The Effect of Contention on the Scalability of Page-Based Software Shared Memory Systems

Eyal de Lara[†], Y. Charlie Hu[‡], Honghui Lu[†],
Alan L. Cox[‡], and Willy Zwaenepoel[‡]

[†] Department of Electrical and Computer Engineering
[‡] Department of Computer Science
Rice University, Hoston TX 77005, USA

Abstract. In this paper, we examine the causes and effects of contention for shared data access in parallel programs running on a software *distributed shared memory* (DSM) system. Specifically, we experiment on two widely-used, page-based protocols, Princeton's home-based lazy release consistency (HLRC) and TreadMarks. For most of our programs, these protocols were equally affected by latency increases caused by contention and achieved similar performance. Where they differ significantly, HLRC's ability to manually eliminate load imbalance was the largest factor accounting for the difference. To quantify the effects of contention we either modified the application to eliminate the cause of the contention or modified the underlying protocol to efficiently handle it. Overall, we find that contention has profound effects on performance: eliminating contention reduced execution time by 64% in the most extreme case, even at the relatively modest scale of 32 nodes that we consider in this paper.

1 Introduction

In this paper, we examine the causes and effects of contention for shared data access in parallel programs running on a software *distributed shared memory* (DSM) system. Specifically, we analyze the execution of a representative set of programs, each exhibiting a particular access pattern that causes contention. In each of these cases, to quantify the effects of contention on performance, we have either modified the application to eliminate the cause of the contention or modified the underlying protocol to efficiently handle that particular access pattern. Overall, we find that contention has profound effects on performance: eliminating contention reduced execution time by 64% in the most extreme case, even at the relatively modest scale of 32 nodes that we consider in this paper.

Our experiments are performed on a network of thirty-two single-processor nodes using both Princeton's home-based (HLRC) protocol [7] [11] and Rice's TreadMarks (Tmk) protocol [6]. Both are widely-used, page-based, multiple-writer protocols implementing Lazy Release Consistency (LRC) [5]. From our experiments, we derive three specific conclusions.

First, in comparing the results on 8 nodes to 32 nodes, we find that the effects of increasing contention for shared data are evident in the increasing latency to retrieve data. In the worst case, latency increased by 245%.

Second, in one case, the Barnes-Hut program from the SPLASH benchmark suite [9], the HLRC protocol handles contention more effectively than the Tmk protocol. It more evenly distributes the number of messages handled by each node.

Third, the distribution of the number of messages handled by each node is no less important than the total number of messages. For example, in Barnes-Hut, eliminating the message load imbalance under Tmk (through protocol modifications), brought Tmk's performance to the same level as HLRC's, even though Tmk sends 12 times more messages than HLRC.

The rest of this paper is organized as follows. Section 2 provides an overview of TreadMarks and HLRC's multiple-writer protocols. Section 3 discusses the sources of contention in greater detail and defines the notion of protocol load imbalance. Section 4 details the experimental platform that we used and the programs that we ran on it. Section 5 presents the results of our evaluation. Section 6 compares our results to related work in the area. Finally, Section 7 summarizes our conclusions.

2 Background

2.1 Lazy Release Consistency

The TreadMarks (Tmk) protocol [5] and the Princeton home-based (HLRC) protocol [11] are multiple-writer implementations of lazy release consistency (LRC) [5].

Lazy release consistency (LRC) [5] is an algorithm that implements the release consistency (RC) [4] memory model. RC is a relaxed memory model in which ordinary accesses are distinguished from synchronization accesses. There are two types of synchronization access: acquire and release. Roughly speaking, acquire and release accesses are used to implement the corresponding operations on a lock. In general, a synchronization mechanism must perform an acquire access before entering a critical section and a release access before exiting. Essentially, the benefit of RC is that it allows the effects of ordinary memory accesses to be delayed until a subsequent release access by the same processor.

The LRC algorithm [5] further delays the propagation of modifications to a processor until that processor executes an acquire. Specifically, LRC insures that the memory seen by the processor after an acquire is consistent with the happened-before-1 partial order [1], which is the union of the total processor order of the memory accesses on each individual processor and the partial order of release-acquire access pairs.

2.2 TreadMarks and Home-based LRC

The main difference between Tmk and HLRC is in the location where updates are kept and in the way that a processor validates its copy of a page. In Tmk,

processors validate a page by fetching diffs from the last writer or writers to the page. In HLRC, every page is statically assigned a home processor by the programmer. Writers flush their modifications to the home node at release time. To validate a page a processor requests a copy of the entire page from the home. The difference between the protocols is the most evident for falsely shared pages. The home-based protocol uses significantly fewer messages as the number of falsely sharing readers and writers increases. Specifically, for R readers and W writers, the home-based protocol uses at most $2W + 2R$ messages and the Tmk protocol uses at most $2WR$ messages. On the other hand, Tmk's reliance on diffs to validate pages can result in substantial bandwidth savings.

3 Contention and Protocol Load Imbalance

In this section, we introduce the concepts of contention and protocol load imbalance. We give some intuition for the characteristics of Tmk and HLRC that may lead to contention and protocol load imbalance.

3.1 Contention

We define *contention* as simultaneous requests on a node. In our platform, contention can be attributed to limitations in the node or the network. In the former case, the time that the node requires to process a request is longer than it takes for the next request to arrive. In the latter case, the node fails to push out responses fast enough due to bandwidth limitations in the network link. Most systems, under this condition, wait for the interface to free an entry in its output queue. Contention is said to be single-paged, when all requests are for the same page, or multi-paged, when distinct pages are being requested from the same node.

3.2 Protocol Load Imbalance

We refer to the work performed to propagate updates as protocol load (PL). We then define PL imbalance as the difference in PL across the nodes of the system. Under Tmk, PL reflects time spent servicing requests. For HLRC, it also includes time spent pushing modifications to home nodes. As Tmk and HLRC differ in the location where updates are kept, the protocols may have a different effect on the PL balance.

To illustrate the difference, consider a multi-page data structure that has a single writer followed by multiple readers. Both Tmk and HLRC handle each page of the data structure in similar ways. Each reader sends a request message to the processor holding the latest copy of the page. That processor sends back a reply message containing either the changes to the page in Tmk, or a complete copy of the page in HLRC. Where the protocols differ is in the location of the updates. In Tmk, the last writer is the source of the updates, while in HLRC the updates are kept at the home nodes. Hence, Tmk places the load of distributing

multiple copies of the entire data structure on the last writer. In contrast, in HLRC, a clever home assignment may result in a more balanced distribution of the load among the nodes.

4 Experimental Environment

4.1 Platform

We perform the evaluation on a switched, full-duplex 100 Mbps Ethernet network of thirty-two 300 MHz Pentium II-based uniprocessors running FreeBSD 2.2.6. On this platform, the round-trip latency for a 1-byte message is 126 microseconds. The time to acquire a lock varies from 178 to 272 microseconds. The time for a 32-processor barrier is 1,333 microseconds. The time to obtain a diff varies from 313 to 1,544 microseconds, depending on the size of the diff. The time to obtain a full page is 1,308 microseconds.

Table 1. Program Characteristics.

Program	Size, Iter.	Seq. Time (sec.)	Home Distr.
SOR	8kx4k, 20	72.23	Block
3D FFT	7x7x7, 10	101.35	Block
Gauss	4096, 1	477.68	Cyclic
Barnes-Hut	65536, 3	125.69	Block

4.2 Programs

We use four programs: Red-Black SOR and Gaussian Elimination are computational kernels that are distributed with TreadMarks; 3D FFT is from the NAS benchmark suite [2]; and Barnes-Hut is from the SPLASH benchmark suite [9].

Table 1 lists for each program the problem size, the sequential execution time, and the (static) assignment strategy of pages to homes for HLRC. These strategies were selected through considerable experimentation and yield the best performance.

Red-Black SOR is a method for solving partial differential equations by iterating over a two-dimensional shared array. Each processor is assigned a band of rows. Communication is limited to the elements in boundary rows.

Gauss implements Gaussian Elimination with partial pivoting on linear equations stored in a two-dimensional shared array. The computation on the rows is distributed across the processors in a round-robin fashion. In addition to the shared array for storing the coefficients of the equations, an index variable storing the pivot row number is also shared.

3D FFT solves a partial differential equation using three-dimensional forward and inverse FFT. The program has two shared data structures, an array of

elements and an array of checksums. The computation is decomposed so that every iteration includes local computation and a global transpose.

Barnes-Hut performs an N-body simulation using the hierarchical Barnes-Hut method. There are two shared data structures: a tree used to represent the recursively decomposed subdomains (cells) of the three-dimensional physical domain containing all the particles, and an array of particles corresponding to the leaves of the tree. Every iteration rebuilds the tree on a single node followed by a parallel force evaluation of the particles, during which most of the tree is read by all nodes.

5 Results

In this section, we present the results of running each of the programs on 8 and 32 processors. Figures 1 and 2 present the speedups and a breakdown of the execution time for each of the programs. Table 2 shows the variation in average response time. It also provides the average number of page update requests per node, the total data transferred, and the global message count. Figures 3, 5, 7, 9, and 10 show the response time histograms. Finally, Figures 4,6, 8, 11, and 12 show the protocol load histograms. For brevity, we only provide plots for HLRC when the results differ significantly from Tmk (i.e. Barnes-Hut). For most applications, we only present results for runs on 32 nodes. The response time and protocol load histograms for 8 node runs were all similar to the contention free Red-Black plots (Figures 3 and 4), as in our platform at least, contention effects become significant only at a scale larger than 8 nodes. Some of the above figures and tables also present results for optimized versions of Tmk, which will be discussed in Section 5.2.

The breakdown of the execution time for each program (Figure 2) has three components: `memory` is the time spent waiting to update a page; `synchro` is time waiting for synchronization to complete; and `computation` includes all other time.

The response time histograms plot the time necessary to fulfill every request sent by every node in the system. On the horizontal axis is time in hundreds of microseconds to complete a request. On the vertical axis is the percentage of requests sent by all nodes that required a given amount of time to complete.

The protocol load histograms plot the time spent servicing remote requests by each node. Each bar is composed of three elements: `communication` corresponds to time spent receiving and decoding requests, as well as sending replies; `diff` corresponds to time spent building diffs; and `spin` corresponds to time spent waiting for the network interface to free an entry in its output queue. For Gauss, the protocol load histograms reflect only the time elapsed during the 8th iteration, instead of the entire execution. This finer resolution is necessary to show the imbalance in protocol load that occurs during an iteration.

Red-Black SOR is included as a control program. It is a program that achieves good scalability (with a speedup of 25.7 on 32 processors) and does not suffer from increases in response time, as can be appreciated by the small memory

component in Figure 2, as well as by the minor variations in the shape of the response time histograms of Figure 3. Furthermore, it exhibits little contention and has a good balance of protocol load, which is evident in the similar size of the bars of the histogram in Figure 4. For the rest of this section, we will use the Red-Black SOR response time measurements, response time histograms, and the protocol load histograms to illustrate how response time, response time histograms, and protocol load histograms should look in the absence of contention and protocol load imbalances.

We argue that the increase in access miss time experienced by our programs is largely a result of the increase in latency for individual requests due to contention and not solely the result of an increased number of request messages.

This trend is most evident for 3D FFT and Gauss. In 3D FFT, the average number of requests per node drops significantly as the size of the cluster increases. In Gauss, it increases moderately (10.7%). Both programs, however, experience sharp increases in response time. The average request latency increases for 3D FFT and Gauss by 46% and 245%, respectively (see Table 2).

The rest of this section is divided into two parts. First, we talk about the various types of contention exhibited by our programs and how they increase the latency of individual requests. Second, we quantify the effect that contention has on the programs' speedups by eliminating contention manually or automatically.

Table 2. Average response time, average number of pages updated per node, total data transfered and total message count of Tmk, HLRC, and Tmk with optimizations.

		Avg. resp. time (micro sec.)		Avg. per node updates		Data (MBytes)		Messages (thousands)	
Application	Protocol	8	32	8	32	8	32	8	32
SOR	Tmk	1592.33	1668.53	100	125	6	27	2	10
	HLRC	1400.43	1474.77	83	106	7	34	2	10
3DFFT	Tmk	2017.00	2963.19	4041	1125	265	295	66	91
	HLRC	1988.77	2918.72	4041	1125	265	297	65	74
	Opt.	1668.26	1870.31	4041	1125	265	295	66	91
Gauss	Tmk	2595.65	8954.12	8957	9910	357	1581	201	888
	HLRC	2882.89	8640.11	8957	9910	562	2586	201	888
	HLRC	1036.02	1036.02	8957	9910	46	60	73	268
Barnes	Tmk	1630.94	5534.01	2442	2026	130	488	144	1535
	HLRC	1655.81	2033.12	2072	1930	154	529	34	129
	Opt.	1510.39	1734.28	2442	2026	130	448	144	1535

5.1 Types of Contention

3D FFT suffers from multi-page contention. In 3D FFT, processors work on a contiguous band of elements from the shared array. Since HLRC assigns homes in block fashion, in both protocols the processor modifying the band will be

Fig. 1. Speedup of applications for Tmk, HLRC, and Tmk Optimized.

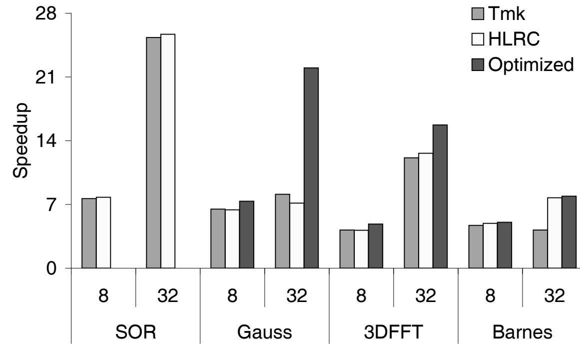


Fig. 2. Execution time breakdown for Tmk, HLRC, and Tmk Optimized.

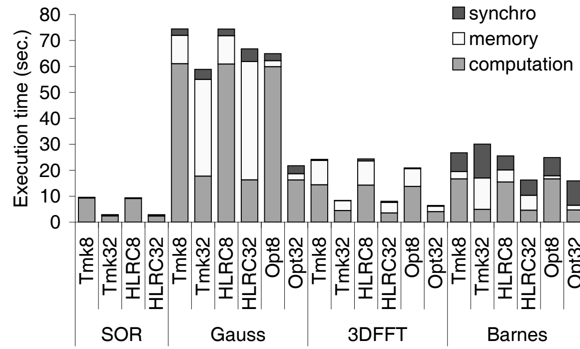


Fig. 3. RB-SOR response time for Tmk with 8 (a) and 32 (b) nodes.

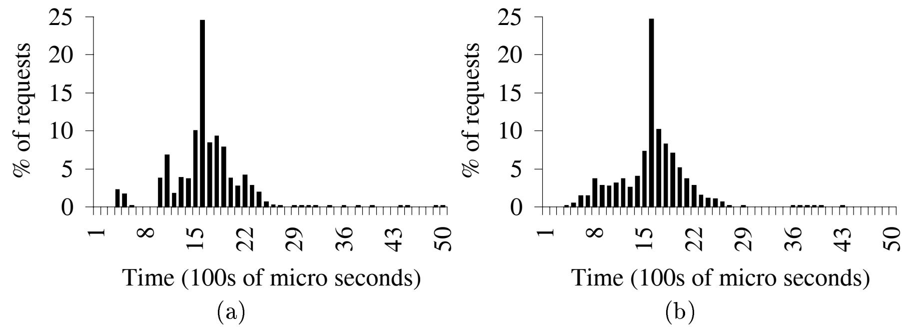


Fig. 4. RB-SOR protocol load for Tmk with 8 (a) and 32 (b) nodes. The plots show the time spent in communication, creating diffs, and waiting for the network interface.

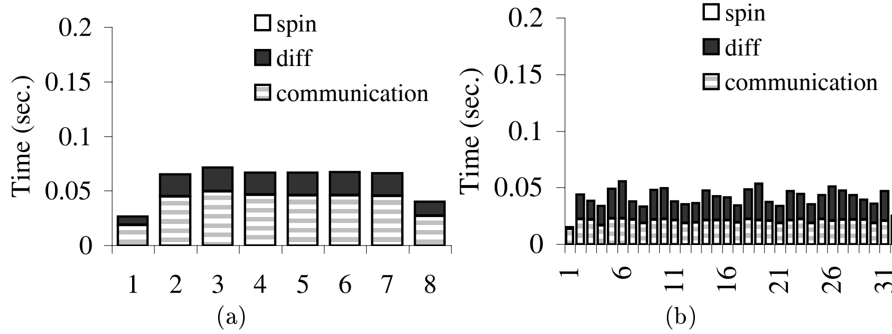


Fig. 5. 3D FFT response time for Tmk (a) and Tmk Optimized (b) on 32 nodes.

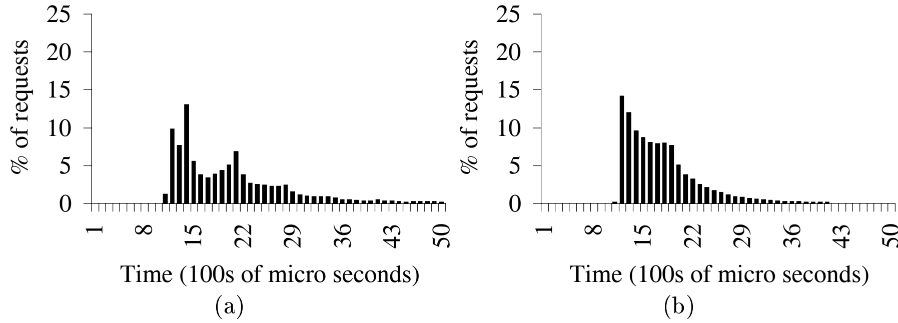


Fig. 6. 3D FFT protocol load for Tmk (a) and Tmk Optimized (b) on 32 nodes.

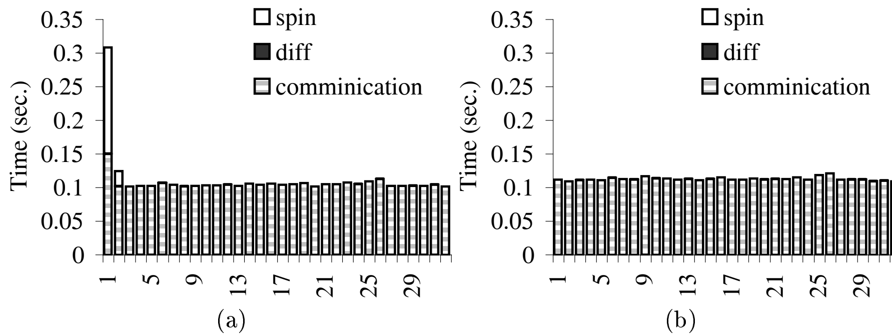


Fig. 7. Gauss response time for Tmk (a) and Tmk Optimized(b) on 32 nodes.

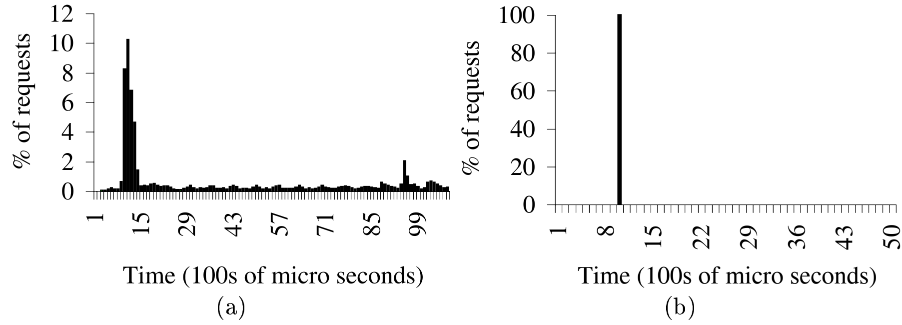


Fig. 8. Gauss protocol load for Tmk (a) and Tmk Optimized (b) on 32 nodes.

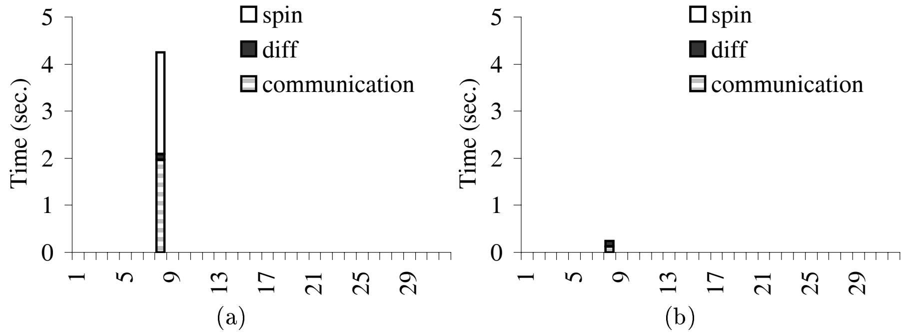


Fig. 9. Barnes-Hut response time for Tmk (a) and Tmk Optimized (b) on 32 nodes.

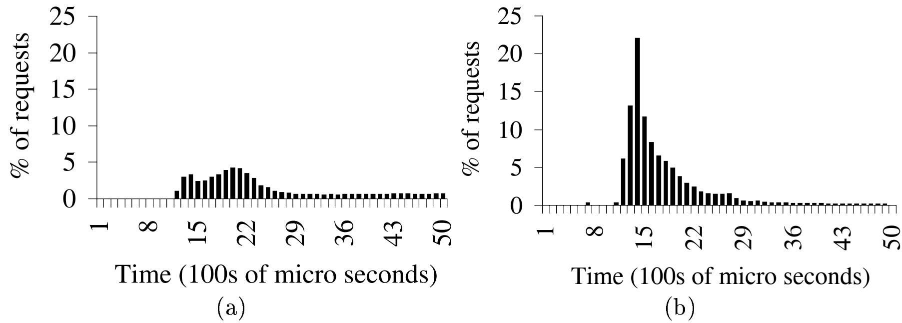


Fig. 10. Barnes-Hut response time for HLRC on 32 nodes.

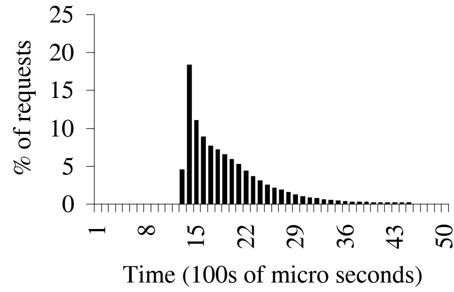


Fig. 11. Barnes-Hut protocol load for Tmk (a) and Tmk Optimized (b) on 32 nodes.

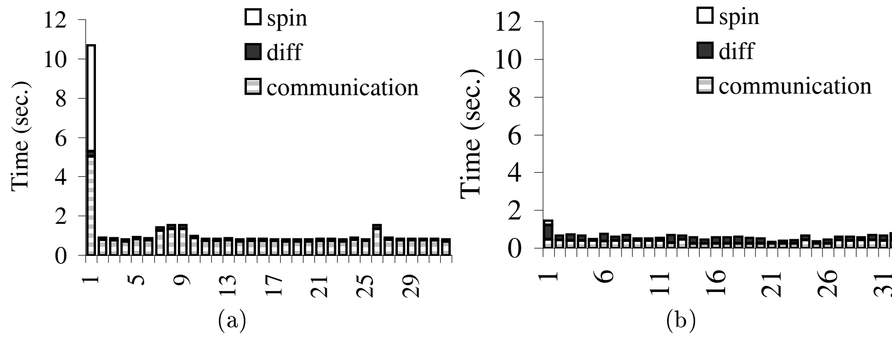
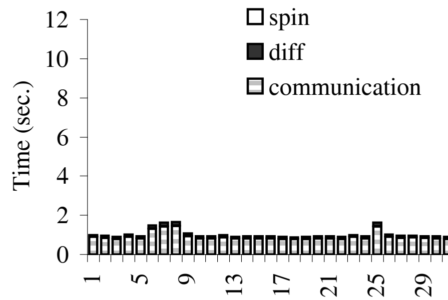


Fig. 12. Barnes-Hut protocol load for HLRC on 32 nodes.



the one hosting the update. The computation on the 3D arrays is partitioned along one axis among the processors. As a result, the global transpose leads to all processors trying to read some pages first from processor 0, then some pages from processor 1, and so on, resulting in a temporal contention at a single processor at a time. The effects of contention are visible in the response time histogram in Figure 5(a). When we compare this plot to the Red-Black SOR response time histogram (Figure 3(b)), we see that it consists of shorter bars that are shifted to the right, which suggest a significant increase in response time.

The effects of contention are also visible in the protocol load histogram (Figure 6(a)), which shows an imbalance in protocol load, with the two left most processors having significant higher loads. This imbalance is somehow surprising, since during an iteration, all nodes service the same number of requests. Closer consideration, however, reveals that, as mentioned above, at the start of an iteration all processors, at once, try to fetch data from processor 0. The simultaneous requests create contention on this processor, as is evidenced by the high spin component in the plot. As processor 0 services these request, however, it staggers the replies, implicitly ordering and distributing them over time. The staggering of replies produces less contention on the next round of communications, as is evidenced by the smaller wait component of the next processor. Processor 1 further accentuates this trend.

Gauss suffers from single-page contention. The access pattern for the pivot row is single-writer/multiple-reader. There are only two such pages for our problem size, but their producer changes on every iteration. The severity of contention in Gauss within each interval is reflected in the large increase in response time shown in Table 2 (almost 245%) and in the response time histogram of Figure 7(a). The large degradation in response is reflected in the shape of this plot, which shows bars that are more spread out, shorter, and shifted to the right, compared to the contention-free Red-Black SOR plot (Figure 3(b)). The increase in response is attributed to the large imbalance in protocol load shown in 8(a), where all the protocol load is concentrated in processor 8, the last writer to the pivot row and pivot index. The large spin component in this Tmk plot results from the backlog of large reply messages containing the pivot row and index in the network interface's output queue.

Barnes-Hut suffers from single-page and multi-page contention. The tree and the array of particles in Barnes-Hut suffer from single-page contention when read in the same order by multiple nodes. Additionally, multi-page contention occurs when updates for multiple parts of the data structures reside at any given node. For Barnes-Hut, there is a significant difference in the rate of latency increase between Tmk and HLRC. While the average response time for HLRC increases slowly, it almost doubles with an increase in the number of nodes for Tmk (see Table 2). This is supported by the rapid dispersion of the bars in the Tmk response time histogram (Figure 9(a)) while the HLRC response time histogram (Figure 10) resembles the contention free Red-Black SOR plot (Figure 3(b)). We argue in the next section that differences in protocol load balance account for the disparity in response time.

5.2 Quantifying the Effects of Contention

We quantify the effects of contention on our programs by manually tuning the application or the protocol to remove the sources of contention. For each application we describe the contention removal technique we used and discuss the effects on the speedup, load balance and response time. Figures 1 and 2 also include the speedups and breakdown of execution time for the optimized version of Tmk, while Table 2 shows the variations in average response time. Finally, Figures 5(b), 7(b), and 9(b) show the response time histograms, and Figures 6(b), 8(b), and 11(b) show the protocol load histograms.

3D FFT The contention in 3D FFT can be eliminated by carefully restructuring the transpose loop. By staggering the remote accesses of different consumers such that they access pages on different producers in parallel, the speedup of 3D FFT using Tmk is improved from 12.62 to 15.73 on 32 nodes. The optimization achieves a reduction in response time of 27% (see Table 2), and response time and protocol load histograms that resemble the contention-free case (compare Figures 5(b) and 6(b) to Figures 3(b) and 4(b)).

Gauss The contention in Gauss can be eliminated by broadcasting the pages containing the pivot row and pivot index. Using a manually inserted broadcast improves Tmk speedup for Gauss from 8.11 to 22.05 on 32 nodes. The optimization achieves a 88% reduction in response time (see Table 2) and 94% in protocol load (compare Figures 8(a) and (b)). Figure 7(b) shows all responses clustered in a single bar at the 1000 microseconds bin. In fact, response times vary within tens of micro seconds, but because of the scaling effects this is not visible. The average response time of 1036 microseconds, which is lower than the contention free time for updating a page, results from the eliminating the request message.

Barnes-Hut Protocol load imbalance accounts for the difference in speedup (4.18 vs. 7.73) and request latency experienced between Tmk and HLRC for Barnes-Hut.

To prove this claim we added *striping* to the Tmk protocol. Striping reduces contention by automatically eliminating protocol imbalances created by multi-page data structures with a single writer and multiple readers. Striping identifies these data structures and automatically distributes them to other processors (i.e. new homes) at the next global synchronization point. As a result, the writer's effort on behalf of each page that it off-loads is limited to constructing and sending the diff to a single processor. The processor receiving this diff is then responsible for servicing the requests from all of the consumers. Overall, neither the number of messages nor the amount of data transferred is reduced, but the average time that a reader waits for its requested diff drops.

The high load imbalance in Tmk (Figure 11(a)) results from the distribution of the updates for the tree in Barnes-Hut. In Tmk updates are always fetched from the last writer, hence processor 0 has to supply all updates to the tree in

Barnes-Hut. As processor 0 gets overwhelmed by requests for updates to the tree, it has to spend an increasing portion of its execution time servicing requests. On the 32 node cluster this time accounts for 35.6% of the total execution time. Of that time, 50.42% is spent blocked waiting for the network interface to clear the output queue.

In contrast (Figure 12), HLRC greatly alleviates the contention for reading the tree by spreading the homes for these data structures across the processors. Specifically, if the tree covers n pages and every processor reads the whole tree, then Tmk requires processor 0 to service $(p-1)*n$ page requests. HLRC instead distributes the tree in $n*(p-1)/p$ messages. After that the load of servicing the tree requests is evenly distributed.

Tmk with striping eliminates protocol load imbalance caused by requests from multiple processors to processor 0 in order to obtain the tree data and achieves response time (Figure 9(b)), protocol load imbalances (Figure 11(b)), and speedups (Figure 1) that are comparable to HLRC.

The results from Tmk with striping demonstrate that (for this application, at least) there is a relationship between protocol balance and response time: as protocol imbalance grows, so does response time. This is an expected result; when the proportion of updates originating from any given node grows, the likelihood of simultaneous requests (i.e., contention) on that node increases. Furthermore, these results show that the distribution of the messages is no less important than the number of messages. For example, Tmk with striping and HLRC transfer roughly equal amounts of data and have identical message distributions. Although Tmk with striping sends 12 times more messages than HLRC, they achieve the same speedup.

Nonetheless, Tmk/stripe and HLRC still suffer from contention, as shown by the difference in average response time when compared to the control (1734.28 for Barnes-Hut vs. 1474.77 for SOR). This contention results from the several nodes transversing the data structures in the same order. As various parts reside in the same node, simultaneous requests for updates may still reach the nodes. That is, the distribution of updates to various processors does not eliminate contention for the data, it only makes it less frequent on a particular node.

6 Related Work

A large number of software shared memory systems have been built. Many of the papers looking at the performance of software DSM on thirty-two or more processors have used SMP-based nodes [7, 8, 10]. Thus, the actual number of nodes on the network is typically a factor of two to eight less than the number of processors. Because requests for the same page from multiple processors within a node are combined into one, the load on the processor(s) servicing the page may not be as high as when the number of nodes in the network equals the number of processors. These studies have ignored the effects of contention and protocol load imbalance, as these become significant only on a network with a large number of nodes.

Two papers that look at large networks of uniprocessors are Zhou et al. [11] and Bal et al. [3]. Zhou et al. evaluated the home-based lazy release consistency protocol against the basic LRC protocol on an Intel Paragon. The relatively large message latency, page fault, and interrupt times compared with memory and network bandwidth, and the extremely high cost of diff creation on the Paragon architecture are uncommon in modern parallel platforms and favor the HLRC protocol. The high diff cost led the authors to conclude that the performance gap between Tmk and HLRC results from the vast differences in message count. We show that the performance gap results, instead, from the difference in protocol load balance.

Bal et al. evaluated Orca, an object-based distributed shared memory system, on a Myrinet and a Fast Ethernet network of 32 200MHz Pentium Pro computers. The object-based DSM system decreases the number of messages and data resulting from reduced false sharing at the cost of the programmer's extra effort to explicitly associate shared data structures with objects. Objects with low read/write ratio are stored in a single processor, while those with high read/write ratio are replicated on all processors using multicast.

7 Conclusions and Discussion

We show that memory latency increases due to contention and protocol load imbalances are a significant obstacle to the scalability of software DSM systems. For example, in one case, memory latency increased by 245% as the number of nodes increased from 8 to 32. Furthermore, there is relationship between contention and protocol load balance: Higher protocol load imbalance usually results in increased contention. Intuitively, an increase in the proportion of data distributed from a node increases the likelihood of simultaneous requests to that node. Thus, we argue that contention reduction and protocol load balancing should be considered, in addition to message reduction, by designers of scalable DSM systems.

Overall, on our platform, contention has a profound effect on performance even at the modest scale of 32 nodes. In 3D FFT, the contention was caused by multiple processors accessing different single-writer/single-reader pages at the same time. By manually restructuring the transpose loop, we found that the execution time could be reduced by 20% on 32 nodes. In Barnes-Hut, we found that protocol load imbalance caused HLRC to outperform Tmk by 84% on 32 nodes. Eliminating the load imbalance brings the performance of Tmk on par with HLRC. Finally, in Gauss, the contention is due to a single-writer/multiple-reader sharing pattern. In this case, two or three pages are read at a time, and each page is only read once by each processor (other than its producer). Using a manually inserted broadcast, we were able to reduce the execution time on 32 nodes by 64%.

In our future work, we hope to automate the use of broadcast and load balancing to achieve these results transparently.

References

1. S.V. Adve and M.D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.
2. D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, August 1991.
3. H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M.F. Kaashoek. Performance evaluation of the Orca shared object system. *ACM Transactions on Computer Systems*, 16(1), February 1998.
4. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.
5. P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.
6. P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.
7. R. Samanta, A. Bilas, L. Iftode, and J.P. Singh. Home-based SVM protocols for SMP clusters: design and performance. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.
8. D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A low overhead software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.
9. J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.
10. R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.
11. 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 Second USENIX Symposium on Operating System Design and Implementation*, pages 75–88, nov 1996.