The Relative Importance of Concurrent Writers and Weak Consistency Models

Peter J. Keleher
Department of Computer Science
University of Maryland
College Park, MD 20742-3255
keleher@cs.umd.edu

Abstract

This paper presents a detailed comparison of the relative importance of allowing concurrent writers versus the choice of the underlying consistency model. Our comparison is based on single- and multiple-writer versions of a lazy release consistent (LRC) protocol, and a single-writer sequentially consistent protocol, all implemented in the CVM software distributed shared memory system.

We find that in our environment, which we believe to be representative of distributed systems today and in the near future, the consistency model has a much higher impact on overall performance than the choice of whether to allow concurrent writers. The multiple writer LRC protocol performs an average of 9% better than the single writer LRC protocol, but 34% better than the single-writer sequentially consistent protocol. Set against this, MW-LRC required an average of 72% memory overhead, compared to 10% overhead for the single-writer protocols.

1 Introduction

Sophisticated page-based distributed shared memory (DSM) systems achieve high performance through a combination of weak memory models and multiple-writer protocols. Although these techniques are often cited as co-equal factors in good performance, no study has quantified their individual contributions to overall performance.

Hardware shared memory systems typically use *single-writer* protocols to keep caches coherent. These protocols allow multiple readers to access a given datum simultaneously, but require a writer to gain ownership and exclusive access to a page before modifying it. Such protocols are relatively straightforward to implement because all copies of a given datum are identical. Faults can be satisfied by retrieving a new copy of the data from any other processor that has a copy. Unfortunately, simplicity comes at the expense of message traffic. Before a datum can be modified, all other copies must typically be invalidated, requiring those processors to take *access faults* and retrieve new copies of the page if they are still accessing it.

The unit of sharing in a page-based DSM is a virtual memory page; much larger than the cache lines used in hardware shared memory systems. The larger coherence granularities used by DSMs cause them to suffer increased coherence traffic because of *false sharing*, or simultaneous accesses by different processors to unrelated parts of the datum.

Software DSMs such as TreadMarks [9] Munin[2], and CarlOS [12], alleviate the effects of false sharing by supporting multiple-writer protocols. These protocols allow multiple nodes to simultaneously modify different sections of the same page. The modifications are later reconciled by creating summaries of each of the modifications, called *diffs* [2], and applying the diffs to all copies of the page.

The advantages of multiple-writer protocols for software DSMs are clear: the effects of false sharing are minimized because processors can make local decisions to write valid page without communicating with other processors. The disadvantages are also clear: protocols are more complex, a diffing mechanism must be used to merge multiple modifications to the same page, and the memory overhead is high. To date, however, there has not been a careful analysis of this tradeoff.

This paper presents such an analysis in the context of the Coherent Virtual Machine (CVM)[10] software DSM. CVM is a portable, user-level follow-on to the TreadMarks DSM. CVM was specifically written in a modular fashion in order to allow fair comparisons to be made between different protocols. CVM provides a set of basic classes that implement a generic protocol, lightweight threads, and network communication, complete with efficient end-to-end protocols that add reliability to the base UDP protocol. Additional protocols are created by deriving new classes from the base protocol class.

For the comparison described in this paper, we implemented three protocols: a multiple-writer LRC protocol

(MW-LRC), a single-writer LRC protocol (SW-LRC), and a single-writer SC protocol (SW-SC). SW-LRC and MW-LRC are single- and multiple-writer protocols that implement the lazy release consistent (LRC) [8] memory model. While SW-LRC requires processors to gain ownership of a page before modifying it, the lazy protocol allows any number of readers to co-exist with a single writer. We compare the performance of the two protocols in order to gauge the importance of allowing multiple simultaneous writers. LRC was chosen as the consistency model because it allows false sharing to be hidden more effectively than other memory models.

We then compare the performance of SW-LRC to that of a carefully tuned sequentially consistent protocol (SW-SC) in order to gauge the importance of the consistency model relative to the choice of single or multiple-writer protocols. Other that the fact that the weaker memory semantics allow SW-LRC to delay coherency actions longer than SW-SC, the protocols are quite similar, and in fact share much code.

Our comparisons show that the performance of SW-LRC trails MW-LRC's by nine percent overall, but SW-LRC actually averages three percent *better* than MW-LRC for six of the eight applications in our study. This result has several root causes. First, write-write false sharing is much less common than read-write[4] sharing. Second, the weak memory model allows even SW-LRC to hide most of the effects of read-write false sharing by allowing multiple readers to co-exist with a single writer. Finally, communication in this environment has a high startup cost for each message, while the per-byte cost is relatively low. The single-writer protocol has higher bandwidth requirements than the multiple-writer protocol because it transfers entire pages instead of diffs. However, the high startup cost on messages means that the number of messages is usually more important than the total amount of data sent.

The aggregate effect of these differences is to slightly favor the single-writer protocol for those applications that do not write-share pages. However, SW-LRC's performance, and indeed the performance of any single-writer protocol, can drop drastically in the presence of write-sharing.

The performance gap between the two single-writer protocols averages 34%, much larger than between the two LRC protocols. We conclude from this result that the choice of memory model has a larger effect on performance than the choice of a single- or multiple-writer protocol.

While this study presents data on only a single point in the spectrum of possible system characteristics, our testbed is typical of current systems. We also expect future technology trends to favor the single-writer protocol. The widening disparity between memory bandwidth and processor speed will increase the cost of diff creation relative to network communication. The latter is quickly growing less costly as current architecture research has focused on the creation of zero-copy, memory-mapped network interfaces that make communication latency independent of the memory hierarchy.

Section 2 describes the CVM system and the two protocols in detail. Section 3.1 describes our experimental setup and presents detailed cost breakdowns of the component parts of both protocols. Section 3 describes the overall performance of both protocols on a suite of shared memory programs, and relates their performance back to the application characteristics. Finally, in Section 4, we present our conclusions.

2 CVM and Protocols

This section provides a brief overview of lazy release consistency, a description of the CVM system in which the protocols are implemented, and a description of the protocols themselves.

2.1 Lazy Release Consistency

Lazy Release Consistency [8] is a variant of *eager* release consistency (ERC) [6], a relaxed memory consistency that allows the effects of shared memory accesses to be delayed until selected synchronization accesses occur. Simplifying matters somewhat, shared memory accesses are labeled either as *ordinary* or as *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. Acquires and releases may be thought of as conventional synchronization operations on a lock, but other synchronization mechanisms can be mapped on to this model as well. Essentially, ERC requires ordinary shared memory accesses to be performed only when a subsequent release by the same processor is performed. ERC implementations can delay the effects of shared memory accesses as long as they meet this constraint.

Under LRC protocols, processors further delay performing modifications remotely until subsequent acquires by other processors, *and* the modifications are only performed at the other processor that performed the acquire. The central intuition of LRC is that competing accesses to shared locations in correct programs will be separated by synchronization. By deferring coherence operations until synchronization is acquired, we can piggyback consistency information on existing synchronization messages. In comparison to ERC, LRC generally improves performance by eliminating consistency messages, further hiding the effects of false sharing, and enabling new optimizations, such as piggybacking data movement on synchronization.

We use lazy release consistent protocols for this study because they delay consistency actions longer than other protocols, and therefore are more successful at hiding the effects of false sharing as well.

2.2 CVM

The Coherent Virtual Machine (CVM)[10] system is a software DSM that supports multiple protocols and consistency models. Like commercially available systems such as TreadMarks [9], CVM is written entirely as a user-level library and runs on most UNIX-like systems. Unlike TreadMarks, CVM was created specifically as a platform for protocol experimentation.

The system is written in C++, and opaque interfaces are strictly enforced between different functional units of the system whenever possible. The base system provides a set of classes that implement a generic protocol, lightweight threads, and network communication. The latter functionality consists of efficient, end-to-end protocols built on top of UDP.

New shared memory protocols are created by deriving classes from the base `Page` and `Protocol` classes. Only those methods that differ from the base class's methods need to be defined in the derived class. The underlying system calls protocol hooks before and after page faults, synchronization, and I/O events take place. Since many of the methods are inlined, the resulting system is able to perform within a few percent of a severely optimized system, TreadMarks [9], running a nearly identical protocol. However, CVM was designed to take advantage of generalized synchronization interfaces, as well as to use multi-threading for latency toleration. We therefore expect the performance of the fully functional system to improve over the existing base. In order to simplify the comparison process, however, we do not use either of these techniques in this study.

2.3 Protocols

2.3.1 SW-SC

SW-SC is an implementation of a page-based, single-writer protocol. Either a single writer or multiple readers may have copies of a given page at any one time. Page ownership is migrated to each processor as a copy of the page is requested, regardless of whether the request was for a write or a read copy. Ownership could be retained by the current owner while servicing read faults, but the status of the owner's page still needs to be downgraded to read-only. We chose to migrate ownership as an optimization favoring migratory data.

Like Mirage [5], we address the *ping-pong* problem by guaranteeing a processor a minimum quantum of time with any newly retrieved page before it can be invalidated by another processor. The ping-pong problem occurs when multiple processors simultaneously attempt to write the same page. A processor may request and gain ownership of a page, but receive an ownership request for the page from another processor before the fault handler exits. In this case, the modification (however minor), is not completed before ownership is lost, and ownership must be re-requested.

Since modifications are often small, even a very small quantum completely hides the problem in most cases. We found that a quantum of two microseconds sufficed for both single writer protocols. Execution times for SW-SC when the guaranteed quantum was not used went up by a factor of four or five for some applications.

2.3.2 SW-LRC

SW-LRC differs from SW-SC in that a single owner can co-exist with multiple readers. Pages only need to be invalidated when a processor receives a write notice via synchronization.

The other major difference is a consequence of this choice. Since servicing read faults does not require the owner to downgrade its writable copy to a read-only copy, we do not migrate ownership on read misses, and writable pages are not downgraded to read-only copies. This optimization improves performance for all the applications that we tested.

2.3.3 MW-LRC

MW-LRC differs from SW-LRC in that multiple writers are allowed to concurrently modify the same page. These concurrent modifications are merged using *diffs* to summarize the updates. A diff is created by performing a page-length comparison between the current contents of the page and a *twin* of the page that was created at the first write access. If each concurrent writer summarizes its modifications as a diff, the system can create a copy that reflects all modifications by applying the concurrent diffs to the same copy. Concurrent diffs only overlap if the same location is written by multiple processors without intervening synchronization, which is probably a data race. All of our applications are free of data-races.

The cost of creating a diff is substantial (approximately three fourths of the cost of an RPC in our system), and will probably grow relative to processor speed as memory latency falls further behind processor speed. Systems such as Midway [1] avoid the page-length copies and comparison by using a modified compiler to annotate all shared writes with code that tracks accesses by using software dirty bits [17]. When the diff needs to be created, the software dirty bits are used to determine exactly which words have been modified. While the copy and comparison are avoided, the software dirty bit approach requires language support and adds overhead to every shared write. Recent work [3] shows that well-implemented diffing mechanism can outperform software dirty bits in object-based systems, but the tradeoff is less clear for page-based systems.

2.3.4 Tradeoffs

The most obvious advantage of the multiple-writer protocol is that it allows concurrent modifications of the same page without network communication. However, write-sharing is less common than other forms of sharing, so this aspect of the multiple-writer protocol's performance is likely to be insignificant for many applications.

A less obvious advantage of MWP is that the decision to modify a page that is present in read-only state is purely a local decision. Any page that is readable locally may be written with undertaking any arbitration with other processors. Single-writer protocols, by contrast, require *ownership* of the page to be gained before a page can be modified. If ownership is not gained with the read fault in the above example, a further network RPC must be performed in order to get it. Furthermore, once ownership is achieved, it inevitably migrates away within a short time. Therefore, unless single-writer protocols are carefully crafted, producer-consumer interactions will require two network RPCs, each of which can consist of up to three messages.

Note that this is the case for any single-writer protocol, including the sequentially consistent protocol implemented in IVY [13] and the eager release consistent protocol implemented in Munin [2].

The obvious disadvantage of multiple-writer protocols is that they must use diffs to merge concurrent updates to the same page. While the use of diffs also decreases the amount of data transmitted across the network, the number of messages sent is generally more important than the overall amount of data sent because of the high message startup cost in our environment.

Only improved CPU speed or dedicated hardware support will improve diff creation cost, but improved network interfaces and OS communications systems are likely to decrease the cost of network communication at a significantly faster pace in the near future. Furthermore, byte-copying continues to get more expensive relative to floating point operations as the pace of CPU clock rate improvement continues to outpace memory access time improvements. These trends imply that the cost of diffing, already high, is likely to rise in the future.

2.3.5 Page Location

The means of locating valid pages under the single-writer protocols needs to be explained a bit further. An access miss is serviced by sending a page request to the page's *manager*, which forwards the request to the current owner. This method requires three messages to satisfy a request in the usual case, or two when the manager is also the owner. Both IVY and Munin use a scheme based on following chains of *probable owners* until the real owner is located, collapsing the probable owner pointers as a request is forwarded.

Li and Hudac [13] showed that in the worst case, k faults of a page in an n-processor system can result in a worst case of $O(n + k \log n)$ hops. Since k is in practice much larger than n for the system we are looking at, we can neglect the first term and see that k faults can require $k \log n$ hops, for an average of log hops per fault. An algorithm based on static-ownership has better worst-case performance even for a system of only 8 nodes, and the advantage grows larger as the size of the system increases.

We confirmed that the static ownership case performs better in practice by driving a simple simulation with a trace of faults incurred by the applications discussed in the next section. Our simulation shows that in the eight processor case, our static ownership scheme requires an average of 1.83 messages per page miss, while the probable owner version required 1.86.

3 Results

3.1 Environment

Our experimental environment consists of a 16-node IBM SP-2, although all performance numbers reflect eight-processor executions. The SP-2 has a high-performance Omega switch in which each bi-directional link is capable of a sustained bandwidth of approximately forty megabytes per second. Each processor is a 66MHz RS/6000 Power2.

The applications were run on a version of CVM ported to MPI [14]. MPI does not yet allow handlers to be called asynchronously on receipt of messages, so the system polls for incoming messages when outgoing messages are sent. Unfortunately, two of the applications, TSP and QS, get poor performance using automatic polling because they each have phases where the sharing is very coarse-grained. Explicit polls were inserted into these programs.

Simple RPC's in our environment require 160 μsecs. A *one-hop* lock, the case where the lock manager is also the owner, requires two messages and 228 μsecs. *Two-hop* locks require three messages and 329 μsecs. One and two-hop page faults are defined similarly, and require 939 and 1376 μsecs. In the best case, AIX requires 128 μsecs to call user-level handlers for page faults, and `mprotect` system calls require 12 μsecs. However, virtual memory primitive costs in the current system are location-dependent, occasionally increasing these costs to a millesecond or more.

3.2 Applications

The applications used in this study include four applications from the SPLASH-2 [16] suite of shared-memory programs: Water-Nsquared (Water), Water-spatial (SPA), FMM, and LU. The other four programs were locally written: FFT, SOR, Quicksort (QS), and Traveling Salesman

Problem (TSP). Table 1 summarizes the inputs and characteristics for the applications. Six of the applications use barriers, five use locks, and two use condition variables. “Data” shows that the data segments used by the applications vary in size from 349 kilobytes, for SPA, up to nearly eight megabytes for SOR. “NS Runs” gives the average time between synchronizations for the single processor case. The single processor case is used so that the numbers to not reflect a rate statistic affected by the underlying protocol.

	Input Set	Sync Type	Data (kbytes)	NS Runs (msecs)
FFT	64x64x16	b	3146	143.6
FMM	2048	l, b, c	929	2.1
LU	512x512	b	2139	10.1
QS	256k	l, c	4196	27.7
SOR	2048x2048	b	8056	75.3
SPA	512 mols.	l, b	349	460.1
TSP	19 cities	l	1604	554.0
Water	512 mols	l, b	351	122.9

Table 1. Application Characteristics

3.3 Performance

Figure 1 shows speedup for the eight applications under each of the three protocols. Four of the applications, QS, SOR, TSP, and Water, get at least a speedup of five for MW-LRC. The other applications perform less well, with FMM actually slowing down for all three protocols.

Figure 2 shows the percentage of running time spent on read faults, write faults, locks, pause flags, and barriers. For MW-LRC, the “read” category corresponds to diff requests and there is no “write” category. SW-LRC performs better than MW-LRC for FFT and LU, nearly as well in four others, and significantly worse for QS and SPA. SW-SC performs well relative to the other protocols only for the three most coarse-grained applications. Overall, MW-LRC performs 9% better than SW-LRC, and 34% better than SW-SC.

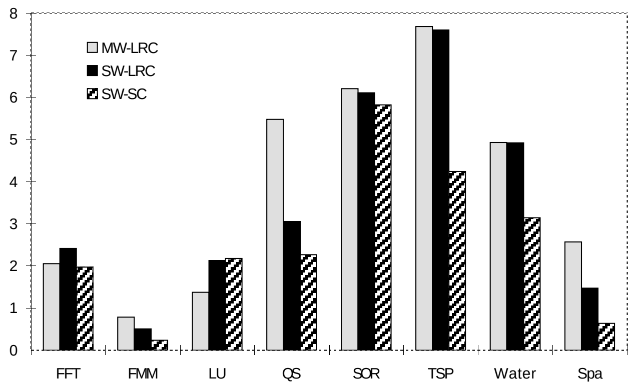

Figure 1. 8-Proc Speedup

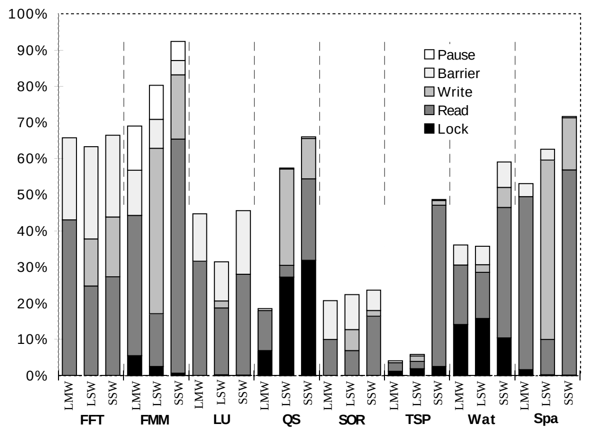

Figure 2. Latency Contribution

A common expectation is that relaxed-consistency-model DSMs are tightly limited by synchronization, while batching and pipelining of data movement largely limits the effects of data movement. Over all applications and protocols, however, 28% of running time was spent handling access faults, and only 15% of the time was spent waiting on synchronization. Applications spent 22% of their time handling access faults under MW-LRC, 28% under SW-LRC, and 41% under SW-SC. Furthermore, two of the applications with the highest non-sync runs, FFT and SPA, perform poorly, suggesting that run time heuristics which predict and initiate data movement before requests could be very useful. As the applications average a processor efficiency of 41% and total recorded time in CVM is 48%, a further 11% of running time was presumably lost to operating system interactions while application code executed, such as TLB misses and context switches. No paging occurred during any of the runs.

Figure 3 shows message totals for the protocols. MW-LRC has lower message requirements because of the absence of write faults. In all but two cases, however, the difference is not major. The two exceptions are FMM and SPA, two applications with fine-grained sharing. While direct evidence of sharing granularity is difficult to obtain without detailed simulation, FMM has more than double the fault rate of any other application, and the two applications create smaller diffs than all applications other except TSP. TSP performs well despite fine-grained sharing because it synchronizes infrequently (see Table 1).

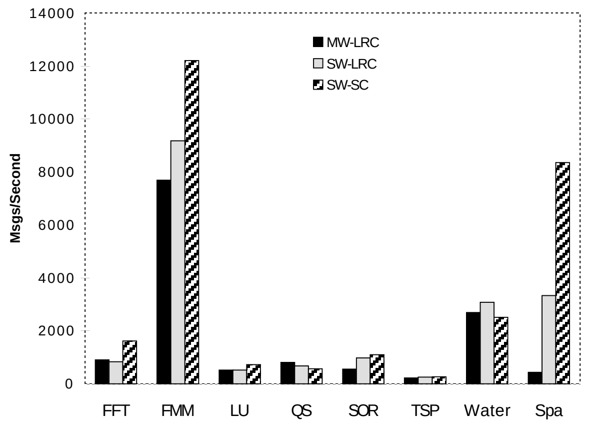

Figure 3. Messages Per Second

3.3.1 Sharing

We used our run-time system to generate traces showing all page protection changes during executions. These traces, timestamped by the globally synchronous clock on the switch of the SP-2 [15], drive a post-mortem analyzer that tracks how long individual pages are shared in various modes. Table 2 shows the results.

Several items are of interest. First, two applications, SPA and QS, have substantial write sharing under MW-LRC. Pages are write-shared an average of 6% of the time in QS, and 61% of the time in SPA. While the other applications spend 12% of their running time on write faults under SW-LRC, these two applications spend 39%. Unsurprisingly, MW-LRC performs markedly better than the single-writer protocols for these two applications, even though it performs an average of 3% *worse* than SW-LRC for the other six.

An average of 68% of all write faults under SW-LRC are to pages that are already valid, and hence would not require any network communication under a multiple-writer protocol. Overall, these *promotions* account for 40% of the total message count and 10% of running time.

Several other numbers are worth explaining. Under MW-LRC, thirty-two percent of the pages in QS are not valid anywhere in the system. Pages may become completely invalid under multiple-writer protocols when concurrent writers exchange invalidations at synchronization points. The pages are not re-validated in this specific implementation of QuickSort because the sorted array is never read.

One of the most interesting trends in the table is that not only is write-sharing converted to read-write sharing under SW-LRC, but write-only sharing is converted to read-write sharing as well. The latter situation occurs because read faults are more likely to be serviced by an up-to-date copy of the page under SW-LRC than under MW-LRC. Hence, a subsequent synchronization between the read faulter and other processors is less likely to invalidate the page.

		w-w	w-o	r-w	r-r	r-o	inv
FFT	MW	0.7	59.7	7.0	32.4	0.1	0.3
	SW	0.0	47.9	23.0	29.1	0.0	0.0
FMM	MW	0.0	0.0	0.0	0.0	0.0	0.0
	SW	0.0	84.7	7.2	7.7	0.3	0.0
LU	MW	0.1	63.7	6.0	30.0	0.2	0.0
	SW	0.0	34.4	36.0	29.6	0.1	0.0
QS	MW	5.9	77.2	9.9	5.4	1.6	32.5
	SW	0.5	76.1	18.3	3.7	1.4	0.0
SOR	MW	0.0	92.2	0.5	0.6	6.7	0.0
	SW	0.0	69.2	7.6	0.6	22.5	0.0
SPA	MW	61.0	9.4	11.9	16.7	1.0	1.1
	SW	0.0	10.1	78.3	11.6	0.0	0.0
TSP	MW	0.4	20.0	28.9	50.6	0.1	0.0
	SW	0.0	7.4	28.6	63.9	0.1	0.0
Wat	MW	0.1	11.8	5.9	82.0	0.2	0.0
	SW	0.0	19.1	8.1	72.7	0.0	0.0

Table 2. Sharing Statistics for 8-Processor LRC Runs: r-read, w-write, o-only

3.3.2 Diff Costs

The use of diffs adds four types of overhead: creation, application, handling, and garbage collection. The average diff creation cost during our tests was 125μseconds. Diff creation consumed between 0.2% and 2.2% of total run time for our applications. Diff application and handling both consume only a small fraction of this time. As discussed in Section 3.3.3, garbage collection usually consumes between 1% and 4%, although the actual number is highly dependent on application behavior and system parameters.

3.3.3 Space Overhead

	Comm Buffer	Twin Space	Diff Space	Total Ohead
FFT	100	390	680	1082
FMM	100	591	564	1168
LU	100	485	565	1063
QS	100	287	514	814
SOR	100	1008	99	1120
SPA	100	126	45	184
TSP	100	59	799	870
Water	100	99	227	339

Table 3. Memory Overhead for 8-Processor Runs Under MW-LRC (kbytes)

Table 3 shows the memory overhead cost of communication buffers, twins, and diffs for each application under MW-LRC. MW-LRC's space overhead for twins varies between 7% and 28% of the total amount of application data, between 1% and 227% for diffs, and 15% and 273% for all overhead combined. By comparison, the average space overhead for the other two protocols is 10%.

Note that diff space requirements can be arbitrarily reduced by garbage collecting, at the cost of increased CPU overhead.

Garbage collection is initiated whenever any process notices that diff or write notice buffers are becoming exhausted. The *initiating* processor adds a garbage collection request to its next barrier arrival message, and the master re-distributes this request to all processors with the barrier release.

The request requires each of the processors to re-validate every page that had at one point been valid on that processor, and then to inform the barrier master. The barrier master waits for validation acknowledgments, and then distributes *collection-release* messages. Upon receipt of a collection-release message, each processor releases all resources used to hold diffs, write notices, or twins.

This mechanism validates more pages than strictly necessary, but spatial locality ensures that most of the re-validated pages will be accessed again. Despite the two extra rounds of communication required to validate the pages, garbage collection never reduced performance by more than 4%, and usually less than 1%.

Neither of the other protocols use either diffs or twins, but SW-LRC does use write notices. However, the notices use a trivial amount of space (one word per notice) and can be garbage collected at each barrier without global coordination.

4 Conclusions

The primary contribution of this paper is a better understanding of the tradeoffs involved in allowing concurrent writers to the same page in DSM systems. We have implemented and compared the performance of three DSM protocols in the context of the CVM distributed shared memory system. The protocols are MW-LRC, a multiple-writer LRC protocol, SW-LRC, a single-writer LRC protocol, and SW-SC, a single-writer SC protocol. Overall, the multiple-writer version of LRC performed 9% better than the single-writer variant and 34% better than the sequentially consistent protocol. Stated another way, the performance impact of the choice in consistency models is approximately three times greater than the choice of whether to allow concurrent writers.

The primary performance difference between the two LRC protocols is in their handling of write sharing. Contrary to our expectations, two of our eight applications exhibited significant write-sharing. Pages were write-shared 6% of the time in QS, and 61% of the time in SPA. Write sharing in the other applications was at least an order of magnitude less. While MW-LRC performed an average of 3% worse than SW-LRC for the other six applications, it performed an average of 43% *better* for these two applications.

Set against this, MW-LRC required an average of 72% memory overhead, compared to 10% overhead for the single-writer protocols. Two thirds of this extra overhead is used for diff storage. Diff storage requirements can be greatly reduced by garbage collecting, but only at the cost of increased CPU overhead.

The primary effect of MW-LRC's appetite for memory in future systems may be in cache and TLB pollution. Such effects are becoming more important as memory hierarchies deepen. Additionally, the diffing and twinning mechanisms needed by multiple-writer protocols make heavy demands on the memory system because of the large block comparisons and copies. Fortunately, these are exactly the types of memory accesses that non-blocking caches are designed to address.

We also found that applications spent much more time waiting on data than on synchronization, suggesting that run-time mechanisms that automatically prefetch data could be of significant benefit.

Our final contribution is the design and evaluation of SW-LRC. SW-LRC achieves performance comparable to MW-LRC in most cases, but has less space overhead and is less complex. The down side is that SW-LRC's performance is more sensitive to write-sharing and has higher bandwidth requirements. Nonetheless, we feel that the simplicity and space advantages make SW-LRC a natural choice for the current generation of DSM systems that is even now making its way into the marketplace.

References

[1] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the '93 CompCon Conference*, pages 528–537, February 1993.

[2] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[3] Alan Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. DRAFT: submitted for publication, August 1995.

[4] S.J. Eggers and R.H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International*

Symposium on Computer Architecture, pages 373–383, May 1988.

[5] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[6] 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.

[7] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency*. PhD thesis, Rice University, 1994.

[8] 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.

[9] P. Keleher, S. Dwarkadas, A. 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.

[10] Pete Keleher. The Coherent Virtual Machine. Technical Report Maryland TR93-215, Department of Computer Science, University of Maryland, September 1995.

[11] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29(2):126–141, September 1995.

[12] Povl T. Koch, Robert J. Fowler, and Eric Jul. Message-driven relaxed consistency in a software distributed shared memory. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, pages 75–85, Monterey, CA, November 1994. USENIX Assoc.

[13] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface*, 1994.

[15] C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, M. Tsao, and P. R. Varker. Architecture and implementation of vulcan. In *Proceedings of the 8th International parallel Processing Symposium*, pages 268–274, April 1994.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[17] Mathew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software write detection for distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.