Comparison of Shared Virtual Memory across Uniprocessor and SMP Nodes Princeton University Technical Report TR-550-97

Angelos Bilas, Liviu Iftode, and Jaswinder Pal Singh

Department of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08544

{bilas, liv, jps}@cs.princeton.edu

Abstract

As the workstation market moves form single processor to small-scale shared memory multiprocessors, it is very attractive to construct larger-scale multiprocessors by connecting symmetric multiprocessors (SMPs) with efficient commodity network interfaces such as Myrinet. Using a shared virtual memory (SVM) layer for this purpose preserves the shared memory programming abstraction across nodes. We explore the feasibility and performance implications of one such approach by comparing the performance of two SVM protocols on systems with uniprocessor and SMP nodes. *AURC* (Automatic Update Release Consistency) is a protocol developed for the SHRIMP multicomputer that takes advantage of the available Automatic Update hardware support. *HLRC* (Home-Based Lazy Release Consistency) is an all-software version of *AURC*. These protocols were demonstrated to achieve good performance and to outperform earlier all-software systems on configurations with uniprocessor nodes.

In this paper we use an architectural simulator and ten applications to investigate the potential of using SMP nodes. Through detailed simulations we find that using SMPs improves performance for most of the applications, because of more sharing, effective prefetching and lower synchronization costs. The protocols, however, exhibit different behavior for some applications. *HLRC* is in general more robust and predictable, whereas *AURC* exhibits more problems with contention due to the increased traffic.

1 Introduction

Recently, small-scale shared-memory SMPs have become increasingly widespread. Inexpensive SMPs based on Intel PC processors are on the market, and SMPs from other vendors are increasingly popular. Given this development, it is natural to examine whether the SVM mechanisms developed for uniprocessor nodes extend well to using an SMP as the base node, and how these mechanisms can be improved for this task. In this environment, the bus coherence protocol operates at cache line level within an SMP, and the higher-overhead SVM mechanism provides communication and coherence at page granularity across SMPs. Another view of this approach is that the less efficient SVM is used not as the basic mechanism with which to build multiprocessors out of uniprocessors, but as a mechanism to extend available small-scale machines to build larger machines while preserving the same desirable programming abstraction. The trends in the marketplace make this two-level approach attractive.

AURC (Automatic Update Release Consistency) [10, 12] is a *home-based* protocol, designed in the context of the SHRIMP multicomputer [2]. *AURC* employs hardware to snoop the memory bus for writes to shared data. Every shared page has a *home* node, and writes observed to a page are automatically propagated to the home at a fine granularity in hardware. Pages that are actually shared are mapped write-through in the caches so that writes appear on 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. Data are maintained consistent according to a software consistency protocol such as lazy release consistency [16].

Hardware assisted home–based protocols based on automatic update (*AU*) appear to outperform previous all-software non-home-based protocols like TreadMarks [15] across a range of applications [12], on systems with uniprocessor nodes. This result suggests that hardware support for *AU* may be very useful for providing a shared address space across commodity PCs or workstations.

Emulated all-software versions of the home based protocols were also demonstrated to perform well. The idea, proposed in [11], is to compute diffs as in previous all-software protocols, but to propagate the diffs to the home at a release point and apply them eagerly, keeping the home up to date according to lazy release consistency. In a preliminary evaluation on the Intel Paragon multiprocessor [23], this software home-based protocol (called *HLRC*) was found to outperform earlier distributed all-software protocols.

Moreover the appearance of efficient commodity NIs such as Myrinet [3] in the marketplace raises the possibility of implementing *AU* with these, rather than customized NIs. On the other hand, the write-through capable caches that *AU* relies on are becoming rare, calling the approach into question.

These new developments raise the question of, how will using k, c-processor SMPs, connected this way, compare

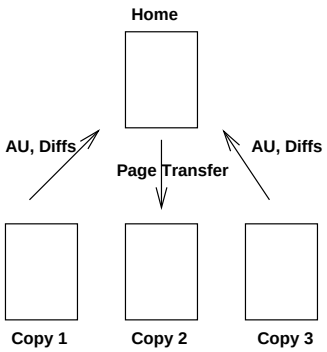

Figure 1: Updates are sent to the home node with a protocol specific mechanism: Diffs for *HLRC* and *AU* for *AURC*. Whole pages are fetched from the home on demand.

to using SVM across $k * c$ uniprocessor nodes. Clustering processors together using a faster and finer-grained communication mechanism has some obvious advantages; namely prefetching, cache-to-cache sharing and overlapping working sets [7]. The hope is that for many applications a significant fraction of the interprocessor communication may be contained within each SMP node. This reduces the amount of expensive (high latency and overhead) cross-node SVM communication needed during the application's execution. However, it unfortunately *increases* the bandwidth (i.e. communication per unit time) demands on the node-to-network interface. This is because the combined computation power within the SMP node typically increases much faster with cluster size c (linearly) than the degree to which the per processor cross-node communication volume is reduced. This means that depending on the constants, the node-to-network bandwidth may become a bottleneck if it is not increased considerably when going from a uniprocessor to an SMP node.

We explore these issues with two SVM protocols, *AURC* and *HLRC*. Furthermore, attached to all performance issues are, the classes of applications for which the results hold. The tradeoffs must be understood in light of real applications, and through detailed analysis of their high-level and low-level system interactions. We address these issues through detailed simulations and careful performance analysis in this paper, for both regular and irregular applications.

We find that out of ten applications, both protocols improve substantially with the use of SMPs in five of them, in three there is a smaller improvement (or they perform the same as in the uniprocessor node case) and for the other two results differ for each protocol with *AURC-4* performing worse than *AURC-1* for one application.

2 Protocols

We present results for two forms of an SVM protocol based on a lazy release consistency model (LRC): *AURC* and *HLRC* [1]. Both protocols use timestamps to maintain ordering of events and they are home based. As mentioned before, *AURC* uses *AU* and customized network interfaces and *HLRC* is the all–software version of *AURC*. Descriptions of *HLRC* and *AURC* can be found respectively in [23] and [10].

AURC and *HLRC* were developed for systems that have one compute processor per node [10, 23]. Both have recently been extended to support SMP nodes [1].

Avoiding the details of protocol extensions to support SMP nodes, we should mention that some of the issues are quite involved and extensions are not straightforward. There are two types of issues that require attention. Correctness issues and performance issues. In the first category, for instance, fall TLB invalidations whereas, in the second, barrier implementation.

TLBs of other processors within the node need to be invalidated whenever a processor issues a page fetch, so that the incoming page does not overwrite in the meantime writes by other processors to the shared page.

In this work all protocols use hierarchical barriers for the SMP configurations. Moreover there is a tradeoff in the amount of protocol processing that needs to be done in the barrier master. Either the barrier master sends to each processor the necessary write notices or it forwards all the write notices to everybody and lets them select the ones they need. This tradeoff between size of messages and protocol processing is difficult to resolve. Finally, in *HLRC* diffs are a run length encoding over the addresses in the page, along with the data in these addresses.

The goal of the SMP protocol extensions is to reduce the frequency and cost of SVM protocol invocation by taking advantage of hardware cache-coherence and synchronization whenever possible within a node.

In implementing the protocols, we try to keep basic aspects fixed across protocols. This allows us to focus on the impact of the different propagation mechanisms of data to the home (Figure 1). As mentioned in the introduction, *AURC* uses write propagation to update the home nodes whereas *HLRC* uses traditional diffs.

In both *AURC* and *HLRC*, the home node for a page experiences less overhead than other nodes: pages at home are mapped write-back in *AURC*, no diffs are computed at the home in *HLRC* and there are no page fetches at the home in any case.

Finally, *HLRC* achieves ordering of all messages completely in software whereas *AURC* requires point-to-point FIFO delivery in the NIs. This may lead to more flexibility in protocol design for *HLRC*.

Several variations of these protocols are possible. For instance delaying diff computation until acquire, minimizing the use of interrupts etc. In the implementation of the protocols, we choose to maintain a uniform framework for all protocols, so that the comparison is more direct.

3 Simulated Platforms

The simulation environment we use is built on top of augmint [20], an execution driven simulator using the *x86* instruction set and runs on *x86* systems.

The simulated architecture (Figure 2) assumes a cluster of c–processor SMPs connected with a commodity interconnect like Myrinet [3]. Contention is modeled at all levels except 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 32B. The write buffer [21] has 26 entries, 1 cache line wide each, and a retire-

[1] Although these are versions of the same basic home-based protocol, we will refer to them as separate protocols to ease the discussion.

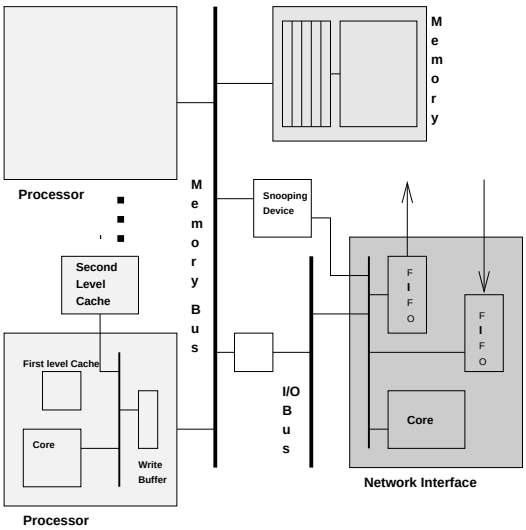

Figure 2: Simulated node architecture.

at-4 policy. Write buffer stalls are simulated. 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 4x 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 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 clock of 200MHz, the memory and I/O buses are 400 MBytes/s and 100 MBytes/s respectively.

Each network interface (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 system [6, 17, 5] that supports explicit messages. Initiating a message takes on the order of tens of I/O bus cycles. If the network 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. This cost must be paid in these commodity NIs proposed for use in next-generation systems.

Issuing an Interprocessor Interrupt (IPI) costs 500 processor cycles, and invoking the handler is another 500 cycles. This is very aggressive compared to what current operating systems provide, but is implementable and prevents interrupt cost from swamping out the effects of other system parameters. Protocol handlers cost a variable amount of cycles. The page size is 4 KBytes, and the cost to access the TLB from a handler running in the kernel is 50 processor cycles. In accordance to simple experiments, 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.

In setting the simulation parameters we tried to be as close as possible to an actual configuration. Our main goal was not so much to use the exact absolute values of the parameters but to maintain the important relations among them. Since the processor is less aggressive than the latest generation of processors (1 IPC at 200MHz versus 2-3 IPC at 200-400 MHz), we scaled down the values that affect the performance of the memory subsystem and the NI as well. Thus we use somewhat smaller caches and slower memory and I/O buses and.

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. One limitation at this point is that the simulations do not capture the burstiness of certain events, for instance traffic on the memory bus. Also protocol handlers can not be simulated since the simulator itself is not multi-threaded.

The programming model provided by the simulator is threads and the ANL macros. The simulator performs first touch allocation. To avoid allocating all the pages to the thread that initializes the pages, we do not simulate the initialization phase. Statistics are reset in accordance with SPLASH-2 guidelines.

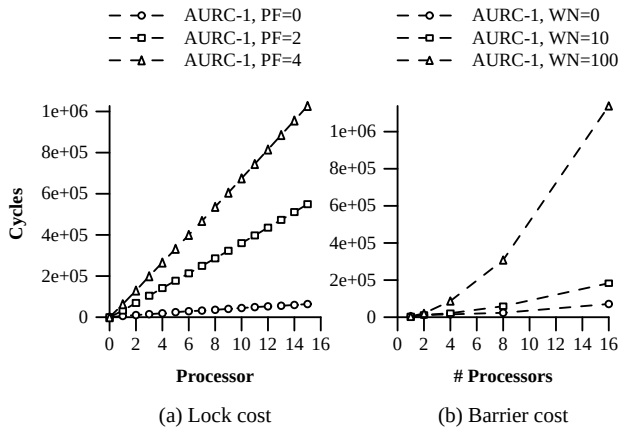

Figure 3: Lock and Barrier times for simple cases. The first graph from the left shows the time needed by 16 processors to acquire a lock one after the other. Each curve represents different number of page fetches in the critical section. The second graph presents the barrier cost for various numbers of processors and write notices.

4 Micro-benchmark Analysis

To understand the costs of the basic protocol and synchronization operations in this complex system, and to gain confidence in the simulator, we use a set of micro-benchmarks. These measure:

- The time to fetch a page, including the request message, the page transfer itself, and the handlers at both ends.
- The cost to acquire a lock for different numbers of competing processors, page fetches (misses) in the critical section, and write notices created in the critical section.
- The cost of a barrier for different numbers of processors, write notices and diff sizes.

The unloaded cost of a page fetch is about 15000 processor cycles, or 75μs with a 200 MHz processor and the default

network configuration. The one way latency for a one-word message is about 10μs, and the time to deliver interrupts and run handlers is similar. This results in an uncontended network bandwidth of about 70-75 MBytes/s for a one-way page transfer, out of the theoretical 100 MBytes/s of the NI. The latency and bandwidth numbers are in agreement with the reported performance numbers for a real implementation [5].

Uncontended lock acquisition from a remote node costs 5800 processor cycles or 29μs with no write notices or page fetches in the critical section, and from the local node it is about 2100 cycles. As page fetches inside the critical section are increased, lock wait time increases dramatically in SVM systems due to serialization, (Figure 3).

The cost of barrier synchronization can be seen to go up sharply with the number of write notices produced in the previous interval, since these are communicated to the barrier master and from there to all nodes.

5 Protocol Efficiency: a Metric for Comparison

While speedup is an important metric, factors unrelated to the SVM protocol can cause speedups to be high even when the protocol itself is not well suited to the application. 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. We will see an example of this in the Ocean application. 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:

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

where E_i, C_i, S_i are the elapsed, compute and cache stall time of the i^{th} processor in the parallel execution. Obviously $f_N \in [0, 1]$. Note that the costs in the sequential execution are not involved in this definition. Alternative definitions are also possible. We use this definition since it captures the effects we are interested to quantify.

6 Applications

In our evaluation we use the SPLASH-2 [22] application suite. We will now briefly describe the basic characteristics of each application. A more detailed classification and description of the application behavior for SVM systems with uniprocessor nodes is provided in the context of *AURC* and *LRC* in [12]. The applications can be divided in two groups, regular and irregural.

6.1 Regular Applications

The applications in this category are FFT, LU and Ocean. Their common characteristic is that they are optimized to be 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 mostly local. 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 [22, 12], which affect their performance and the impact on SMP nodes.

FFT: The all-to-all, read-based communication in FFT is essentially a transposition of a matrix of complex numbers. We use two problem sizes, 256K(512x512) and 1M(1024x1024) elements.

If we assume that each of the k nodes has c processors ($c \ge 1$) and $D = n/p$ is the number of elements per processor, computation time is $O(D\ log\ D)$ and cross-cluster SVM communication volume from the p processors in the node is $O(\frac{c*D}{k} * (k-1)) = O(\frac{c*\frac{n}{p}}{\frac{p}{c}} * (\frac{p}{c} - 1)) = O(c * \frac{n*(p-c)}{p^2})$. This gives a communication to computation ratio of $r = O(c * \frac{p-c}{p*\log \frac{n}{p}})$.

For small problem sizes r in FFT is relatively high. As problem size grows, r decreases. This is reflected in the results for the two problem sizes we examined.

Another factor in r is the number of processors in each SMP. r is a factor of c larger in an SMP system with c processors per node compared to a uniprocessor node system. Thus as we move to SMPs, we benefit from sharing but communication increases as well.

LU: The contiguous version of LU allocates on each page the data of only one processor. LU exhibits a very small communication to computation ratio but is inherently imbalanced. We used a 512x512 matrix.

The communication to computation ratio is $O(\sqrt{p}/n)$, where n is the dimension of the array and p the total number of processors. Note that the communication to computation ratio does not depend on the number of processors in each node.

Ocean: The communication pattern in the Ocean simulation application is largely nearest-neighbor and iterative on a regular grid. We run the contiguous (4-d array) versions of Ocean on two problem sizes, 258x258 and 514x514, with an error tolerance of 0.001.

If we consider a k node system with c processors per node (a total of $p = k * c$) and an n-by-n grid, the computation time is $O(n^2/p)$ (a processor's partition area) and the cross-cluster communication volume is $O(n/\sqrt{k}) = O(n/\sqrt{p/c})$ (the perimeter of the total partition assigned to all processors in the cluster). It is interesting to note that, in the case of multiprocessor nodes, even ignoring intra-cluster communication and its interference with the cross-cluster communication, the inter-node communication bandwidth needs are greater by a factor of $\sqrt{c}$.

6.2 Irregular Applications

The irregular applications in our suite are Barnes, Radix, Raytrace, Volrend and Water.

Barnes: We ran experiments for different data set sizes, but present results for 16K particles. Access patterns are irregular and fine-grained. We use two versions of Barnes. The first version (Barnes-rebuild) builds the tree from scratch after each computation phase. The second version (Barnes-nolocks) [13] is optimized for SVM, and it avoids locking as

much as possible. It stems from another version of Barnes, where each processor works on its own partial tree of the data, and all partial trees are merged to the global tree after each computation phase.

Radix: Radix sorts a series of integer keys. It is a very irregular application with highly scattered remote writes. The version we use is the unmodified SPLASH-2 version.

Raytrace: Raytrace renders complex scenes in computer graphics. The version we use is modified from the SPLASH-2 version to run more efficiently on SVM systems. A global lock that was not necessary was removed, and task queues are implemented better for SVM and SMPs. Inherent communication is small. We only present numbers for the SMP protocols due to simulation cycle limitations.

Volrend: The version we use is also slightly modified from the SPLASH-2 version, to provide a better initial assignment of tasks to processes before stealing. This improves SVM performance greatly. Inherent communication volume is small.

Water: We use two versions of Water, Water-nsquared and Water-spatial. Water-nsquared can be categorized as a regular application, but we put it here to ease the comparison with Water-spatial. In both versions, updates are accumulated locally between iterations and performed at once at the end of each iteration. The inherent communication to computation ratio is small. We use a data set size of 512 molecules.

7 Presentation of Results

In the next sections we present our results that address the issues raised in the introduction. We compare two system configurations for the two different protocols. The first system has uniprocessor nodes, whereas the second one uses SMP nodes. In all configurations the speed of the memory and I/O buses is set to 400 MBytes/s and 100 MBytes/s respectively. These values as mentioned in Section 3 result in a realistic commodity configuration, given the relative performance of the different components.

The simulator provides us with detailed statistics for most hardware and protocol events. This allows us to look into what is actually happening in the system and to avoid guesses about its behavior.

In the analysis we present overall performance data in two main forms. Tables with speedups and efficiency factors in each section and bar-graphs with cost breakdowns for the most interesting cases (Figures 4–10). Each section has a table that contains the speedups and protocol efficiencies achieved for each application for the protocols and configurations under consideration.

In the bar graphs the execution cost is divided in 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, including idle wait time and operation overhead. *I/O Stall Time* gives the time the processor spends initiating page and message transfers and waiting for the network interface queues to drain. The last component, *Handler Compute Time*, is the time spent in protocol handlers. Some of this cost is also included in other costs, for instance *Thread Data Wait Time*.

Tables 3-4 in Appendix A give more analytical statistics about each application. These should be consulted while reading the analysis of each application. Important statistics are highlighted in the analysis of each application as well.

To simplify the discussion, we now 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 (e.g. after the last processor arrives at the barrier). We will separate these out, calling the former wait time and the latter the protocol cost. As we saw in Section 4, idle time for locks is often increased greatly in SVM systems due to page misses occurring frequently inside critical sections and increasing serialization [12].

Finally, we adopt a naming convention in which (for example) *AURC-1* is *AURC* on a system with one processor per node and *HLRC-4* is *HLRC* on a system with 4 processors per node.

8 System Comparison

In this section we discuss how the protocols behave when moving from uniprocessor to SMP nodes. Tables 1 and 2 present these results.

From Table 1 we can divide the applications in different classes. The first class is applications for which both protocols improve with SMPs. These applications are LU, Ocean-contiguous, Barnes-rebuild, Volrend and Water-spatial. We differentiate the behavior of these applications in three subgroups.

The first group consists of Ocean (Figure 5), which improves dramatically because of the patter of communication and the amount of synchronization. In both *AURC-4* and *HLRC-4* data wait times are reduced significantly, compared to the uniprocessor configuration. The page fetch cost however, is increased by about 100% and 30% in *AURC-4* and *HLRC-4* respectively because of the larger contention in the memory bus and network interface. The reduction in data wait time comes from the sharing pattern in Ocean. The communication pattern is nearest neighbor and in Ocean-contiguous the data needed by each processor are mostly owned by the SMP node.

Besides lower data wait times, Ocean exhibits in the SMP case much lower synchronization as well. This applications, uses a large number of barriers to synchronize between phases and timesteps of the computation. The use of hierarchical barriers in *AURC-4* and *HLRC-4* reduce the cost of synchronization dramatically.

AURC-4 gives a speedup of more than 16 for the small problem size in Ocean-contiguous. As was mentioned before, the super-linear speedup is due to cache effects. More precisely the sequential run, suffers from extremely high stall

Application	Problem Size	Speedups			
		AURC-1	AURC-4	HLRC-1	HLRC-4
FFT	18	5.24	6.20	4.43	5.74
	20	8.53	8.29	7.73	8.28
LU (contiguous)	512	10.78	12.42	10.16	12.25
Ocean (contiguous)	258	6.10	16.86	5.43	15.23
	514	6.52	12.83	6.12	12.80
Barnes (rebuild)	16K	5.30	6.24	2.45	3.82
Barnes (no-locks)	8K	12.94	12.81	11.69	12.19
	16K	13.30	12.95	10.94	11.52
Radix	1024	2.82	1.25	0.63	3.41
Raytrace	car	6.38	11.82	14.06	14.79
Volrend	head	9.14	11.95	7.86	8.81
Water (nsquared)	512	9.09	9.71	8.56	8.84
Water (spatial)	512	7.89	8.52	7.41	10.05

Table 1: Speedups for the uniprocessor and the SMP node configurations.

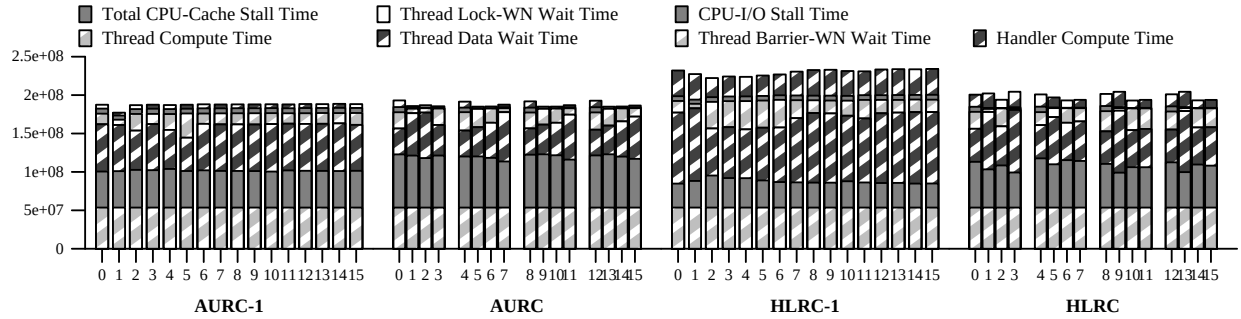

Figure 4: Cost breakdown for the 1M FFT for *AURC* and *HLRC*.

time (2-3 times the compute time). The parallel application takes advantage of the smaller working set size in each processor, and the stall time is reduced substantially.

In the second group belong LU and Barnes-rebuild (Figure 6), where the improvement comes again from sharing of data and lower synchronization costs as well, but at a much smaller degree than Ocean. These are applications, where clustering helps in data sharing and prefetching, without the inherent communication pattern of the applications to match the architecture as in Ocean. Also cheaper synchronization makes a noticeable difference.

In the third subgroup belong applications where there is an improvement but at a different degree for *AURC-4* and *HLRC-4*. These are Volrend (Figure 9), and Water-spatial (Figure 10). Volrend uses a task stealing mechanism to achieve load balancing. This mechanism uses locks to perform atomic operations on the task queue. Although both protocols benefit from cheaper local locks, remote locks are more expensive in *HLRC*. Moreover the distribution of local–remote locks in each processor is somewhat more uneven in *HLRC* which leads to higher imbalances. As a result the improvement in *HLRC-4* is smaller than *AURC-4*.

Contrary to Volrend, Water-spatial exhibit a bigger improvement with *HLRC-4*. In this case, update traffic creates contention in the network queues. In *AURC-4* this problem is exaggerated because more processors use the same network interface and memory bus to communicate.

The second class of applications consists of FFT (Figure 4), Barnes-nolocks (Figure 7), and Water-nsquared. These are applications that do not benefit (or benefit a little) from the use of SMP nodes in any protocol. In these cases, performance in already high and using SMPs improves certain costs, but in some cases other costs are increased or imbalances are created and the effects of sharing, prefetching and cheaper synchronization are overall small. For instance in FFT, cache stall time is increased in the SMP case, because of the increased load in the memory bus. In Barnes-nolocks there is a reduction in the data wait time because of sharing and prefetching. However, this reduction appears only in some of the processors in each node and it creates imbalances that limit the benefits of clustering.

In the third class belong Radix and Raytrace (Figure 8) that exhibit different behavior under each protocol. In *HLRC* Radix improves with SMP nodes, whereas in *AURC*

Application	Problem Size	Efficiency Factors			
		AURC-1	AURC-4	HLRC-1	HLRC-4
FFT	18	36%	42%	26%	40%
	20	57%	65%	45%	59%
LU (contiguous)	512	58%	68%	53%	66%
Ocean (contiguous)	258	21%	51%	19%	47%
	514	33%	82%	29%	79%
Barnes(rebuild)	16K	33%	39%	15%	24%
Barnes (no-locks)	8K	86%	83%	74%	80%
	16K	89%	84%	70%	76%
Radix	1024	17%	8%	4%	21%
Raytrace	car	39%	72%	86%	90%
Volrend	head	58%	76%	50%	56%
Water(nsquared)	512	58%	62%	55%	57%
Water(spatial)	512	51%	55%	46%	63%

Table 2: Efficiency factors (as %) for the uniprocessor and the SMP configurations.

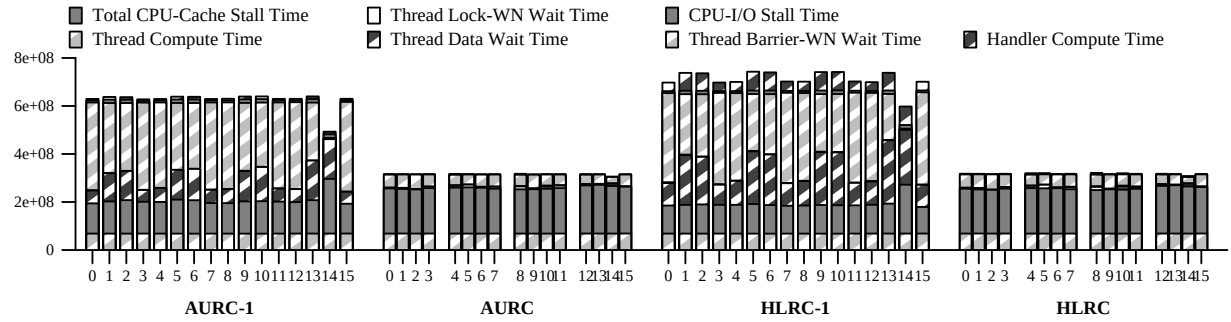

Figure 5: Cost breakdown for Ocean-contiguous (514x514) for *AURC* and *HLRC*.

it performs worse.

AURC-4 performs worse than *AURC-1* although the amount of data sent as replies to page requests decreases to about one fourth in the SMP configuration. Performance is much worse because *AU* traffic per processor is about the same as in *AURC-1* and buses are shared. This creates contention in all components of the path between the sender and the receiver, delaying both *AU* and request/control messages. Consequently all forms of communication and synchronization are slowed down. For instance the page fetch cost is huge, (1312487 processor cycles on the average). Protocol efficiency is very low for *AURC-4*.

HLRC-4 performs much better than *HLRC-1*. *HLRC-4* does not suffer from increased traffic as much as *AURC-4*. Messages are delivered faster and fetch time, lock time and barrier costs are much smaller, so performance improves.

In Raytrace *AURC* benefits a lot from sharing whereas the improvement for *HLRC* is very small. The reason for the improvement in *AURC* is the small messages that are used to update data. When SMP nodes are used the number of messages is reduced substantially (the amount of data is reduced by a factor of 4) and as a result the contention problem is much less severe than the uniprocessor node case. In *HLRC* the number of messages is not a problem since large messages are used to update the home nodes. Thus the performance of *HLRC-1* is much better than *AURC-1* and the benefit of using SMP nodes much smaller.

9 Related Work

Our study on uniprocessor systems is consistent with the results obtained in [12] for their "slow bus" case, although we compare *AURC* against *HLRC* instead of *LRC*. *HLRC* and *LRC* have been compared for some applications on the Intel Paragon in [23].

Holt et al. in [9] present a metric similar to protocol efficiency. They use it to characterize the performance of applications on a hardware cache coherent machine.

Several papers have discussed the design and performance of shared virtual memory for SMPs [4, 14, 8, 19] for different protocols. Erlichson et al. [8] conclude that the transition from uniprocessor to multiprocessor nodes is nontrivial and performance can be seriously affected unless great care is taken.

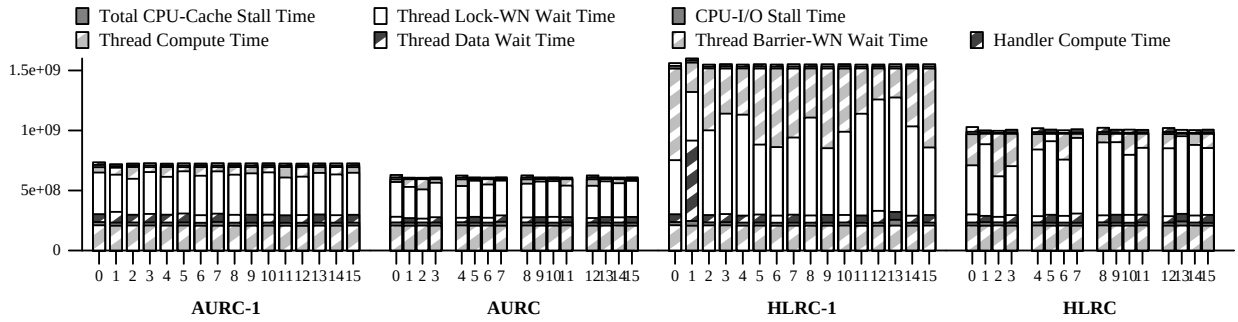

Figure 6: Cost breakdown for Barnes-rebuild for *AURC* and *HLRC*.

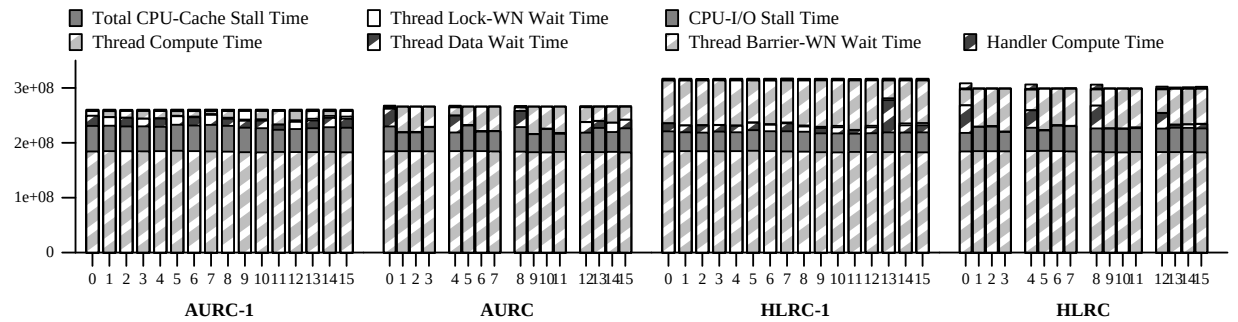

Figure 7: Cost breakdown for Barnes-nolocks for *AURC* and *HLRC*.

[1] is a preliminary version of this work. For this work, we use a much more detailed simulation environment and improved implementations of the the protocols.

10 Summary and Conclusions

Using SMP nodes, for the same total processor count improves performance for both protocols, particularly when the protocols and applications are designed and used properly. This is despite the fact that the traffic pressure on the memory and I/O buses tends to increase. In some cases imbalances are introduced because of increased contention and overall benefits are reduced. However, the reduction in the number of remote page fetches and in synchronization cost tends to overshadow the increased contention, resulting in a small improvement in some applications and a substantial improvement in others.

We find that out of ten applications, both protocols improve substantially with the use of SMPs in five of them, in three there is a smaller improvement (or they perform the same as in the uniprocessor node case) and for the other two results differ for each protocol with *AURC-4* performing worse than *AURC-1* for one application.

For two out of the three regular applications performance improves significantly for both protocols. An exception to this is FFT that exhibits no significant improvement. From the irregular applications Barnes-rebuild, Volrend and Water-spatial benefit in both protocols, Barnes-nolocks and Water-nsquared do not exhibit any significant improvement and the last two application Radix and Raytrace behave differently under each protocol. Radix improves with *HLRC-4* but performs worse with *AURC-4* and Raytrace improves with *AURC-4* and exhibits no improvement with *HLRC-4*.

In doing this research, we found that modifying and restructuring applications a little—sometimes trivially and sometimes algorithmically—can go a long way toward improving the performance of SVM systems, with or without automatic update support. The tree-building in Barnes, the elimination of a lock and management of task queues in Raytrace, and the change in initial assignment of tasks in Volrend are examples. The performance impact is much greater than the tradeoff among these protocols.

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 would like to thank Hongzhang Shan for making available to us improved versions of some of the applications.

References

[1] A. Bilas, L. Iftode, D. Martin, and J. Singh. Shared virtual memory across SMP nodes using automatic update: Protocols and performance. Technical Report TR-517-96, Princeton, NJ, Mar. 1996.

[2] 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.

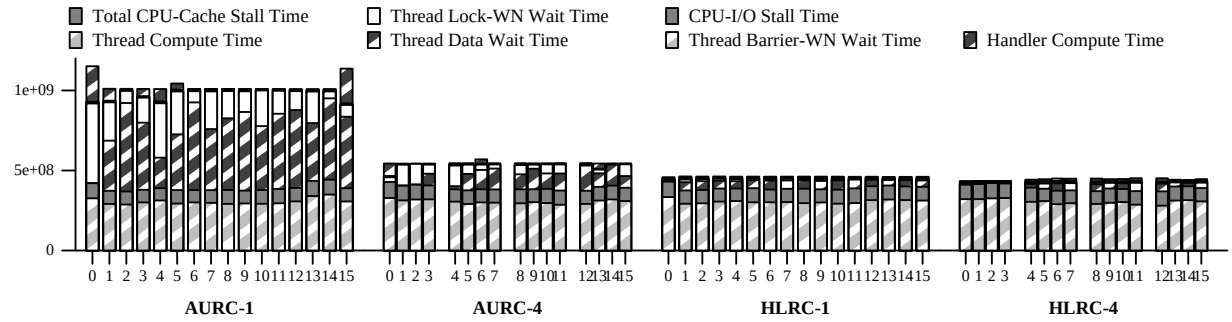

Figure 8: Cost breakdown for Raytrace for *AURC* and *HLRC*.

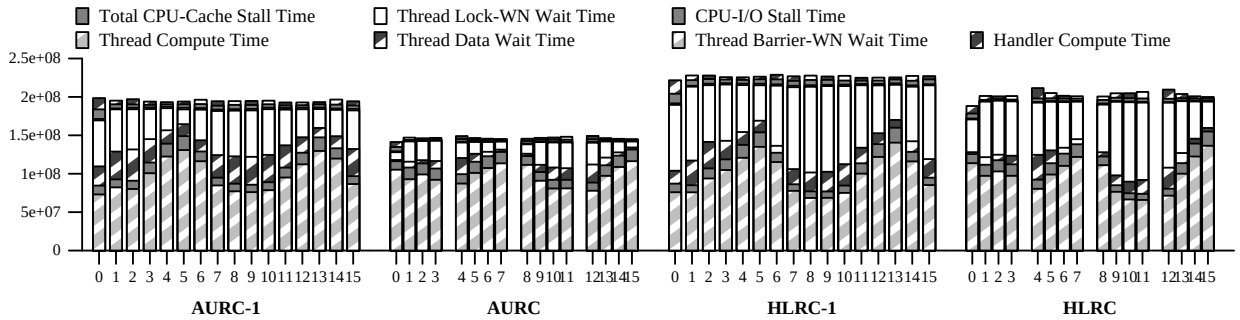

Figure 9: Cost breakdown for Volrend for *AURC* and *HLRC*.

[3] 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.

[4] 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.

[5] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of virtual memory-mapped communication on myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, April 1997.

[6] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[7] A. Erlichson, B. Nayfeh, J. Singh, and K. Olukotun. The benefits of clustering in shared address space multiprocessors: An applications-driven investigation. In *Supercomputing '95*, pages 176–186, 1995.

[8] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. 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.

[9] C. Holt, M. Heinrich, J. Singh, E. Rothberg, and J. Hennessy. The effects of latency, occupancy and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Stanford, Jan. 1995.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] S. Pakin, M. Buchanan, M. Lauria, and A. Chien. The Fast Messages (FM) 2.0 streaming interface. Submitted to Usenix'97, 1996.

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

[19] 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.

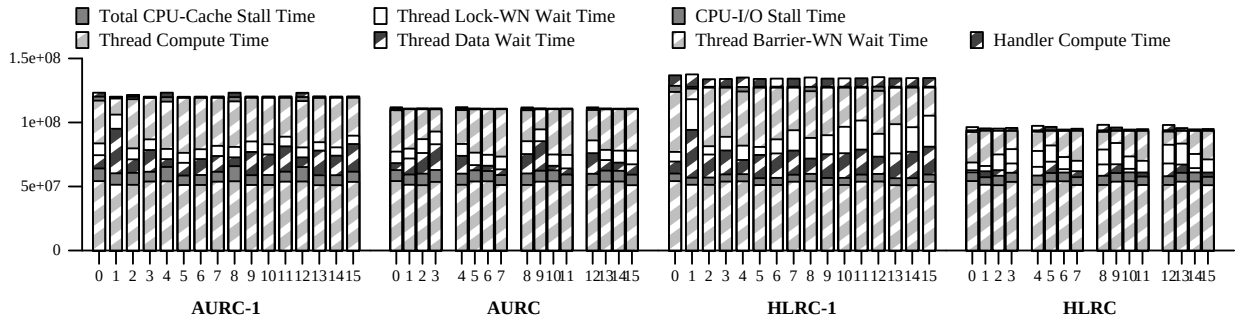

Figure 10: Cost breakdown for Water-spatial for *AURC* and *HLRC*.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

Appendix A: Detailed Performance Data

Table 3 compares *AURC-1* and *AURC-4*, and Table 4 compares *HLRC-1* and *HLRC-4* in more detail. For each application there are two columns. The first column presents the average percentile cost of each component of the execution time with respect to the thread compute time for the base case. The second column presents the change from the base case for the other protocol. In these tables *Barriers* refers to the total barrier cost, whereas Barrier Wait refers to the component of this cost until all processors reach the barrier (due to imbalances). Similarly *Locks* refers to the total lock cost and *Lock Wait* to the component up the point where the lock is released and can be granted to the next processor. Finally *PFetch Time* is the average cost of a page fault.

Cost Breakdown	FFT		LU-contiguous		Ocean-contiguous		Barnes-rebuilt		Radix		Volrend		Water-nsquared		Water-spatial	
Protocol	9%	-2%	1%	0%	12%	-11%	8%	-3%	4%	-2%	7%	-4%	5%	-3%	2%	-1%
I/O Wait	12%	-8%	2%	-1%	16%	-16%	6%	-2%	6%	-4%	6%	-4%	6%	-4%	2%	-2%
Barriers	29%	2%	91%	-33%	441%	-373%	28%	-11%	171%	296%	2%	-1%	24%	-3%	67%	-10%
Locks	0%	0%	0%	0%	1%	1%	162%	-26%	20%	243%	61%	-35%	41%	1%	15%	3%
Data Wait	109%	-26%	17%	-6%	139%	-131%	32%	-11%	336%	279%	36%	-25%	16%	-11%	29%	-9%
CPU Stall	89%	34%	42%	3%	203%	75%	12%	0%	10%	1%	17%	-1%	15%	0%	17%	-1%
Compute	100%	0%	114%	0%	100%	0%	100%	0%	100%	0%	133%	-8%	104%	0%	103%	0%
Barrier Wait	24%	4%	60%	-5%	344%	-304%	26%	-10%	164%	282%	1%	-1%	18%	2%	57%	-9%
Lock Wait	0%	0%	0%	0%	0%	1%	150%	-24%	14%	75%	53%	-31%	17%	11%	14%	3%
PFetch time	21625	28443	20695	9417	20069	20156	20850	8232	172213	1140274	24025	4480	23765	11311	28062	59742
# PFetches	2705	-1793	258	-135	4591	-4434	3243	-1740	2082	-1565	1136	-808	394	-298	511	-399
# Local Locks	0	0	0	0	0	5	1	511	1	7	0	223	0	748	0	6
# Remote Locks	0	0	0	0	15	-6	2226	-514	47	-7	430	-200	1170	-748	19	-5

Table 3: Changes in Protocol Costs from *AURC-1* to *AURC-4*.

Cost Breakdown	FFT		LU-contiguous		Ocean-contiguous		Barnes-rebuilt		Radix		Volrend		Water-nsquared		Water-spatial	
Protocol	57%	-31%	11%	-6%	82%	-79%	8%	4%	5%	2%	7%	2%	5%	0%	13%	-9%
I/O Wait	11%	-7%	2%	-1%	16%	-16%	8%	-3%	1233%	-1231%	9%	-4%	8%	-4%	3%	-3%
Barriers	42%	-10%	100%	-39%	426%	-355%	235%	-169%	2142%	-1849%	2%	-1%	23%	3%	66%	-32%
Locks	0%	0%	1%	-1%	2%	0%	337%	-76%	9%	-7%	118%	-15%	57%	-2%	30%	-9%
Data Wait	153%	-49%	29%	-14%	231%	-222%	50%	-21%	588%	-469%	30%	-12%	13%	-7%	36%	-26%
CPU Stall	63%	39%	37%	4%	181%	94%	13%	-1%	10%	1%	18%	1%	15%	0%	10%	3%
Compute	100%	0%	114%	0%	100%	0%	100%	0%	100%	0%	143%	6%	104%	0%	103%	0%
Barrier Wait	28%	1%	70%	-12%	364%	-319%	181%	-116%	2012%	-1741%	1%	0%	20%	5%	57%	-25%
Lock Wait	0%	0%	1%	-1%	1%	1%	289%	-62%	2%	-1%	79%	-4%	18%	7%	26%	-7%
PFetch time	32395	27367	31978	6890	33270	10448	28459	11698	298878	-38977	20077	17310	19687	18655	34089	5742
# PFetches	2541	-1569	285	-162	4647	-4492	3298	-1778	2056	-1584	1033	-699	382	-282	543	-408
# Local Locks	0	0	0	0	0	5	1	528	1	6	0	192	0	725	0	7
# Remote Locks	0	0	0	0	15	-6	2228	-531	47	-6	438	-188	1170	-725	17	-4

Table 4: Changes in Protocol Costs from *HLRC-1* to *HLRC-4*.