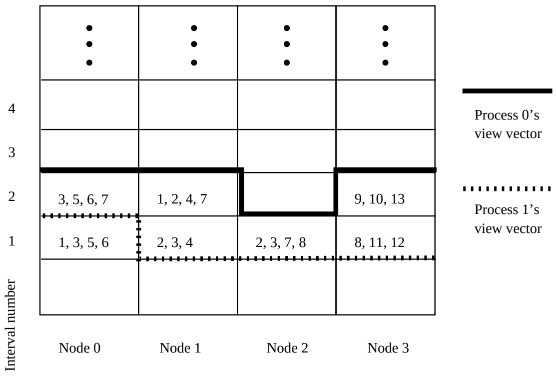

Figure 3: The bins data structure

When we end an interval this update-list is placed in a data structure called the *bins* (see Figure 3). In the SMP protocol we use a single column for each node and not for each processor. Intervals are therefore maintained per node rather than per process. This ensures that our data structure scales with the number of nodes as opposed to the number of processors, providing greater scalability than schemes whose data structures are proportional in size to the number of nodes.

Each node has a single copy of the bins. In each column of the bin structure we have an entry for each time interval. The entry for node i and interval j is the list of pages modified by node i during interval j. The bins are not only updated locally at a release, but also during remote synchronization operations. In the latter case, a remote node sends over all the entries (in the form of *write notices*) that it has in it's bins data structure but the local node is lacking. It is important to realize that these bins merely contain information about pages modified by the processes in the system, it does not imply that these pages have been invalidated at our local node.

To manage invalidations and determine what din data to transfer, there is one more crucial element which we refer to as the *view vector*. Essentially this is the "view of the world" that a particular process (not node) has. In terms of our data structures, this vector tell us what portion of the bins have been seen (i.e. the corresponding invalidations have been performed) by this process. This vector is the key element that tells us how much invalidation information has been acted on by a process. While bins are maintained on a per-node basis, view vectors are maintained on a per-process basis; this is what allows us to implement a lazy invalidation scheme. Let us look at some examples of lock acquisitions to see how this works.

During a remote lock acquire operation (when the requested lock is available at a remote node), the requester sends over the view vector and a vector that indicates what bins are present at this node. After this write notices are sent along with the view vector of the releasers of the lock. At this point the requester matches it's view vector with the lock releaser's view vector (at the time of the lock release

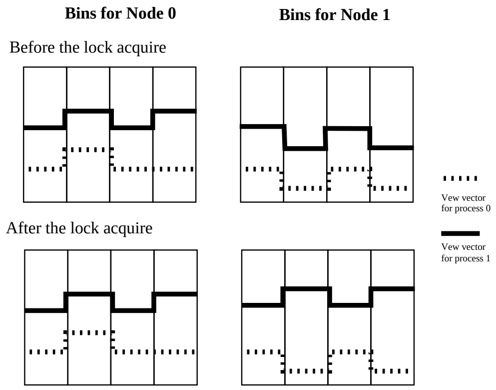

Figure 4: Remote lock acquire: process 1 on node 1 acquires a lock previously released by process 1 on node 0

operation). Matching of view vectors involves invalidating (locally) all the pages indicated in the bins that are "seen" by one view vector but not the other. This operation is illustrated in Figure 4. In this example process 1 on node 1 is acquiring a lock which was previously released by process 1 on node 0.

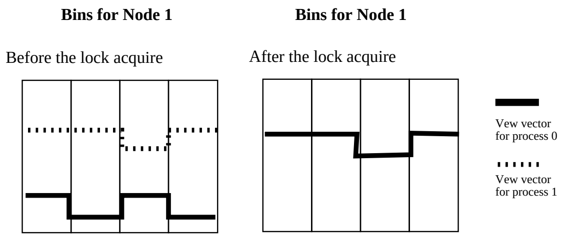

Figure 5: Local lock acquire: process 0 acquires a lock released by process 1

Our scheme allows a local lock acquire operation to be completely local with no external or internal messages. All that is required is the matching of the requesters view vector with the releaser, hence there is a very small amount of protocol processing. As we can see in Figure 5, we only require bins available at our node to perform this operation, no data transfer is involved.

For a barrier operation all the bins generated at this node which have not been propagated are sent to the barrier manager. The barrier manager then disperses this information to all the nodes in the system. At this time all processes in the cluster match their view vectors to one that includes all the available bins, invalidating all the pages necessary.

The mechanism that we have yet to explain is : once we do fault on an invalid page how do we retrieve a current version of this page ? Since this is a home-based scheme we know that we can fetch a current version from the home of the page. However, we do not wish to fetch a version that is more recent than necessary. To avoid this problem we utilize a system of *lock timestamps* and *flush timestamps*. Each page has associated with it a flush timstamps that indicates what is the latest interval (at each node) for which the updates are currently available. One may think of this as a "version" of the page. The lock timestamp indicates what the flush timestamp of the page should be in order to ensure we see the relevant updates by other processes. Basically this is the version of the page we should have. This is the vector that is sent over when we request a page from it's home.

5 Prototype implementation

5.1 Hardware Platform

The nodes used in our system are Intel SMPs with 4 PentiumPro CPUS with a clock speed of 200 Mhz. The PentiumPro processor has a 8KB Data and an 8KB Instruction cache. The processors are equipped with a 512KB unified L2 cache and 256MB of main memory.

There are currently 4 nodes in the configuration which are connected by a Myrinet network. The Myrinet network used for the system provides a peak bandwidth of 160MB/s per channel. The network cards have a programmable chip (the LANai v4.1) which facilitates the implementation of efficient user-level communication libraries. All systems are connected directly to an 8-way switch.

5.2 Basic Software

To control the communication hardware in the system we use an implementation of the Shrimp Basic Library (SBL) [xx] on the Myrinet network. This general-purpose, memory-mapped communication layer provides a minimum one-way latency of approximately 15 μs and a peak bandwidth of about 90MB/s. The operations used are send, receive (asynchronous) and some setup routines. No buffering is assumed since the SBL layer does not provide any. The SBL API allows regions of memory to be exported to other nodes. These nodes then *import* these buffers and receive a *proxy address*. The node can then send data by simply passing a pointer to some data and the proxy address they wish to send that data to (along with the size of the data, of course). The process receiving the data has to simple check a flag in the buffer that it has exported to check if data is available.

The SMP nodes run the SMP version of the Linux operating system (version 2.0.24). There are two slight modifications made to the Linux kernel to provide better performance for certain operations.

In order to detect reads and writes to a certain page in the shared address space we use the `mprotect` function call. In Linux the `mprotect` call requires a system call invocation (12 μs on the average) each time. In order to speed this up we implemented a modified version of the call which allows to specify an entire array of pages that are to be `mprotect`-ed at once. This call avoids multiple system calls.

The SMP support provided in the Linux kernel attempts to hide the SMP nature of the system from both the user and programmer. Unfortunately interrupts are sent only to processor 0. In order to provide better performance for our system, we added a simple call to the kernel to allow us to pin the process which is designated to handle interrupts to processor 0.

5.3 Basic costs

The various costs of basic operations in our environment (Linux) are shown in Table 1.

Operation	Cost (μ-secs)
mprotect (original)	12
mprotect (optimized)	6
Read fault (null handler)	20
Write fault (null handler)	20
bcopy (4KB)	~150

Table 1: Some basic costs

6 Preliminary results

In this section we present performance results for the home-based protocol we have implemented on a cluster of 4 quad Intel PentiumPro SMPs. For this purpose we choose 3 applications from Splash-2 suite (LU, Water- Spatial and Raytrace). Speedups reported in Table 2 correspond to three configurations. 4x4 is the largest configuration: 4 nodes each containing 4 processors. 2x4 and 4x2 are two 8-processor configuration 2 node of 4 processor and 4 node of 2 processor, respectively.

Application	4 x 4	4 x 2	2 x4
LU (2048)	7.94	4.55	4.7
WAter-Spatial (4096)	10.99	6.34	6.31
Raytrace (balls4)	12.33	7.2	7.0

Table 2: Speedups for 3 applications

Overall, speedups are good. Except for LU which suffers from an inherent computation imbalance, the other two applications perform well. Given the high speed of the processor (200 MHz), the speedups we report show that our SVM protocol for SMP clusters is efficient. The results reported in the other two columns (2x4 and 4x2) are similar because the communication to computation ratio is rather low in these applications.

We should note that these results are preliminary and we intend to further analyze in detail the performance of our prototype using execution time breakdowns, detailed protocol statistics and and a larger set of applications.

7 Conclusion

The performance on our system (for 16 processors) is very promising and we hope that code optimizations and tuning of some expensive operations will help boost performance even further. One of the choices that we made but did not explore fully is the lazy invalidations scheme. We need to implement the eager scheme as well in order to quantitatively understand the tradeoffs. The choice to have processes within an SMP share allocation data was also not explored quantitatively, but is hard to imagine that our choice would perform worse than ones where no data was shared and we did not fully exploit the cache-coherent nature of the SMP. A larger number of applications would also help to explain system behavior in a more comprehensive manner and we are currently working on expanding the test suite of application used.

References

[1] A. Bilas, L. Iftode, D. Martin, and J.P. Singh. Shared Virtual Memory Across SMP Nodes Using Automatic Update: Protocols and Performance. Technical Report TR-517-96, Princeton, NJ, March 1996.

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

[3] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of The 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[4] M. Karlsson and P. Stenstrom. Performance Evaluation of Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *Proceedings of IEEE 2nd International Symposium on High-Performance Computer Architecture*, February 1996.

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

[6] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of The 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[7] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, and D.A. Wood. Fine-grain Access for Distributed Shared Memory. In *Proceedings of The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[8] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23nd Annual Symposium on Computer Architecture*, May 1996.

[9] 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 Implementat ion Symposium*, October 1996.