Evaluation of Hardware Write Propagation Support for Next-Generation Shared Virtual Memory Clusters

Angelos Bilas[1], Liviu Iftode[2], and Jaswinder Pal Singh[1]

[1] Department of Computer Science, Princeton University
Princeton, NJ 08544
[2] Department of Computer Science, Rutgers University
Piscataway, NJ 08855

{bilas, jps}@cs.princeton.edu, iftode@cs.rutgers.edu

Abstract

Clusters of symmetric multiprocessors (SMPs), connected by commodity system-area networks (SANs) and interfaces are fast being adopted as platforms for parallel computing. Page-grained shared virtual memory (SVM) is a popular way to support a coherent shared address space programming model on these clusters. Previous research has identified several key bottlenecks in the communication, protocol and application layers of a software SVM system that are not so significant in more mainstream, hardware-coherent multiprocessors. A key question for the communication layer is how much and what kind of hardware support is particularly valuable in improving the performance of such systems. This paper examines a popular form of hardware support—namely, support for automatic, hardware propagation of writes to remote memories—discussing new design issues and evaluating performance in the context of emerging clusters. Since much of the performance difference is due to differences in contention effects in various parts of the system, performance is examined through very detailed simulation, utilizing the deep visibility into the simulated system to analyze the causes of observed effects.

1 Introduction

The performance of an application running on a parallel system is affected by several layers of software and hardware. For page-grained shared virtual memory (SVM) on clusters, the layers are as shown in Figure 1 [26]. A key question for such systems is how much and what kind of limited hardware support is most effective in accelerating their performance, thus bringing it closer to that of hardware coherence. The goal is to make the shared address space model attractive for application users on clusters as well, thus making it competitive with message passing in performance portability across high-end multiprocessors and clusters. The most popular form of hardware support used so far is the propagation of fine-grained writes to remote memories [6, 18, 11]. This support inspired the design of a family of new, "home-based" protocols for page-based software shared virtual memory (SVM), which differ from earlier all-software protocols not only in hardware support but also in the manner in which they propagate changes and solve the multiple-writer problem (i.e., in the protocol layer). Essentially, every shared page has a *home* node, and writes observed to a page are propagated to the home at a fine granularity in hardware [13, 15, 21], without interrupting the processor at the home. In the automatic write propagation (also called automatic update or AU) approach, shared pages are mapped write-through in the caches so that writes can be snooped off the memory bus. When a node incurs a page fault, the page fault handler retrieves the page from the home where it is guaranteed to be up to date [13, 15]. Data are kept consistent according to a page-based software consistency protocol such as lazy release consistency [20]. Thus, consistency in maintained at page granularity, while there is some hardware support for fine-grained communication. The SHRIMP system [6] provides both the write snooping hardware as well as a customized network interface (NI) to support automatic update, and the Automatic Update Release Consistency (AURC) protocol has been implemented on it.

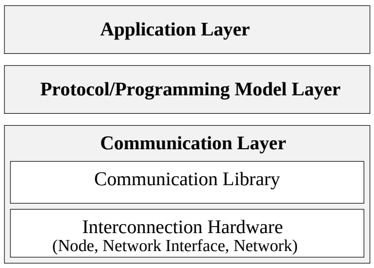

Figure 1: The layers that affect the end application performance in software shared memory.

Previous studies have shown these hardware-supported, home-based protocols to outperform earlier, non home-based all-software protocols, and have discussed the key causes of performance bottlenecks in each case [15]. However, the systems compared differ in both the communication and the protocol layers, so the value of hardware AU support is not isolated. As we develop the next generation of SVM systems, many recent developments make it necessary to revisit the design as well as the benefits of hardware support for automatic write propagation.

Consider the communication layer. Snooping-based automatic update has relied on write-through support in cache hierarchies so shared writes appear on the memory bus. However, the current generation of microprocessor based systems, including Intel-based PCs and especially SMPs, do not support write-through in second-level caches. To use AU in SVM protocols, we must design new mechanisms that work with write-back caches. Another important development is the appearance of efficient commodity NIs such as Myrinet [7] in the marketplace. How beneficial AU support will be with these NIs rather than the customized NIs in SHRIMP or Memory Channel [11] is unclear. Finally, an important trend is the use of commodity SMPs rather than uniprocessors as the building blocks for clusters, which can improve software shared memory performance [8, 17, 3, 24].

In the protocol layer, all-software versions of the home based protocols have also recently been developed [14, 30]. The result is a protocol very much like the hardware-supported *AURC*, except that propagation of changes to the home is done either at release or acquire time, using software diffs, rather than using AU or other write propagation at the time of the writes themselves. In an evaluation on the Intel Paragon multiprocessor [30], this software home-based protocol (called *HLRC*) was found to outperform earlier distributed all-software protocols even without hardware support.

This paper restricts itself to a home-based approach in the protocol layer, providing a consistent framework, and examines the use of hardware AU support in the communication layer in the context of emerging clusters. Specifically, the following questions are addressed:

(i) Given the decline of write-through capable second-level caches—and the bus traffic problems they raise with SMP nodes—can hardware automatic update support be provided even with write-back caches?

(ii) Within the same (home-based) protocol framework, does hardware support for automatic update snooping deliver a performance benefit large enough to justify its cost, assuming both write-through capable and write-back caches?

(iii) Does this hardware support for *AU* work well enough with the new commodity NIs like Myrinet, or does it require customized NIs like that in SHRIMP to achieve substantial benefits? That is, is *AU*-based SVM an argument for building customized NIs?

We address these questions through detailed simulation and for a wide range of real applications and computational kernels. The simulator gives us deep visibility into all aspects of the simulated hardware and software, including queues, buffers, and sources of contention, which is especially useful since the focus here is on the communication architecture. An associated flexible visualization tool allows us to easily examine the detailed event frequencies and contention-related bottlenecks in all parts of the node and NI on a per-node basis, and proves very useful in analyzing the observed performance effects. We use simulation because appropriate real systems don't exist, and real systems don't allow detailed enough analysis without additional

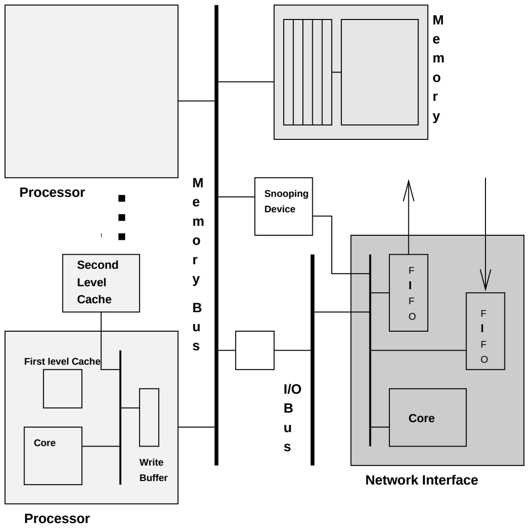

Figure 2: Simulated node architecture.

hardware/software instrumentation of NIs. Most of the key performance bottlenecks themselves have been diagnosed in detail in earlier research [15, 16, 5, 3, 23]; the focus here is on the impact of write propagation support. The major remaining bottlenecks, with or without *AU*, also guide us to where application and system design energy should be spent for SVM.

We find that: (i) AU support with write-through capable caches does significantly improve performance of some irregular applications, especially when diff-related costs dilate critical sections and increase serialization, but it can also hurt substantially in some cases; (ii) AU support with write-back caches solves the problems caused by the increased traffic and performs best, but is very intrusive into the underlying node; (iii) overall, the benefits do not appear to justify the design effort with commodity NIs, and NIs customized for *AU* packet generation may be important for using AU effectively.

2 Simulated Platforms

The simulation environment we use is built on top of augmint [25], an execution driven simulator using the *x86* instruction set, and runs on *x86* systems. The simulated architecture (Figure 2) models a cluster of 4-processor SMPs connected with a commodity interconnect like Myrinet [7], for a total of 16 processors. Contention is modeled in detail at all levels except in the network links. The processor is P6-like, but is assumed to be a 1 IPC processor. The data cache hierarchy consists of a 8 KBytes first-level direct mapped write-through cache and a 512 KBytes second-level two-way set associative cache, each with a line size of 32 Bytes. The write buffer [27] has 26 entries, 1 cache line wide each, and a retire-at-4 policy. The read hit cost is one cycle in the write buffer and first level cache and 10 cycles in the second-level cache. The memory subsystem is fully pipelined.

The memory bus is split-transaction, 64 bits wide, with a clock cycle four times slower than the processor clock. Arbitration takes one bus cycle, and the priorities are, in decreasing order: second level cache, write buffer, memory, incoming path of the network interface, outgoing path of the network interface. The I/O bus is 32 bits wide and has a clock speed half that of the memory bus. The relative bus bandwidths and processor speed match modern systems. If we assume that the processor has a 200MHz clock, the memory and I/O buses are 400 MBytes/s and 100 MBytes/s respectively.

Each NI has two 1 MByte memory queues for incoming and outgoing packets. Network links operate at processor speed and are 16 bits wide. We assume a fast messaging library [9] that supports explicit messages. Initiating a message takes on the order of tens of I/O bus cycles. If the NI queues fill, the NI interrupts the main processor and delays it to allow queues to drain.

A snooping device on the memory bus forwards AU traffic to the NI. The NI sets up network packets using its programmable core, which incurs a cost per packet (about 1000 cycles). In Section 8, we will examine the impact of using a customized NI that greatly reduces the cost of packet setup (to about 100 cycles).

Packets are delivered directly to memory, without processor intervention at the receive side. The packet size in the network is 256 bytes for DMA transfers and 4 bytes (1 word) for AU transfers. AU writes within a cache line are combined in the NI to reduce the number of packets.

Issuing an interprocessor interrupt costs 500 processor cycles, and invoking the handler is another 500 cycles. This is aggressive compared to what current operating systems provide, but is implementable [28] and prevents interrupt cost from swamping out the effects of other system parameters [5]. The page size is 4 KBytes, and the cost to access the TLB from a handler running in the kernel is 50 processor cycles. Each protocol handler is charged a cost depending on the work it does. The cost of creating and applying a diff in HLRC is computed by adding 10 cycles for every word that needs to be compared and 10 additional cycles for each word actually included in the diff.

The simulation parameters are as close as possible to an actual configuration, and the HLRC simulations validate very well against a real implementation. Our main goal was not so much to use the exact absolute values of the parameters but to maintain the important relationships among them. The simulator provides detailed statistics about all events in hardware, as well as statistics that help identify contention in the various components of the system. The programming model provided by the simulator is threads and the ANL macros. The simulator performs first touch allocation and statistics are reset in accordance with SPLASH-2 guidelines.

3 Protocol Efficiency: a Metric for Comparison

While speedup is an important metric, factors unrelated to the communication and protocol can cause speedup to be high even when protocol performance is poor. For example, a sequential execution can perform very poorly due to the working set not fitting in the cache, a problem which may go away in a parallel execution and thus lead to very high speedups. To understand how well protocols themselves perform, we use both speedups and a metric we call *protocol efficiency*.

We define protocol efficiency for a given application and protocol for N processors as:

$$f_N = \frac{\sum_{0 \leq i \leq N} (C_i + S_i)}{\sum_{0 \leq i \leq N} E_i},$$

where E_i, C_i, S_i are the elapsed, compute and local cache stall time of the i^{th} processor in the parallel execution. Obviously $f_N \in [0, 1]$. Protocol efficiency is the ratio of total compute plus local stall time for all processors to the total elapsed time for all processors. Note that the sequential execution is not involved in this metric.

4 Applications

We use the SPLASH-2 [29] application suite. A more detailed classification and description of the application behavior for SVM systems with uniprocessor nodes is provided in [15]. Here we describe only the characteristics of greatest relevance to this study. The applications can be divided in two groups, regular and irregular.

The regular applications are FFT, LU and Ocean. Their common characteristic is that when structured for good performance they are essentially single-writer applications: A given word of data is written only by the processor to which it is assigned. Given appropriate data structures they are single-writer at page granularity as well, and pages can be allocated among nodes such that writes to shared data are almost entirely local. Thus, in $AURC$ we do not need to use a write through cache policy, and in *HLRC* we do not need to compute diffs. Protocol action is required only to fetch pages. The applications have different inherent and induced communication patterns [29, 15], which affect their overall performance, but we should expect the different protocols to perform very similarly on their best versions. The irregular applications in our suite are Barnes, Radix, Raytrace, Volrend and Water.

5 Presentation of Results

The next few sections present our results. First we describe how automatic update support can be provided with write–back caches (*WB-$AURC$* protocol). Then we compare the three protocols (*HLRC*, $AURC$, *WB-$AURC$*) on a system with four, four-way SMP nodes for a total of 16 processors. We present overall performance data in two main forms: tables with speedups and protocol efficiencies, and per-processor bar-graphs with execution time breakdowns for the most interesting cases (Figures 3–7).

In the bar graphs the execution cost is divided into the following components. *Thread Compute Time* is essentially a count of the instructions executed by each thread. *CPU Stall Time* is the time the CPU is stalled on local memory references. *Thread Data Wait Time* (or page fetch time) is the time each thread is waiting for data to arrive from a remote node (page fetches). *Thread Lock Time* and *Thread Barrier Time* are the times spent in synchronization events (local or remote), including idle wait time and operation overhead. *I/O Stall Time* is the time the processor spends initiating page and message transfers and waiting for the network interface queues to drain. *Handler Compute Time* is the time spent in protocol handlers themselves. Some of this cost is also included in other costs, for instance *Thread Data Wait Time*.

To simplify the discussion, we clarify a couple of issues up front. *Thread Data Wait Time* can be large either because many pages need to be fetched (*frequency*) or because the cost per page fetch is high due to *contention*. Imbalances in data wait time also stem from imbalances in either of these factors, and we will indicate which dominates. Lock and barrier synchronization times also have two components: the time spent waiting (for another processor to release the lock or for all processors to reach the barrier) and the time spent exchanging messages and doing protocol work such as page invalidations (`mprotect`) and diffing. The simulator lets us separate these out too, and we call the former wait time and the latter the protocol cost. Idle time for locks is often increased greatly in SVM systems due to protocol activity and page misses occurring frequently inside critical sections and the resulting serialization [15, 16], as well as due to synchronization requests getting stuck behind other messages in queues.

6 Automatic update support with write–back caches

This section presents a new protocol (*WB-AURC*) that provides AU support with write–back caches, and its differences from *AURC*. [1]

The problem with write-back caches for AU support is that not all writes are visible to the memory bus snooper to be propagated to the home. In *WB-AURC* changes are propagated to remote homes either when modified lines are replaced or by flushing modified cache lines at release time [2]. In addition to snooping hardware, extra hardware support is needed to identify which words were modified in each cache line and only propagate those words. Otherwise, if entire cache lines are propagated, valid words at the home from more recent writes to other parts of the cache line could be overwritten by older values due to false sharing. One possible implementation, which we assume, is to add a dirty bit *per word* to the second level cache. Caches can either flush only the dirty words, or an external agent on the bus can snoop the evicted cache lines and propagate only the dirty words. Other possibilities include having the memory controller perform diffs in hardware on cache lines as they are written back to local memory, to detect the modified words.

Details about extending home-based protocols for systems with SMP nodes can be found in [3, 23]. The protocols are designed to be very similar, except for the different propagation mechanisms of data to the home.

7 Protocol Comparison

For the regular applications, FFT, LU, and Ocean, all three protocols perform very similarly for the better (contiguous) versions of the applications in SPLASH-2 and very poorly for the others (Table 1). The contiguous versions are essentially single writer applications at page granularity as discussed earlier. With proper data placement, they do not exercise the update propagation features that distinguish these protocols. There is essentially no update or write through traffic, and no diffs. The irregular applications are more interesting.

[1]Pages in the home nodes can either be mapped write-through, as assumed in [13], or can be mapped write-back since coherent DMA will provide the latest data to an incoming page fetch request; we assume the latter here since it performs better especially with SMPs.

[2]If the processor architecture does not allow flushes at user level, this may have to be done through a system call.

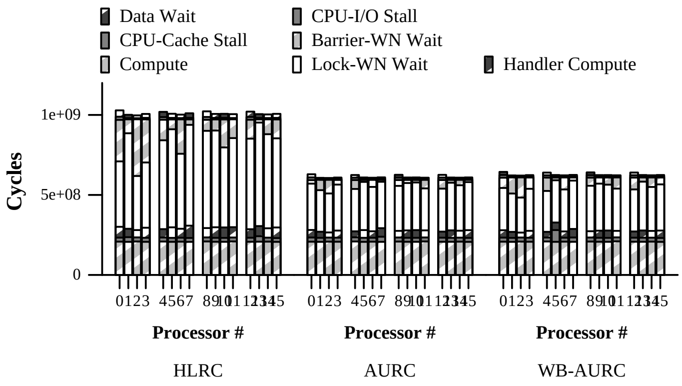

Figure 3: Cost breakdown for Barnes-rebuild for *HLRC*, *AURC* and *WB-AURC*.

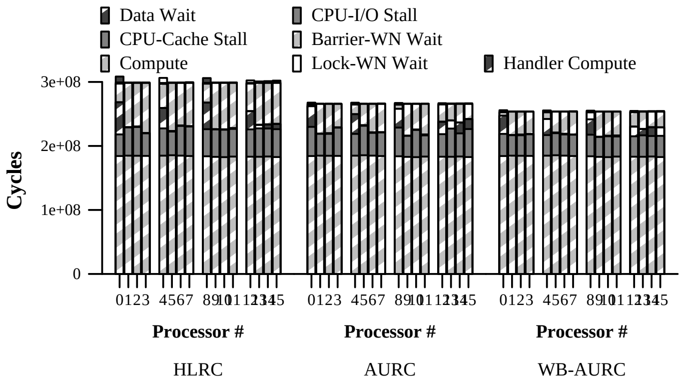

Figure 4: Cost breakdown for Barnes-no-locks for *HLRC*, *AURC* and *WB-AURC*.

Barnes-rebuild (Figure 3): Barnes-rebuild has an efficiency of less than 40% in all three protocols due to extremely high lock wait time [3]. Most of this time is spent in waiting to acquire the locks and not in protocol processing. This effect is exaggerated due to the dilation of critical sections because of page faults occurring in them, as discussed in Section 5. *HLRC* performs worse by a factor of two compared to *AURC*, mainly because of the much higher lock wait time: diff generation and application happen at synchronization time as well, dilating critical sections further and greatly increasing serialization at locks. *WB-AURC* performs similarly to *AURC*, since traffic is not a major issue.

Barnes-nolocks (Figure 4): By eliminating the locks, Barnes-nolocks performs very well under all protocols (Table 1). The main overhead is barrier cost. *HLRC* is somewhat worse because of the increased barrier cost due to diff computation and application. Restructuring the application not only increases performance greatly under all protocols, but also diminishes the differences among them.

[3]Locks are introduced to label point to point flag synchronization. When they are replaced by separate *acquire* and *release* primitives, the wait time is spent at those instead.

Application	Problem Size	Speedups			Efficiency Factors		
		AURC	HLRC	WB-AURC	AURC	HLRC	WB-AURC
FFT	18	6.20	5.74	5.99	42	40	45
FFT	20	8.29	8.28	8.29	65	59	65
LU (contiguous)	512	12.42	12.25	12.39	68	66	68
LU (non-contiguous)	512	0.40	-	4.02	3	-	27
Ocean (contiguous)	258	16.86	15.23	15.33	51	47	49
Ocean (contiguous)	514	12.83	12.80	13.17	82	79	81
Ocean (non-contiguous)	258	0.39	-	0.53	2	-	3
Ocean (non-contiguous)	514	0.55	-	-	2	-	-
Barnes (rebuild)	16K	6.24	3.82	6.08	39	24	38
Barnes (no-locks)	8K	12.81	12.19	13.28	83	80	84
Barnes (no-locks)	16K	12.95	11.52	13.56	84	76	85
Radix	1M	1.25	3.41	3.72	8	21	23
Raytrace	car	11.82	14.79	10.80	72	90	66
Volrend	head	11.95	8.81	11.86	76	56	77
Water (nsquared)	512	9.71	8.84	9.73	62	57	63
Water (spatial)	512	8.52	10.05	10.48	55	63	67

Table 1: Speedups and efficiency factors (as %) for the SMP configuration.

Radix: *AURC* performs very poorly with Radix, with a speedup of 1.25. The reason for the very low performance is high data wait and synchronization times. Contention, created by high AU traffic due to scattered remote writes, results in high page fault costs. The number of page faults is similar across processors but the costs vary significantly due to contention (from about 450K to 1900K cycles per page fetch) and are more than an order of magnitude higher than the uncontended page fetch cost of about 15K cycles. Moreover, the simulator shows that control (e.g. request) messages that are in the critical path are delayed in the outgoing and incoming network queues by this traffic as well. Most of the synchronization time is wait time due to imbalances and not the protocol overhead itself, although the imbalances are not in computation but are caused by contention. A lot of traffic is incurred just before barriers, further increasing their cost.

HLRC performs better than *AURC* (speedup is 3.41) because of the smaller number of messages. Updates to shared data in *HLRC* occur not at every write but at a release, which results in fewer and larger messages. The messages, including requests, are delivered faster due to less contention, so fetch time, lock time and barrier costs are much smaller and performance improves. Diff-related activity slows down barriers, but other effects dominate the differences between protocols.

WB-AURC does not have the increased traffic problems of *AURC* and thus regains the loss in speedup, improving it from 1.25 to 3.72. Data wait times and barrier costs are reduced, the latter even compared to HLRC due to lack of diffing. However, imbalances are still present in data wait times.

In general, scattered remote writes and high communication in the permutation phase make Radix a difficult application for SVM systems, particularly for *AURC* due to increased write through *AU* traffic. As in other applications, *WB-AURC* is the best performing protocol. To reduce scattering of writes and hence traffic and contention, we tried a version of Radix that first gathers the changes locally in a copy buffer and then propagates them to the remote nodes. This improved performance on 16 processors by a factor of 1.5-2 across protocols.

Raytrace (Figure 5): All protocols perform very well for this application. *HLRC* performs better than *AURC* and *WB-AURC* because of the smaller and better balanced data wait time. Although the total traffic is similar in all protocols, *HLRC* uses fewer, bigger messages for the update mechanism and the simulator shows that this reduces contention in the network queues. This is the only application where *WB-AURC* performs worse than *HLRC*.

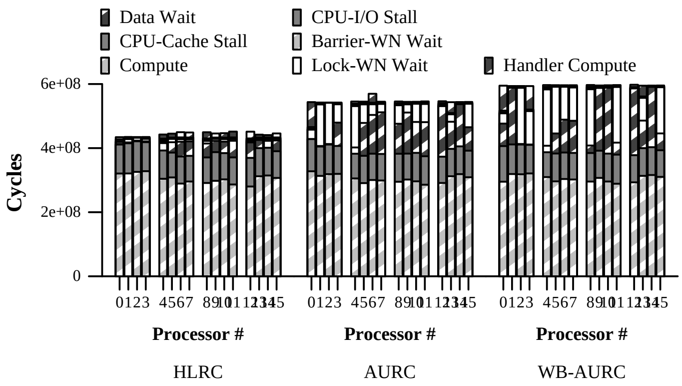

Figure 5: Cost breakdown for Raytrace for *HLRC*, *AURC* and *WB-AURC*.

Volrend (Figure 6): Data traffic and wait time are not very substantial in Volrend. *AURC* does very well. The major problem arises from imbalances in the compute time and high lock wait time due to the high overhead of task stealing [16]. Locks in Volrend protect task queue entries and hence migratory data. For all protocols, many processors contend for the same, relatively small number of locks, and requests for locks are often found to be stuck waiting in queues for other messages to be delivered first or for protocol handlers to run. First, contention occurs at the homes of the locks, especially since the homes are not assigned in a clever way for locality. Second, forwarded lock requests are queued at the current owner, waiting for earlier messages to be processed or for the current lock to actually arrive there. Third, page faults that occur within critical sections often have to be satisfied remotely (especially when

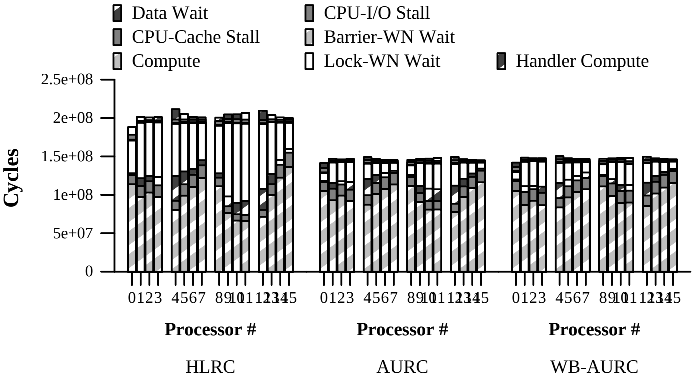

Figure 6: Cost breakdown for Volrend for *HLRC*, *AURC* and *WB-AURC*.

the locks and protected data are migratory), dilating the critical section [15, 16]. This becomes worse as page fetches themselves incur contention, exacerbated by lack of locality in the assignment of page homes. Observed lock wait times are at least an order of magnitude longer than the uncontended time for accessing an owned lock, and the times are imbalanced due to the different numbers of local and remote locks acquired by each processor. These per-lock times are much larger than in Barnes, where the problem is not so much the contention at locks as the sheer number of locks executed. For *HLRC* in particular, the migratory locks also causes a lot of diffing and diff transfers, further dilating critical sections. This last increase in lock time and serialization is alleviated with AU-based methods, which do not require diffs; together with the lack of need for interrupts on write propagations, this leads to much better performance. The behavior of *WB-AURC* is similar to *AURC*.

Water-nsquared: All protocols perform well with Water-nsquared since traffic is low and much of the application is single-writer. Protocol overheads are low and balanced once again. *HLRC* is somewhat worse than the other two protocols due to more expensive locks. Improving locks would be the best way to improve Water-nsquared performance.

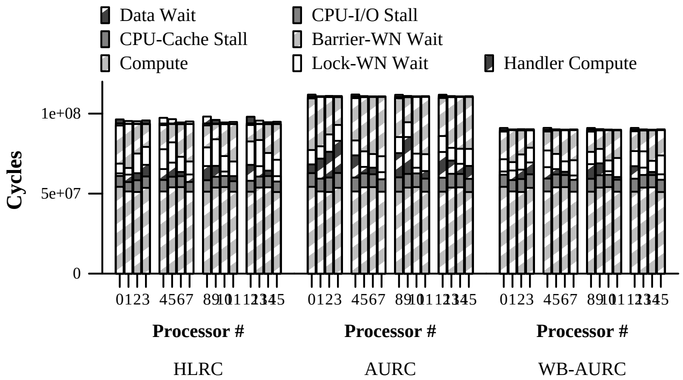

Figure 7: Cost breakdown for Water-spatial for *HLRC*, *AURC* and *WB-AURC*.

Water-spatial (Figure 7): *AURC* performs at about 50% parallel efficiency with Water-spatial. However, AU traffic creates contention in the outgoing network queues. The simulator shows that this slows down outgoing page requests, which get stuck behind AU traffic, and results in imbalances in data wait times. In *HLRC*, unlike *AURC*, protocol overheads are balanced and performance is considerably higher. There is no contention in the outgoing network queues and request messages are sent out immediately for all the processors. Data wait time is considerably smaller, (min 3%, avg 10%, max 19%) in *HLRC* as opposed to (min 6%, avg 20%, max 45%) for *AURC*.

WB-AURC performs best by a significant amount. It does not suffer from *AURC*'s contention-induced load balancing problems.

8 Customized Network Interfaces

We also examined the impact of customized network interfaces like the one in SHRIMP [6] on *AU* based protocols. A customized NI allows for much shorter packet preparation occupancies since it consists of a dedicated state machine instead of a more general purpose processor on commodity interfaces like Myrinet. Customized NIs yield greater benefits for protocols that make use of automatic update support, which tend to transfer more packets. The per-packet occupancy for customized NIs is set to about 100 cycles as opposed to about 1000 for the commodity NIs.

The results show that the performance of *AURC* improves for the two applications, Radix and Water-spatial, where raw *AU* traffic creates contention in the network queues. The speedup for Radix is improved by a factor of more than 2 (to speedup 2.5), since contention between AU and page traffic is reduced, and the effect of contention between AU and requests in Water-spatial disappears. For Radix, *WB-AURC* improves even more and the result is a speedup of more than 9 with this protocol! Also the speedup of Raytrace improves to almost 13.5 in both *AURC* and *WB-AURC*. The performance improvement in the other applications, which do not suffer from very high AU traffic, is marginal. Customized NIs make AU-based SVM more attractive for some applications.

9 Related Work

Several papers have discussed the design and performance of shared virtual memory for SMPs [8, 17, 10, 24, 3, 23] for different protocols. We focus here on studies that propose or evaluate hardware support.

Our results for AURC on uniprocessor systems are consistent with the results obtained in [15]. The problem of write through caches is touched upon in [22], in the context of different SVM protocols with uniprocessor nodes.

Bianchini et al. in [2] propose diffing in hardware as another form of hardware support for SVM. They assume write through caches and uniprocessor nodes. A dedicated protocol processor performs diffs on-the-fly and offloads overheads from the computation processor. Their simulations show that using the coprocessor can double the performance of TreadMarks [19] on a 16-node configuration. Results for using a coprocessor for diff computation and application for HLRC on the Intel Paragon are less optimistic [30]. The preeminence of interrupt cost among communication layer parameters is established in [5]. Inspired by this, smarter NIs (like Myrinet) are used to offload some key mechanisms for explicit data movement (remote read and remote put, as opposed to the implicit write propagation studied here) and synchronization from the main processor in [4], and the interactions with the protocol layer are examined.

Holt et al. present a metric similar to protocol efficiency [12] to characterize performance on a hardware cache coherent machine.

10 Discussion and Conclusions

We have examined the effectiveness of automatic update support for shared virtual memory on modern systems. We first discussed how the original AURC protocol, which relies on write-through second-level caches, can be extended to work with modern systems that have write-back caches (the *WB-AURC* protocol). Then we compared the performance of the three protocols, *HLRC*, *AURC* and *WB-AURC*, on a system with commodity network interfaces. *WB-AURC* solves the performance problems that arise with *AURC* due to the increased traffic in SMP systems, but requires hardware support that is quite intrusive into the node.

For many regular applications, hardware support for *AU* is not particularly useful. The different protocols all perform similarly. The nature of these applications is such that if effective data structures are used then they are single-writer even at page granularity, so with proper data placement there is little need for diffing or write propagation; if not, they perform poorly with all the home-based protocols (HLRC is generally less sensitive to poor data placement than AURC). Irregular applications are more interesting in this regard.

The main advantage of *AURC* (and *WB-AURC*) over *HLRC* is that it avoids diff computation and application. This protocol processing overhead is especially damaging to *HLRC* because it is incurred at synchronization points, where it increases the cascaded serialization effect that SVM protocols already suffer from due to dilation of critical sections. This shows up substantially in applications with either a lot of locking (e.g. Barnes-rebuild) or with a few locks that are heavily contended (e.g. Volrend due to task queues and Water-nsquared in the migratory force updates). In the latter, the serialization is worsened by lock requests having to wait a while in incoming node queues, first at the home and then at the previous owner while previous lock or other data requests are being serviced (either locally or, worse yet, remotely). By reducing at least one cause of the serialization (diffing), *AURC* and *WB-AURC* are substantially advantageous in cases with substantial locking, even with commodity NIs.

On the other hand, *AURC* can lead to a lot of AU traffic when the homes of pages being written are not local. It performs worse than *HLRC* in applications that cause a lot of raw AU and data traffic (e.g. Radix). The simulator showed that traffic is not only a problem for data transfers, but control messages that are in the critical path (e.g. page requests or synchronization messages) get stuck behind data messages (e.g. *AU*) even when inherent data traffic is not very high (e.g. Raytrace and Water-spatial). The effects of the resulting contention are imbalanced across processors and thus increase the wait time at barriers. Where traffic was a problem for *AU* we experimented with higher bandwidth memory and I/O buses, but these did not help very much in the latter cases since the real problem was that a few messages of a latency-critical type were stuck behind less critical *AU* packets. *AURC* is also less robust in performance to poor placement of pages across memories. While *WB-AURC* has the potential disadvantage that it occupies system resources with AU traffic over a longer period of time, overall it is always at least as good as the better of the other two protocols (with the exception of Raytrace) since it combines the advantages of both.

While there are some exceptions, most often the advantages of the AU-based approaches are not very large, at least with commodity network interfaces, confirming that much of the performance benefit of AURC over traditional LRC in [15] comes more from the home-based nature of the protocol layer rather than from the AU support [30]. The use of lower-occupancy, customized rather than commodity NIs improves *AURC* and *WB-AURC* performance substantially for the applications in which outgoing AU traffic is a problem; namely, Radix and to some extent Water-spatial. However, Barnes and Volrend, the applications where AU was particularly useful, are hardly affected, since the problem there is lock cost exacerbated by page misses within critical sections, not raw traffic. Applications that suffer due to control messages getting stuck in queues behind outgoing or incoming *AU* packets are also not helped so much. We also compared the protocols in a system with uniprocessor rather than SMP nodes, with similar conclusions. Additionally we examined other system configurations and less robust placement policies, but we omit the results due to space limitations. Overall, the lack of importance in regular applications and the need for customized network interfaces and especially for intrusive hardware support caused us to not pursue *AURC*-based approaches for our current cluster infrastructure of SMPs connected by Myrinet.

Our analysis confirms several key outstanding problems for home-based SVM systems, most of which have been observed earlier (e.g. [15, 16, 3, 5, 22, 1]), and that are not alleviated fully by AU. Let us examine these layer by layer (see Figure 1 and [26]). In the communication layer, the problem of latency-critical control and request messages getting stuck behind others in queues can perhaps be alleviated by using separate queues for different message or packet types, or simply different priorities for them. The problem of interrupts being expensive and this increasing the queuing and lock serialization effects can be addressed by using a smart network interface to perform some of the key general-purpose data movement and synchronization functions instead of interrupting the processor, most of which functionality is supported in the Virtual Interface Architecture industry standard API, and which may be aided by altering the laziness and other properties of the protocol layer as well [4]. In the protocol layer itself, the homes of locks and pages are often not assigned well with respect to computation in irregular applications, which can exacerbate end-point contention (e.g. Volrend). Adaptive placement may be useful here. Highly contended locks that protect migratory data pose a problem, which can perhaps be solved by adaptively using different protocols when this pattern is detected. Protocol overheads and contention due to diffing and page invalidations at barriers also requires addressing. Finally, an interesting observation is that improvements to the application layer [16] increase performance much more than automatic update support (e.g. Barnes in this study), and these improvements tend to reduce the differences among protocols. More research is needed to understand how best to use the three layers in synergy to greatly improve the performance of SVM systems.

11 Acknowledgments

We are indebted to NEC Research and particularly to James Philbin and Jan Edler for providing us with simulation cycles and a stable environment. We thank Hongzhang Shan for making available to us improved versions of some of the applications.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writer. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, pages 261–271, Feb. 1997.

[2] R. Bianchini, L. Kontothanassis, R. Pinto, M. D. Maria, M. Abud, and C. Amorim. Hiding communication latency and coherence overhead in software dsms. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[3] A. Bilas, L. Iftode, R. Samanta, and J. P. Singh. Supporting a coherent shared address space across SMP nodes: An application-driven investigation. In *IMA Workshop on Parallel Algorithms and Parallel Systems*, Nov. 1996.

[4] A. Bilas, C. Liao, and J. P. Singh. Network interface support for shared virtual memory on clusters. Technical Report TR-579-98, Computer Science Department, Princeton University, Princeton, NJ-08544, Mar. 1998.

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

[6] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[7] 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.

[8] A. 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, Apr. 1994.

[9] 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.

[10] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: analyzing the performance of clustered distributed virtual shared memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, October 1996.

[11] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, Feb. 1996.

[12] 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.

[13] 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.

[14] L. Iftode, J. Singh, and K. Li. Scope consistency: a bridge between release consistency and entry consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

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

[16] 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.

[17] M. Karlsson and P. Stenstrom. Performance evaluation of cluster-based multiprocessor built from atm switches and bus-based multiprocessor servers. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[18] M. G. H. Katevenis, E. P. Markatos, G. Kalokerinos, and A. Dollas. Telegraphos: A substrate for high-performance computing on workstation clusters. *Journal of Parallel and Distributed Computing*, 43(2):94–108, 15 June 1997.

[19] 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.

[20] P. Keleher, A. 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.

[21] L. I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, Jr., S. Dwarkadas, and M. L. Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *Proc. of the 24th Annual Int'l Symp. on Computer Architecture (ISCA'97)*, pages 157–169, June 1997.

[22] L. I. Kontothanassis and M. L. Scott. High performance software coherence for current and future architectures. *Journal of Parallel and Distributed Computing*, Nov. 1995.

[23] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based svm protocols for smp clusters: Design, simulations, implementation and performance. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture, Las Vegas*, February 1998.

[24] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[25] 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.

[26] J. P. Singh, A. Bilas, D. Jiang, and Y. Zhou. Limits to the performance of software shared memory: A layered approach. Technical Report TR-576-98, Computer Science Department, Princeton University, Princeton, NJ-08544, Nov. 1997.

[27] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *The 3nd IEEE Symposium on High-Performance Computer Architecture*, Feb 1997.

[28] D. Stodolsky, J. B. Chen, and B. Bershad. Fast interrupt priority management in operating system kernels. In USENIX Association, editor, *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures: September 20–21, 1993, San Diego, California, USA*, pages 105–110, Berkeley, CA, USA, Sept. 1993. USENIX.

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

[30] 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.