Symmetry and Performance in Consistency Protocols

Peter J. Keleher

Computer Science Department
University of Maryland
College Park, MD 20742

keleher@cs.umd.edu

ABSTRACT

A consistency protocol can be termed *symmetric* if all processors are treated identically when they access common resources. By contrast, asymmetric protocols usually assign a home or manager to each resource. Use of the resource by the home incurs less overhead than use by other processors. The key to good performance in such systems is to ensure that the asymmetry of the underlying protocol is skewed in the same way as that of the application.

This paper presents a comparative evaluation of a symmetric and an asymmetric DSM protocol. We pay particular attention to those performance differences caused by symmetric and asymmetric features of the protocols.

We then present the design and evaluation of an improved asymmetric writer protocol that dynamically migrates ownership according to access patterns. We show that the new protocol outperforms and is more stable than the non-migrating asymmetric protocol, and has much less memory overhead than the symmetric protocol.

1. INTRODUCTION

Distributed shared memory (DSM) systems are software systems that support the abstraction of shared memory across networks of workstations. Most such systems are page-based, using virtual memory primitives to trap accesses to shared memory (*write trapping*). For such systems, lazy release consistent (LRC) protocols are generally the fastest. The generic LRC protocols [6] differ from previous protocols in a number of ways, but perhaps most notably in that they are *symmetric*. While pages are each assigned a nominal home, home nodes are used only to satisfy cold misses. Subsequent misses are satisfied by the "last" processors to write the page. This symmetry has a number of advantages, including less "hot spot" contention and less reliance on the original data distribution.

However, recent work [10] has shown that asymmetric approaches can result in significant savings of runtime overhead when application-specific sharing behavior is available. "Home-based" LRC protocols differ from symmetric LRC protocols in that each page has a statically assigned home. Protocol performance is asymmetric with respect to each page: the protocol's performance prefers shared page modifications made by pages' owners over modifications made by other processors.

Another way of stating this is that if the protocol's asymmetry matches the application's asymmetry, home-based protocols can execute with very little overhead. Such a correspondence allows home-based protocols to dispense with consistency actions for a large chunk of shared memory modifications. The disadvantage of this approach is that the protocol's asymmetry becomes a weakness when it does not closely match that of the application, such as when the application's characteristics are difficult to discern, or change at runtime.

This paper analyzes DSM protocol performance from the perspective of this type of symmetry. We argue that many of the performance differences between protocols can be traced directly to the degree of symmetry in the protocol design. Symmetric protocols deal with dynamic conditions better, while asymmetric protocols can offer the best performance for certain types of application behavior.

We cast our discussion in terms of a comparison between symmetric LRC protocols and home-based LRC protocols. Both have advantages, and the case can be made that both are most appropriate for different types of applications. As developers might be unwilling or unable to distinguish between the two situations themselves, we present a new adaptive protocol that attempts to strike a balance between the two.

Home-based LRC protocols [10] differ from "homeless" LRC protocols in that all shared modifications are flushed to the page's home at the end of the current synchronization interval. These protocols benefit from the *home effect*, the ability to dispense with write trapping for modifications made by the home node of a given page. This has the obvious advantage of reducing communication requirements when homes are chosen well. A less obvious advantage is that the reduction in traffic allows the protocol to be less lazy. The protocol can require pages to be made consistent at the end of every synchronization interval because this often does not require any action to be taken.

By contrast, symmetric protocols must explicitly trap all writes because processes do not generally have complete information about page replication. The result is that making a page "consistent" usually implies network communication. This overhead means that symmetric protocols can only obtain reasonable performance by delaying consistency actions as long as possible. Hence, symmetric protocols *require* delays in order to amortize their expensive write trapping.

While symmetric LRC protocols are usually able to delay consistency actions enough to perform on a par with the best asymmetric protocols, the down side is that any delay requires state to be maintained. In the case of LRC, delay in reaching consistency results in a large amount of memory overhead, and the need to periodically garbage-collect consistency information. This state

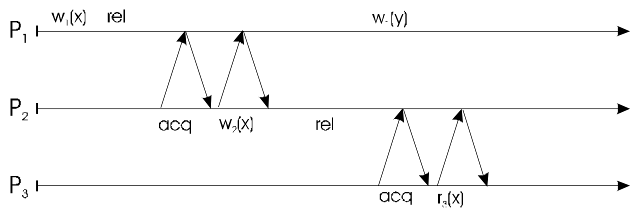

Figure 1: Diff Exchanges

maintenance results in unnecessary overhead. Asymmetric protocols are clearly preferable when the protocol's characteristics closely match those of the application.

Nonetheless, we claim that even asymmetric protocols must by dynamic if good and stable performance is to be achieved. Such protocols not only mask poor initial page placement, but also allow adaptation to changing application behavior. This adaptability could potentially be achieved by toggling between distinct protocols on a page by page basis [1], but we argue for a more integrated approach.

Note that dynamic application behavior can occur on many different time scales. A gross change may occur across iterations as molecules or point masses move and interact with different neighbors. Change can also occur within iterations when different phases of individual iterations have distinct sharing patterns.

The rest of the paper is as follows. Section 2 describes *lmw*, *home*, and *adapt,* our representatives of symmetric, static asymmetric, and dynamic asymmetric protocols. Section 3 describes the performance of these protocols and shows how their differences arise necessarily from their degrees of symmetry. Finally, Section 5 summarizes our results and relates them to the question of consistency maintenance in general.

2. PROTOCOL DESCRIPTION

This section describes *lmw*, our canonical homeless protocol, *home*, our implementation of the home-based algorithm, and *adapt*, our migratory asymmetric protocol. All protocols were implemented inside the same software DSM. Space constraints and the requirements of anonymity preclude more detailed discussion here.

2.1 *lmw* – canonical multi-writer LRC

This section briefly describes the base lazy release consistency (LRC) memory model and *lmw*, our protocol implementation.

Lmw is based on the LRC protocol described in Keleher [6]. Lmw is an all-software, page-based protocol. The central idea is that dissemination of updates to shared pages is delayed as much as synchronization allows.

Modifications to shared pages are tracked by forcing exceptions to occur at the first write attempt. The exception handler creates a copy of the page, or *twin*, before changing the protection to allow subsequent writes to proceed at full speed. The changes made to each page are described in run-length encodings called *diffs* at the end of each synchronization *interval*. Diffs are created through word-by-word comparisons of the current contents of pages with "twins" that were made prior to the initial writes. Each interval is summarized by an interval structure that includes the local timestamp, and *write notices* that enumerate the pages modified during the interval.

Lock grants and barrier releases include interval structures summarizing all intervals seen at the granter, but not at the requester. Incoming intervals are *applied* by invalidating each page named by a write notice in the interval.

Subsequent accesses to an invalidated page cause a page fault. Faults are handled by requesting all of the page's diffs from remote processors, and applying them locally in causal order. In the worst case, page faults can cause *O(n)* communication because each diff has to be collected at the processor that created it. However, the usual case is that one diff dominates (through causality) the others. This implies that all of the other diffs are present at the processor that created the dominating diff, and can be retrieved with a single remote request.

Lmw has two important advantages over prior protocols. Initial page faults are handled by retrieving a copy of the page from the page's manager. However, all subsequent faults are handled applying diffs retrieved from the processors that modified the page. This means that all data traffic for a particular page is between processors that access the page, rather than with the pages's manager (which may or may not be one of the communicating processors).

Second, any processor can modify any locally valid page without prior negotiation with the current owner. The protocol therefore necessarily allows multiple concurrent writers. Allowing multiple concurrent writers is crucial in mitigating the performance effects of false sharing.

2.2 *home*

The home protocol implemented in this work is based on the software-only protocol in Zhou [10]. The home-based multi-writer protocol statically assigns a *home* to each page. A page's home node can modify the page without creating diffs. Other processes that modify the page use the diffing mechanism described in Section 2.1 to update the home process at the end of the current interval.

The home-based protocol has two potential advantages: the "home" effect, and the short lifetimes of many data structures. "Home-less" protocols summarize all modifications to shared state as diffs. Aside from the expense of diff creation and application, diffs can cause homeless protocols to have voracious appetites for memory [5]. The "home" effect refers to the fact that home-based protocols allow the owner of a page (i.e. the home) to dispense with creating diffs describing its own modifications. Diffs are created only to describe modifications made to a page by nodes other than the page's home. Modifications made by the home are merely noted locally. No network communication is required.

The second advantage is that home-based diffs have short lifetimes. The ugliest aspect of homeless protocols is that the data structures used to describe shared modifications can not be discarded until they have been explicitly garbage-collected. For example, consider Figure 1. The example shows processes P_1 through P_3 accessing migratory data x. First, P_1 modifies x and releases a synchronization variable. P_2 then acquires the synchro-

App	Sync	Kbytes / Sec	Sync / Sec	Misses / Sec	Diffs / Miss
barnes	Bar	2223	10.1	602.7	4.8
fft	Bar	3203	19.7	791.3	1.1
nsq	bar, lock	1592	114.4	594.4	2.3
qs	lock	1874	451.0	324.3	4.3
ray	lock	233	187.4	130.7	4.5
sor	bar	713	79.9	176.8	1.0
spatial	bar, lock	1787	22.8	2680.5	1.0
tsp	lock	187	25.0	198.7	3.6
water	bar, lock	1721	1950.5	506.5	2.4

Table 1: Application Characteristics (8-proc, lmw)

nization variable, which causes any local copy of the page containing *x* to be invalidated, and then accesses *x*. Touching *x* causes an access miss that is satisfied by requesting the diff of $w_1(x)$ from P_1. However, the diff can not be discarded by P_1 even after it has been supplied to P_2, because P_1 can not know if or when some other process (P_3, for example) might subsequently request the diff as well. More generally, the diff can not be discarded until the system can guarantee that it will not be requested by any other process. The situation is complicated even more by the fact that if and when P_3 requests $diff_1$, the diff may be requested from P_2 rather than P_1. For performance reasons, then, P_2 can not discard the diff either. The result is that no diff, nor any of the write notices that name diffs, can be discarded until they are explicitly garbage-collected.

By contrast, diffs have very short lifetimes under home-based protocols. Diffs are created at the end of intervals, flushed to the home nodes, and immediately discarded. This is correct under home-based protocols because *all access misses are serviced via complete page copies*, rather than by applying diffs to pre-existing page replicas. Consider Figure 1 again. Assuming that the manager of the affected page is P_3, both P_1 and P_2 will create and send diffs to P_3 prior to their releases. The advantage is that both diffs can be immediately discarded. The main disadvantage is that more messages are sent. The situation can be even worse. Consider the case where a fourth process, P_4, is the home node for the page. In this case, both P_1 and P_2 will send diffs to P_4. Both P_2 and P_3 will request copies of the page from P_4, a node that isn't involved in the communication.

Nonetheless, the results from the original home-based paper showed home-based protocols to have a significant scalability advantage over the generic homeless protocol implementations. One of the purposes of this paper is to extend these results by comparing the home-based protocols against a more realistic homeless protocol, i.e. one with the optimizations described below. We show that these seeming minor optimizations can have a large effect on the relative characteristics of homeless and home-based protocols.

2.3 *adapt*

The adapt protocol differs from home primarily in that the home of a node is dynamic, and can migrate in response to either a read or a write to shared data. The decision of when to migrate ownership is under the control of a simple heuristic. The protocol is equivalent to a single-writer LRC protocol [5] if ownership always migrates on write requests by non-home nodes. It is equivalent to home if ownership never migrates.

Adapt's heuristic activates the diffing mechanism rather than requesting ownership if the use_diff flag is set when a local write fault occurs. The use_diff flag is set on the home node when receiving an ownership request for a page that is currently writable, such that the writing began since the last synchronization. The intuition here is that the old owner is potentially relinquishing ownership before finishing its modifications. Since the new owner is also modifying the page, a potential page thrashing (or ping-ponging) situation is set up unless we constrain the old owner from attempting to regain ownership before the end of the current interval. Since our programming model prohibits data races, this situation can only come about because of false sharing.

The use_diff flag is also used when a process servicing a read request has also serviced a write request on the same page since the last synchronization. In this instance, a flag is returned to the reader, which sets its use_diff flag. The intuition here is that a subsequent write fault by the reader would indicate false sharing with respect to the earlier writer.

Finally, ownership is preemptively migrated to a process that makes a read request if the requester has written the page in the past, and the current owner has not written the page during the current interval. This mechanism is used to handle migratory data.

3. EXPERIMENTS

All of our experiments were run on a sixteen-processor IBM SP-2. Unless otherwise specified, however, our default system size is eight processors. Each node is a 66.7 MHz POWER2 processor. The processors are connected by a 40 MByte/sec switch. The operating system is AIX 4.1.4.

Communication is accomplished using UDP/IP over the switch. Lock acquires are implemented by sending a request message to the lock manager, which then forwards the request to the last requester. This takes only two messages if the manager is also the last owner of the lock. Two-hop lock acquires take 779 µsecs, while three-hop lock acquires take 1185 µsecs. Simple page faults across the network require 1576 µsecs. Page fault times are highly dependent on the cost of mprotect calls, 15 µsecs, and the cost of handling signals at the user level, 120 µsecs. Minimal 8-processor barriers cost 1176 µsecs.

3.1 Application suite

Table 1 shows the nine applications in our application suite. *Barnes*, *fft*, *nsq*, *ray*, *spatial*, and *water* are all from the SPLASH2 [9] suite. *Water* is a largely unmodified form of water-nsquared, whereas *nsq* and *spatial* have both been modified by researchers at Princeton in order to reduce synchronization requirements. Barnes has been modified in order to eliminate implicit synchronization through shared memory (which does not work on an LRC system). Sor is a simple jacobi stencil. Qs implements a quicksort with a centralized task queue, and tsp is the traveling salesman problem, also implemented with a centralized task queue. All but tsp and qs are iterative scientific applications

App		Speedup			Messages			KBytes		
		4	8	16	4	8	16	4	8	16
barnes	l	3.0	4.0	3.6	4041	16137	59807	6188	15414	36705
	h	3.2	4.9	5.3	1760	4111	9203	7709	16835	34800
	a	3.3	5.1	6.6	2160	4556	9646	9141	18231	35713
fft	l	1.7	2.9	4.6	6343	7535	11674	25315	26453	32355
	h	1.3	2.6	5.0	6018	5911	9012	60440	28251	87674
	a	2.1	3.5	5.9	6297	6139	8782	26519	20953	43398
nsq	l	3.5	6.2	8.9	6484	12278	23974	12653	27669	64780
	h	3.5	6.1	8.6	6667	15766	24798	21578	43215	82077
	a	3.5	6.1	8.7	7679	13675	34952	22777	42761	86595
qs	l	2.9	4.2	4.1	3156	5202	8386	8932	15008	18878
	h	2.5	3.2	2.8	5086	7635	8759	12506	20685	23238
	a	2.9	4.3	3.9	5274	6515	12319	12710	10907	22542
ray	l	3.8	7.0	12.3	2202	4635	7866	7391	13333	22872
	h	3.1	5.2	7.9	3393	5891	9479	8927	15358	27077
	a	3.2	5.3	8.0	2666	5285	8928	8269	15367	24794
sor	l	3.9	7.2	12.3	453	1057	2265	975	2288	4955
	h	3.9	7.3	12.1	452	1057	2265	1127	2329	9634
	a	3.9	7.3	12.1	453	1057	2265	958	2264	4974
spatial	l	2.6	4.2	4.5	28038	65437	158663	64842	151340	329144
	h	2.9	4.5	4.9	22287	59900	129136	100065	251019	526822
	a	2.7	4.5	4.9	28067	65486	132994	113256	264352	537215
tsp	l	4.0	7.6	13.3	6185	9207	12961	2981	6911	14827
	h	4.0	7.6	12.2	6182	10121	12618	16145	27156	39327
	a	4.0	7.4	12.7	8407	9452	15154	22932	24986	39652
water	l	2.9	4.5	5.6	11853	23003	44687	7389	16124	38071
	h	2.5	3.6	4.0	16156	32306	62342	12355	25028	51886
	a	2.8	4.4	5.4	12545	23758	45815	11816	22433	45285

Table 2: Scalability

typical of much high performance computing work. Qs and tsp are non-iterative and have much lower locality.

The last four columns of Table 1 show communication and synchronization behavior for eight-processor runs using lmw. All of our applications use either global barriers, mutually exclusive locks, or both. The third column shows the rate of communication in Kbytes/sec. The fourth and fifth columns show the number of synchronizations (of both types) and remote page faults per second. The former gives an idea of the synchronization granularity, while the latter gives an idea of how tightly coupled the applications are. The last column is the average number of diffs that needed to be applied to validate a page under lmw. This number does not directly tell us whether the diffs are concurrent (implying false sharing) or ordered (implying migratory data). However, migratory data will be guarded by synchronization, so we can safely assume that false sharing is implied by a large number of diffs per miss combined with a low synchronization rate. Barnes, for example, has the highest diff per miss rate combined with the lowest synchronization rate. This application will be discussed further below.

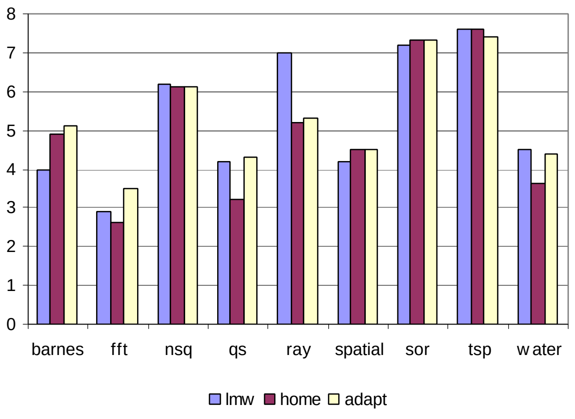

Figure 2: 8-Processor Speedup

Our applications were chosen in order to cover a spectrum of application types. The communication rates and the number of misses per second both span more than an order of magnitude, while the synchronization granularity spans more than two orders of magnitude. The number of diffs per miss varies by almost a factor of five.

The characteristics of *nsq* and *water* are especially interesting, as both have been described in published studies as minor modifications of the same program (water-nsquared from SPLASH2). Clearly “minor” modifications can have major effects on overall performance.

3.2 Base Performance Comparison

Our overall discussion of relative performance in this paper encompasses three categories: raw speedup, performance stability, and memory overhead.

Figure 2 shows the eight-processor performance of our applications for all three protocols. Note that the differences tend to be larger for the more demanding applications, but these differences

App		Seconds	Msgs	Diffs	Intervals	KBytes
barnes	l	2.83	16053	3250	41	15407
	e	2.90	16028	3280	65	15423
fft	l	1.68	8123	7296	120	29692
	e	2.21	8183	15488	120	29822
nsq	l	5.74	12305	5723	120	27674
	e	5.82	12320	6685	874	27875
qs	l	0.95	19673	12700	5065	54270
	e	1.12	20976	25580	5831	96603
ray	l	6.70	1502	790	271	682
	e	6.72	1492	857	298	724
sor	l	0.58	1057	518	296	2288
	e	6.21	1057	75702	296	3317
spatial	l	1.48	33277	4836	265	21766
	e	1.62	33332	9084	317	24500
tsp	l	7.25	9213	3812	636	6915
	e	7.26	9229	3874	634	6928
water	l	3.89	22993	2669	1856	16123
	e	4.17	23015	7807	10785	18411

Table 3: Effect of Lazy Diffing on lmw

tend to be less visible because of the chart's scale. Table 2 shows speedups and overall communication requirements for four, eight and sixteen processor configurations. Table 6 shows the number of diffs created, the number of remote page faults, and the total amount of memory overhead (as a percentage of the application shared segment sizes).

The factors leading to the differences in speedup will be discussed in the following subsections. However, the bottom line is that adapt averages about the same speedup as lmw (within ~1%), and from 10% to 11% higher than home in all three configurations. Table 2 shows that these margins remain relatively stable in configurations from four to sixteen processors. The relative performance of lmw and home differs from that presented in Zhou [10]. We discuss the roots of this difference in the next subsection.

3.3 Lazy diff and interval creation for *lmw*

The above results for lmw reflect several optimizations over the canonical LRC algorithm described in the literature [6]. Chief among these optimizations is *lazy diff creation.* Lazy diff creation refers to creating diffs only on request. The result is that some diffs are never created, and multiple diffs can often be consolidated into one. The most direct performance impact of this reduction is that less time is spent creating twins and diffs. The costs of both of these actions scale with the size of virtual memory pages and the number of intervals during which data is modified, *not* on the amount of data modified. Creating two diffs that describe modifications to different portions of the same page is therefore very nearly twice as costly as creating one. Diff application, on the other hand, is dependent only on the number of distinct words modified. In general, diff application is much cheaper than diff creation. Note that TreadMarks used lazy diff creation, but the exact mechanism has not been described or measured in any detail.

Lazy diff creation has indirect benefits in reduced pressure on memory (fewer twins and diffs), less bookkeeping, and less frequent garbage collection.

Table 3 shows application performance both with ("l") and without ("e") lazy diffing. The table shows runtime in seconds, and communication and protocol information for each of the applications. Overall, lazy diffing reduces the number of diffs by an average of 73%, and communication bandwidth by 23%. Speedup increases by an average of 24%. Even without including sor in the average, speedup still improves by more than 12%. Clearly such a large and widely applicable change could affect scalability as well.

The performance impact of lazy diff creation is most pronounced in sor. Since this application is essentially a simple Jacobi stencil, data on the interior of the data assigned to each process is never requested by other processes. Interior data is effectively treated as private data, and any diff creations on these pages are pure overhead. The lazy diffing approach never creates diffs for these pages because no requests for them are ever received. While a twin is made for each page the first time it is modified, the cost of these twin creations is effectively amortized across the entire application execution. The dormant twins consume virtual memory space, but are either ignored or paged asynchronously to disk by the operating system because they are not accessed after creation. By contrast, the eager diffing approach requires diffs to be made for each page during each iteration.

3.4 Optimizations for *home*

The home protocol discussed in this paper differs from the canonical implementation in at least two ways. The first is a mechanism similar to the lazy diffing discussed above. This mechanism eliminates local page faults for pages modified by their owners. Such faults do not result in network communication, but do require the OS to deliver a signal to the application. sor sped up by more than a factor of three with this optimization.

Our second modification to home addresses the most obvious drawback of statically assigned homes: the initial assignment must be done well. The original formulation of home-based protocols addressed the problem of assignments by requiring user annotations on each section of data, and observed that making such assignments is easy for the majority of cases. This is especially true for scientific applications, which tend to distribute computation among processes as large array slices. Even when this is the case, however, such annotations are an additional burden on the programmer.

We evaluate three automatic methods of assigning homes. The three alternatives are round-robin assignment, assignment of contiguous blocks of pages, and a first-touch runtime method that performs a single reassignment based on early access behavior. The round-robin scheme distributes load equally over all processes, but will only rarely match pages with the processes that modify them. The "block" scheme is similar, but has a better chance of making good assignments because the responsibility for modifying data is often distributed in a chunk-wise fashion by

App		Seconds	Msgs	Diffs	Misses	Kbytes
barnes	b	4.67	4003	2883	3652	16715
	f	4.95	4115	2678	3793	16781
	r	4.67	4003	2883	3652	16716
fft	b	17.72	7704	12677	6512	77524
	f	8.09	7455	112	7280	29806
	r	13.61	8183	13552	6384	80566
nsq	b	17.24	12150	4817	8983	40333
	f	17.13	12103	3821	9103	38784
	r	17.96	15780	6077	9047	43347
sor	b	0.96	1056	276	314	2612
	f	0.95	1057	0	518	2328
	r	4.33	3074	14559	463	61122
spatial	b	13.60	30873	6503	29278	121336
	f	13.14	33446	216	32588	132741
	r	13.52	30334	7812	28571	118885
water	b	11.44	30896	8531	4270	23392
	f	11.21	30938	8482	4318	23416
	r	11.73	32310	10170	4160	25035

Table 4: Impact of Home Assignment Scheme

scientific codes. The first-touch scheme migrates any pages that have not been written by their initial owner, but have been written by at least one other process. The migration decisions are distributed on release messages for the sixth barrier.

Table 4 shows statistics for the three schemes on the six applications that have enough barriers for the migratory scheme to have an effect. The block, first-touch, and round-robin protocols are identified by *b*, *f*,and *r* respectively.

Five of the six applications perform at least as well with the first-touch scheme as either of the others. However, the improvement is minor in most cases. With nsq, for example, the migratory scheme reduces bandwidth and the diff creations, but also slightly increases the number of remote faults. Overall, first-touch is only one percent better than the chunk scheme for nsq, and five percent better than the round-robin scheme.

Most of these applications are relatively insensitive to the choice of home because a high proportion of the pages are write shared (over the course of an entire iteration) by many processes. The exceptions are sor and fft. For sor, little runtime overhead is incurred if the interior pages are assigned to the process that modifies them. However, an enormous number of diffs are created if the pages are assigned to another process. While this sort of mistreatment is unlikely to occur on programs as heavily scrutinized as the applications in this paper, it could well become more common as multiprocessor systems become more common and more complex multiprocessor applications are written.

FFT'S butterfly pattern of data sharing makes simple assignments difficult. However, the migratory scheme is able to eliminate more than 61% of the bandwidth requirements and more than 91% of the diff creations.

3.5 Dynamic sharing patterns

We evaluate the success of adapt according to three criteria: performance, stability, and memory overhead. Section 3.2 showed that adapt performs as well as lmw, and significantly better than home.

Table 5 shows protocol performance for dynamic versions of two of our applications. We created modified versions of fft and sor to test protocol response to dynamic sharing patterns. After computing several iterations normally, each process assumes the responsibilities of the process with the next highest process id. We started timing *after* the shift took place in order to capture steady-state performance after a shift in access patterns.

Relative to the other two protocols, adapt improves performance by approximately 30%, primarily because of the lack of diff creations. Home actually performs relatively well even though it creates a huge number of diffs. After the shift, most pages are written by processes other than the home, so the protocol essentially turns into an update protocol. This has the advantage of sending data before it is needed. More importantly, however, home is able to aggregate updates of nearly ten pages into each diff message sent. Pure invalidate protocols, on the other hand, need to fault over modifications one page at a time.

For sor, the comparison is more clear. The majority of modified pages are read only by the process doing the writing. Hence, incorrect home assignments cause an enormous number of diffs to be created, and a similar number of update messages to be sent. However, most are never used, and performance slows by an order of magnitude.

Note that the large number of diffs create by home imply an equal number of twins. Hence, the memory overhead of home in this case, while temporary, is quite large.

Finally, Table 6 shows that the memory overhead of adapt not only tracks that of home, but is actually lower for all of our applications. The reason is that adapt does a better job of ownership assignment, and therefore creates fewer diffs.

3.6 Causality Representation

Another problem of homeless multi-writer protocols like lmw is the scalability of internal data structures. TreadMarks includes complete vector timestamps in all interval notices communicated between processes. Vector timestamps allow any process to locally determine the ordering between events or intervals, if any. However, vectors are large (at least 32 or 64 bytes for a sixteen-processor system), and scale up in size linearly with respect to

App		Time	Diffs	Messages			Segvs	Remote Misses
				diff	read	write		
fft-r	l	4.29	2832	2940	728	0	4634	2832
	h	4.17	4678	445	2502	0	6619	2502
	a	3.43	15	16	2250	1330	4639	3088
sor-r	l	1.56	126	126	0	0	252	126
	h	14.84	18000	2034	63	0	18063	63
	a	1.38	0	0	126	0	252	189

Table 5: Dynamic application performance

App		Memory	Diffs	Remote
barnes	h	25%	2675	4205
	a	8%	2738	3788
	l	94%	3250	4177
fft	h	37%	36	7282
	a	0%	112	7272
	l	8%	7296	7296
nsq	h	17%	123	11465
	a	9%	6070	9022
	l	281%	5723	10330
qs	h	31%	367	3079
	a	3%	419	4827
	l	47%	2575	2163
ray	h	0%	622	3823
	a	2%	1697	3642
	l	3%	818	528
sor	h	0%	0	518
	a	0%	0	518
	l	1%	518	518
spatial	h	7%	17	64898
	a	5%	13078	57986
	l	90%	9432	64891
tsp	h	18%	350	7879
	a	6%	3447	6458
	l	62%	3814	7335
water	h	78%	55	5382
	a	12%	10170	4160
	l	304%	2639	4746

Table 6: Protocol Comparisons

the number of processes. Since each processor maintains information on all system processors, the total storage space *on each processor* for the vectors is $O(n^2)$, with potentially large constant factors. Vectors require cycles to handle, increase the amount of consistency information passed between processes, and are decidedly inelegant.

By contrast, home-based protocols serialize all updates to pages at the pages' home. Furthermore, all page faults are handled by full page requests. The result is that home-based protocols do not need vector timestamps, and hence can use more scalable data structures.

However, vector timestamps are not intrinsic to homeless LRC protocols. In fact, CVM dispenses with vector timestamps by inferring ordering information from the relative placement of interval structures in memory.

Instead, CVM exploits the fact that diffs are *learned* about in an order that agrees with the above hb1, assuming that no interval is "seen" before all intervals ordered before it have been seen. This property is actually quite easy to maintain. When process P_1 communicates with P_2, it appends all interval information that it does not know that P_2 has. P_2 simply discards any redundant information. P_2 is therefore guaranteed to learn about new intervals in an ordering that agrees with hb1. If this interval information is stored in the same order as it is seen, relative local addresses of consistency information give an ordering consistent with happens-before, and the vector timestamp is not needed. Note that the addresses of interval information form a global ordering, whereas happens-before is only a partial ordering. Therefore, the addresses do not maintain enough information to determine whether two intervals are truly ordered, or are concurrent.

	Barrier Context			Vector Timestamps		
#procs	4	8	16	4	8	16
barnes	0%	0%	0%	0%	0%	0%
fft	0%	0%	0%	0%	1%	4%
nsq	1%	4%	5%	3%	8%	14%
qs	0%	0%	2%	8%	15%	24%
ray	0%	1%	5%	3%	8%	15%
sor	0%	0%	0%	5%	15%	28%
spatial	1%	1%	2%	1%	1%	3%
tsp	0%	0%	2%	3%	5%	7%
Water	0%	2%	4%	3%	8%	17%

Table 7: Consistency Overhead

The result is that the savings in both bandwidth and per-processor storage could potentially be offset by the cost of sending information that vector timestamps would allow us to identify as redundant. This is only a problem for barrier arrivals. The consistency information appended by TreadMarks to each barrier arrival only includes intervals of the arriving processor. The barrier master can infer the causal context of each interval from the vector timestamps. Each processor under CVM, by contrast, must potentially send intervals from all processors in order to establish the intervals' context. This means that much of the consistency data sent is redundant.

We instrumented CVM in order to measure the cost of this redundant information. At the same time, we computed the cost of adding vector timestamps to all communicated intervals. Table 7 shows these costs as a percentage of the total amount of consistency-related information currently transmitted. The three columns labeled "Barrier Context" show the overhead of the extra intervals sent by CVM's scheme. The columns labeled "Vector Timestamp" show the overhead of using vector timestamps, as in TreadMarks, both for four, eight, and sixteen processes. The extra information sent by CVM never amounts to 5% of the total consistency information, while vector timestamp information can amount to 8% for four processes, 15% for eight, and 28% for sixteen. The bandwidth overhead for the TreadMarks case is already significant at sixteen processors for many of the applications, and might become prohibitive at larger system sizes. Note that the vector timestamp overheads assume two-byte values for each entry in the vector timestamp. Four-byte values are more realistic, so these percentages could be doubled.

Since all updates to a given page are serialized at the home, we track page versions with a scalar counter. The home maintains a version number for each page that it owns. Each diff application causes the version number to be incremented. This version number is returned to the diffing process on the diff reply, and included in the write notice for that page. The version number is also incremented when the home process modifies the page. Any local page whose version is less than a version named by an incoming write notice is invalidated.

4. RELATED WORK

Concurrent with this work, Cox et al. published a second comparison of home-based and homeless protocols [3]. Their conclusions were similar to ours: the differences between well-tuned protocols are minor, depending more on the relative costs of communication and memory copying than anything intrinsic to the protocols. Home-based protocols use more bandwidth and benefit more from fast communication protocols. Homeless protocols are preferable when communication bandwidth is at a premium.

Amza et al. [1] have published work on DSM protocols that adapt between different protcols. In their case, the protocols were Keleher's single-writer protocol [5] and the default multi-writer LRC protocol. Although the dimension in which their protocols differ is different than the protocols discussed in this paper, they reached similar conclusions about the value of adaptive protocols. Their study showed that simple runtime information could be used to build an adaptive protocol that performed as well as the best of the two alternatives for all applications that they examined.

5. CONCLUSIONS

Symmetric protocols treat all processors identically when they access shared resources. By contrast, asymmetric protocols commonly assign a specific manager to each resource. Use of the resource by the manager usually incurs less overhead than use by other processors.

This paper has presented quantitative and qualitative evaluations of three consistency protocols: lmw, home, and adapt. The advantages of lmw are runtime performance and stability in the face of changing or unknown access patterns. The advantages of home are good performance when resource managers (homes) are well assigned, and low memory overhead. We have argued that these characteristics are not specific to just these protocols, but to symmetric and asymmetric consistency protocols in general. Finally, we show that that asymmetric protocols can match the performance of symmetric protocols by analyzing the performance of *adapt*, an asymmetric protocol that dynamically adapts protocol asymmetry to that of the application. The performance results in Section 3 show that adapt matches lmw's runtime performance and stability, and home's low memory overhead.

We argue that the tradeoffs between symmetric and non-symmetric approaches exist in other domains as well. For instance, consider a symmetric sequentially consistent (SC) [7] protocol. SC protocols require all prior writes to be performed globally before subsequent writes can be started. With an asymmetric protocol, a page's owner has exact knowledge of the copyset, and may know that no other processor caches the page. Hence, the write's performance may require no coherence actions. With a truly symmetric protocol, by contrast, potential writers will not have exact copyset information, and therefore all writes would require coherence transactions.

Somewhat further afield, Bayou's [8] anti-entropy protocols are nominally symmetric and enforce a relaxed consistency model. The result is that large amounts of state need to be maintained in order to ensure that modifications are performed at all sites. Bayou limits this state explosion by resorting to an ownership scheme for committing updates. Similar problems exist with distributed object stores [2] and distributed simulation [4].

6. REFERENCES

[1] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Software DSM Protocols that Adapt between Single Writer and Multiple Writer," in *Proceedings of the Third High Performance Computer Architecture Conference*, February 1997.

[2] M. Castro, A. Adya, B. Liskov, and A. C. Myers, "HAC: Hybrid Adaptive Caching for Distributed Storage Systems.," in *Proceedings of the 16th Annual Symposium on Operating System Principals*, October 1997.

[3] A. L. Cox, E. d. Lara, Y. C. Hu, and W. Zwaenepoel, "A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory," in *Proceedings of the 5th High Performance Computer Architecture Conference*, January 1999.

[4] D. R. Jefferson, B. Beckman, F. Wieland, and L. Blume, "Distributed simulation and the Time Warp Operating System," *Operating Systems Review*, vol. 21, pp. 77-93, 1987.

[5] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[6] 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*, January 1994.

[7] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, pp. 690--691, September 1979.

[8] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing Update Conflicts in a Weakly Connected Replicated Storage System," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[9] 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*, June 1995.

[10] 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 2nd Symposium on Operating Systems Design and Implementation*, October, 1996.