Locality and Performance of Page- and Object-Based DSMs

Bryan Buck and Pete Keleher
University of Maryland

This paper presents simulated results comparing representatives of two approaches to software DSM: an object-based protocol and a page-based protocol. We explore the performance implications of each approach, including the object approach's advantages in bandwidth consumption and lack of false sharing.

Somewhat surprisingly, the locality and data aggregation advantages of page-based systems prove to be the dominant factors with typical operating system overheads. We show that large page sizes actually improve the performance of multi-writer protocols, primarily because validating a single object validates all other objects on the same page as well. Since our applications have significant spatial locality, these additional validations reduce the number of remote misses, without significantly increasing bandwidth requirements. For three out of the four applications we tested, our page-based protocol matched or outperformed our object-based protocol under typical operating systems costs.

We quantify this effect, and conclude with a discussion of techniques that could allow each approach to benefit from the best features of the other.

1. Introduction

One of the most contentious debates in the software distributed shared memory (DSM) community has been between the proponents of object-based systems [1-3] and page-based systems [4-6]. The former advocate using program objects as a natural consistency granularity. The advantages are clear: specifying objects by name limits the scope of the consistency action to the object's extent. This limitation reduces the amount of data that needs to be transferred, allows the data to be sent as an update at the same time synchronization is acquired, and prevents the consistency action from affecting other objects on the same page (false sharing).

Advocates of the page-based approach usually contend that the object-based synchronization model is more complicated and unintuitive. Page-based systems require only synchronization to be specified; the scope of each synchronization is the entire shared address space. By requiring synchronization accesses to specify the data object, the object model essentially requires the same amount of application information as message-passing systems, albeit in a less cumbersome form. Secondarily, the page-based approach (even with relaxed consistency models) is closer to that seen by programmers on hardware shared memory machines.

Lost in this debate is the question of performance. This is largely because conventional wisdom holds that fine-grained performance and false sharing doom page-based approaches. However, multi-writer systems [5, 7] can largely hide the effects of false sharing, and also offer additional locality and data aggregation advantages.

The locality advantages arise from the fact that validating a single object on a shared page implies that all other objects on that page are validated as well. In effect, the rest of the objects are prefetched automatically. Applications that have significant spatial locality therefore have fewer page faults, and lower consistency and communication requirements. Another way of looking at this is to say that page-based systems automatically aggregate data better than current object-based systems.

The central tradeoff to be explored, then, is the performance effects of the elimination of false sharing and the (potentially) lower communication requirements in object-based systems versus the message-aggregating and locality effects in page-based systems.

Rather than perform a single-point comparison between the different approaches, we use simulation to explore this tradeoff for a variety of simulated environments. These environments differ in the cost of operating system primitives, such as communication and page protection system calls. We use two representative protocols, one based on CRL's (C Region Library) [3] protocol, and another based on the lazy multi-writer protocol [5] used by CVM (Coherent Virtual Machine) and TreadMarks [16]. We implemented both protocols in CVM (which provides a framework for implementing software DSM protocols), and then ported them to the simulator. This latter step was relatively simple because the simulator is basically a variant of CVM that uses a thread library to context-switch between virtual processes, and the ATOM [8] binary rewriter to instrument shared reads and writes. We validated the simulator by comparing simulated results with output from actual performance times on an IBM SP-2 for our applications, and message and event counts from the CRL publication. Our application suite consists of Water, Barnes, and LU from the SPLASH2 benchmark suite [9], and SOR, a common red-black Jacobi program.

The simulator allows us to vary message costs (the importance of data aggregation goes down with message costs), page sizes (larger page sizes increase gains from spatial locality, but also increase false sharing), and the cost of page-handling operating system primitives. The latter is especially important because page-based systems rely heavily on virtual memory primitives to control and detect accesses to shared pages, while object-based systems use other techniques to detect modifications to shared data.

The focus of this paper is an investigation into how these two protocol types perform under a variety of simulated conditions. We focus primarily on communication, the dominant form of overhead on distributed computing platforms. As expected, we found that the page-based approach uses fewer messages, as our applications all have significant locality. Surprisingly, however, we also found that the object-based approach has no intrinsic bandwidth advantage. We investigate the impact of the object-based approach's advantage due to the absence of false sharing, and the page-based approach's advantages in data aggregation and in exploiting spatial locality.

The rest of the paper is as follows. Section 2 describes the simulator and the two protocols, and Section 3 describes our experiments. Section 4 summarizes our results and suggests ways in which the object-based approach can be modified to exploit spatial locality as well.

2. The Simulator

The simulator used in this study is based on the CVM (Coherent Virtual Machine) [7] software DSM. CVM is a user-level library that features a set of base classes that provide a framework for implementing specific DSM protocols. These classes include

a generic protocol class, a class that allows a protocol to hook into the virtual memory system to set page permissions and handle page faults, and efficient, reliable message-passing facilities based on UDP. New protocols are added by deriving classes from these base classes. The fact that all protocols implemented under CVM use the same underlying support for functions such as handling virtual memory and message passing allows them to be fairly compared.

The simulator is made up of a modified version of the CVM library and a set of instrumentation code that is added to an application using the ATOM [8] binary-rewriting tool. It runs on a single processor, using threads to provide multiple virtual processors. The instrumentation code maintains a processor cycle count for each virtual processor, and handles switching between threads. The cycle counts are based on the types of instructions being executed, and are meant to be typical of RISC processors in general, not to model any specific processor. We do not simulate the effects of caches, pipelining, or multiple instruction issue.

Operations such as signal handlers (used to trap write faults and to inform the system of incoming requests) and communication primitives are assigned costs, expressed in cycle times, that can be varied by parameters passed to the simulator when an application is run. A message leaving a virtual processor is tagged with an arrival time based on the cycle count of the sender and the assigned message costs, and the message passing facility in the modified CVM library delivers the message to the destination virtual processor when it reaches the message's arrival time.

The simulator also uses instrumentation to catch shared reads and writes, allowing it to simulate page faults and to record which words have been changed by which processors. This information is used to allow us to know when a processor has not yet propagated a change to another processor through the DSM protocol, even though in our simulator the "processors" are threads that actually share the same address space.

2.1 The Region-Based Protocol

The region-based protocol is based on the consistency model used by CRL [3], and it presents an identical interface to the programmer. It allows a program to create *regions* of shared memory, and provides synchronization calls that affect a single region at a time. Since the system knows which region the programmer intends to protect with a given synchronization call, it can limit the data exchanged to only the data that is necessary to make that region consistent. This data is piggy-backed onto the messages that are used to implement synchronization. If the programmer allocates data in regions at a fine enough grain, then there is no false sharing.

As in CRL, a shared region is created using the `rgn_create` call. This sets aside memory for the region on the machine issuing the call, and returns a value of type `rid_t`, which uniquely identifies the region. This identifier can be passed to the `rgn_map` call to obtain a memory address at which the region can be read or written. The identifier is valid on all CVM nodes; the memory address is not. Memory is set aside for regions allocated on other nodes the first time they are mapped locally.

A node is allowed to read or write a region only when a *read operation* or *write operation* is in progress. A read operation, during which a node may only read data from the region, is started with the call `rgn_start_read` and ended with the call `rgn_end_read`. A write operation, during which the node may read or write the region, is delimited by the calls `rgn_start_write` and `rgn_end_write`. At any given time, multiple read operations or a single write operation on each region may be in progress.

If a node has a valid copy of the region when it makes a `rgn_start_read` call, then a flag is set to indicate that an operation is in progress and the call returns immediately. If the node does not hold a copy of the region, then it must request a copy of the region from the *manager*. The manager for each region is assigned when the region is created and never changes. When the manager receives the request, it forwards it to the current *owner* (unless it is itself is the owner) and makes the requesting node the new owner. If no operation or a read operation is in progress on the owner, then the owner immediately responds with an up-to-date copy of the region. If a write operation is in progress, then the owner places the message on a queue, to be responded to when its operation has completed.

A write operation works similarly. If the node on which `rgn_start_write` was called holds the only valid copy of the region, then a flag is set and the operation is allowed to proceed immediately, without any communication having taken place. If this is not the case, and the node is not the owner of the region, then it must become the owner by sending a message to the manager, which is forwarded to the current owner. If no operation is in progress on the owner, it responds immediately; otherwise, it queues the message and responds when the operation has completed. An up-to-date copy of the region's data is included in the response if necessary. After acquiring owenership of the region, the requesting node then sends invalidate messages to any other nodes with valid copies of the region. When a node receives an invalidate, it waits until any operation it has in progress on the region has completed, then marks its copy invalid and sends a response to the node that sent the invalidate. After all other copies of the page have been invalidated, the node initiating the write operation has the only valid copy of the region and can proceed.

2.2 The LRC Protocol

The page-based protocol used in our comparison is a multi-writer Lazy Release Consistency [10] protocol. Release Consistency [11] is a model that provides the same behavior as Sequential Consistency [12] for a large class of programs, while relaxing the guarantees that the shared memory system must provide. Release Consistency separates shared memory accesses into *ordinary* accesses and *synchronization* accesses. Synchronization accesses are further divided into *acquires* and *releases*. As originally described, Release Consistency requires that ordinary accesses be performed with respect to other processors only when the processor making the accesses performs a release. We refer to this as *Eager Release Consistency*.

Lazy Release Consistency further delays the time when ordinary accesses must be performed, to the point at which a processor other than the releasing processor performs an acquire on the same synchronization object. Furthermore, it guarantees only that the accesses will be performed with respect to the acquiring processor, not all processors in the system. If all of a program's competing accesses to shared memory are separated by synchronization, then the results under Eager or Lazy Release Consistency will be the same as they would have been under Sequential Consistency. By delaying consistency actions, Lazy Release Consistency is able to eliminate many message exchanges that would be necessary with a stronger consistency model. It also allows such optimizations as piggy-backing data movement onto synchronization messages.

Our Lazy Release Consistency protocol uses the virtual memory system to detect which memory pages are changed, and uses

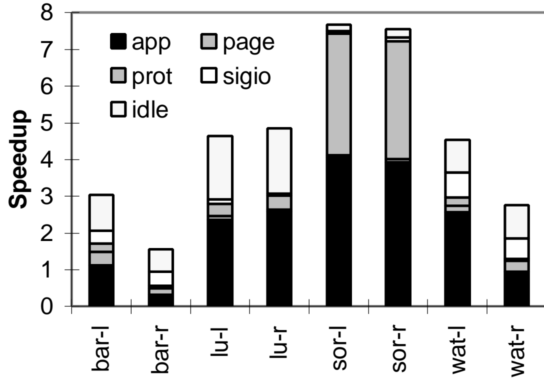

Figure 1: Speedup

diffs, which describe the changes that a processor has made to a page, to allow multiple concurrent writers to a single page. Diffs are generated by comparing the current contents of a page with a copy that was saved before the processor made any changes.

The protocol divides the execution of a parallel program into *intervals* on each node. New intervals begin each time a node performs a release or acquire synchronization operation. At the time of an acquire, the acquiring node sends a *vector timestamp* to the node that last released the same synchronization object. The vector timestamp describes which intervals the acquirer is aware of on each node. The releaser returns *write notices* for all intervals on all nodes of which it is aware but the acquirer is not. Write notices describe which pages were changed during the intervals in question. The acquirer invalidates these pages, and the next time one of them is accessed, it requests the diffs needed to bring the page up to date. The vector timestamp and write notices are piggy-backed on the messages that implement the release-acquire pair, so their use does not increase the number of message exchanges that are necessary.

3. Experiments

This section compares simulated performance of the CRL region-based protocol (hereafter referred to as "REGION") versus the multi-writer LRC protocol ("LRC"). While the protocols differ in many ways, we believe they are representative of the best object- and page-based protocols in general, and both are implemented in the same environment for these experiments.

3.1 Simulation Environment

The overall simulator design is discussed in Section 2. The simulator models processor detail only down to cycle counts based on instruction type; we do not simulate pipelines or multiple-issue. Therefore, it is not fair to say that our simulator reflects any single processor. However, we have roughly based our processor cycle counts and instruction costs on a 100-MHz POWER2 processor.

As a starting point for our experiments, we chose the following overheads: `mprotect` calls (used to change page protections by LRC) take 10 *u*secs, message sends require 80 *u*secs to start, plus 2 *u*secs for each byte of the message, and the cost of delivering a signal (such as SEGV, used to trap access-protection violations, or SIGIO, used to ensure timely handling of incoming messages) is 100 *u*secs. We then systematically varied these values to determine the sensitivity of overall performance to each variable.

All simulations were run on eight simulated processors.

3.2 Application Suite

We tested the two protocols against Barnes, Water-Nsquared, and LU from the SPLASH-2 [9] suite, and SOR, a simple red-black Jacobi application. The three SPLASH applications were taken from the CRL [3] distribution. These contained conditional compilation directives so that they could be used with CRL or with a sequentially consistent shared memory. To run them using the CVM region-based protocol, the applications were compiled using the (unchanged) conditional code for CRL. For lazy release consistency, they were compiled using the conditional code for a sequentially consistent shared memory, with slight changes.

All of the applications synchronize primarily through barriers, though Barnes and Water also use locks, and Barnes, Water, and LU all use small numbers of reductions. Although CVM's page-based protocols support global reductions, we disabled them in our simulator in order to have a purely page-based protocol. The first application, Barnes, uses the Barnes-Hut hierarchical N-body method to simulate the interactions of a number of bodies as determined by gravitational forces. The main data structure is an octree, the internal nodes of which represent space cells, and the leaves of which contain information about the bodies being simulated. The majority of the program's time is spent in a phase in which it repeatedly traverses this octree while computing the forces on each body [9]. This results in a fair amount of spatial locality, and a pattern of data accesses in which data is migratory rather than always accessed by a statically determined set of nodes. The statistics for Barnes were collected beginning at the end of the second time step, in order to eliminate one-time costs such as those associated with mapping a region for the first time in the region-based protocol (which requires a remote procedure call). Barnes was run using a problem size of 4096 molecules.

LU performs blocked LU factorization of dense matrices. The results in the paper are for a 512 x 512 matrix with 32 x 32 blocks.

SOR is a simple red-black Jacobi. Our grid was 1024 rows by 1024 columns, except for the runs in which we varied the page size, in which we used a 1024 by 512 grid because of memory constraints on our simulator. We collected statistics beginning with the second iteration.

Water computes the forces and potentials among a group of water molecules over a number of time steps. It maintains an array of structures that hold information about each molecule. The majority of the program's time is spent in a phase in which each processor computes the interactions of each of *n/p* molecules with the *n/2* molecules following them in the array [9]. This results in a high degree of spatial locality, and a pattern of data accesses in which the data for each molecule is accessed by a statically determined set of nodes. The data for Water was collected starting at the second time step. It was run for three time steps, using a problem size of 512 molecules.

3.3 Experiments

3.3.1 Overall Performance

Figure 1 shows the performance of our applications for each of the two protocols. This is measured in terms of speedup over running the same algorithm on a single processor (with no overhead for the synchronization calls that would be necessary on multiple processors).

Each bar is broken down into time spent executing application code ("app"); time spent validating data by requesting diffs under LRC or requesting copies of objects from other nodes in REGION ("page"); time spent in other protocol code ("prot");

time spent handling incoming requests (“sigio”); and time spent waiting on lock, reduction, or barrier replies (“idle”). “Page” overhead includes the latency of remote diff, region, and page requests. “Prot” overhead includes all time for lock, reduction, and barrier acquisitions, other than the idle time spent waiting for replies. The “-l” and “-r” suffixes indicate whether the bar pertains to the LRC or REGION protocol.

LRC outperforms REGION on two of the four applications, and closely matches REGION’s performance on the other two. The two applications that LRC performs particularly well on relative to REGION are Barnes and Water, the two applications with the highest communication volume and most fine-grained synchronization.

The dominant cost portion for both protocols, and all applications other than SOR, is idle time, which includes all delays on synchronization events. This is primarily lock delay for Barnes and Water, but barrier delay is significant in all cases because of load imbalance induced by uneven fault servicing.

The second largest source of overhead is consistency protocol code, which includes synchronization acquisitions. Neither protocol is preferred for consistency overhead. Although LRC’s protocol code is more complex, REGION’s protocol code is called more frequently due to the fact that all accesses to shared regions must be bracketed with calls to open and close read or write operations, regardless of whether or not those calls are also necessary for synchronization. For instance, between two barriers different processors may read and write prearranged disjoint sets of objects, in which case no calls to consistency code are needed under LRC. Under REGION, however, separate calls would be necessary to open and close read or write operations for each object.

SIGIO overhead runs a close third to consistency overhead. The dominant cost in this overhead is the operating system cost of calling the signal handler. The aggregate of this overhead is higher for REGION because region-based applications require many more remote data fetches than page-based applications (see Section 3.3.4). SIGIOs can be replaced with polling if communication is frequent enough [13], or if fine-grained polling can be inserted into application code by binary instrumentation [14]. “Sigio” also includes diff creation and copying for LRC and the overhead of dealing with requests concerning many small regions for REGION.

3.3.2 Varying Message Costs

Figure 2 shows protocol performance as the cost of messages (and the cost of SIGIO delivery) scales from zero communication overheads up to our default, in increments of 10%. LRC generally performs better relative to REGION with high message passing overhead costs, whereas REGION performs better with low overhead. This is unsurprising, as REGION's performance is determined largely by communication costs. Even with free messages, LRC must pay operating system overheads on each page modification that is trapped by SEGV handlers, and data is transferred only though diff creation and application. These costs are not dependent on communication performance. Note, however, that diffing is not intrinsic to the page-based approach. Shasta [14], for example, uses rewriting of binaries to catch all shared accesses.

LRC significantly outperforms REGION at high message costs for Barnes and Water. This is because REGION exchanges many more messages than LRC for these applications. We will examine why this is the case in section 3.3.4. For Water, REGION does not outperform LRC even when message passing is assigned zero overhead. The reason is that, even with zero cost messages, our simulation assumes that sending or receiving

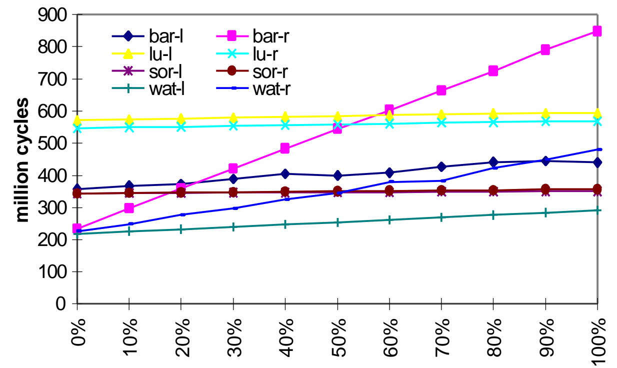

Figure 2: Varying Message Costs

a message will involve the overhead of copying the data into or out of a buffer. The average size of the regions allocated by the REGION version of Water is 668 bytes, but only a very small number of these bytes are changed by the program during each write operation. Because of this, LRC’s use of diffs allows it to send much less data than REGION; LRC sends 6,809,704 bytes, whereas REGION sends 11,303,152 bytes. Of course, LRC must examine the unchanged bytes when creating diffs. Once created, however, a diff can be used to update multiple processors.

Message costs had little effect on the total runtime of LU and SOR. For SOR, performance for both protocols was roughly the same, but for LU, REGION performed somewhat better than LRC over the entire range of message costs. This is explained by the fact that the regions identified to the REGION protocol are relatively large and exactly match the areas of memory that need to be communicated. This means that REGION does exactly the minimum necessary work, whereas LRC must spend time handling page faults and creating and applying diffs in order to obtain the information that is given to REGION by the programmer.

3.3.3 Varying Page Protection Costs

Figure 3 shows the performance of the LRC protocol as page protections costs are scaled from no cost to eight times our default cost. Runs of the applications under the REGION protocol are also included for comparison. For the runs described by this figure, message costs were fixed at one tenth of our default. In this environment, page protection costs take on greater importance to the performance of LRC.

As was seen earlier in Figure 2, with these message costs and the default page protection costs REGION outperforms LRC on all applications but Water, although for SOR there is little difference in running time between the two protocols. The running time of Water under LRC rises fairly slowly with page protection costs, only going above the running time using REGION at more than five times our default cost. Part of the reason for this is that (as mentioned in section 3.2) Water exhibits a high degree of spatial locality [9]. The rest of the explanation is that reads greatly outnumber writes of shared data, meaning that few page faults occur. This version of Water accumulates intermediate results in local memory, only updating global data at the end of a program phase. This leads to few page faults, because reading does not invalidate other processors’ copies of a page. Additionally, writing is often done to objects on the same page in succession, so that the page may stay writable while many objects are written (it is marked unwritable when it is invalidated or when a diff request comes in). The rate of page faults in LU and SOR is also low, resulting again in the run time growing slowly with page protection costs.

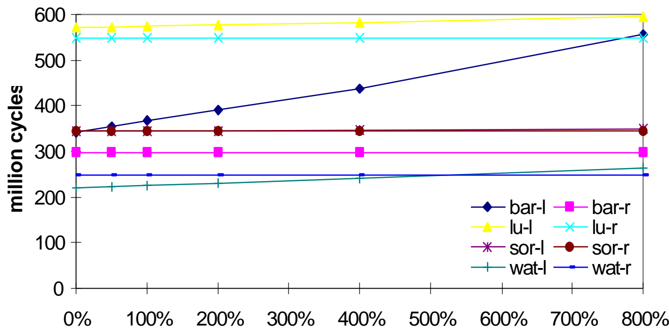

Figure 3: Varying Page Protection Costs

Of the four applications, Barnes is most severely affected by a change in page protection costs under LRC. Barnes uses finer grained sharing than the other applications, and does not have as much spatial locality, and therefore has a higher rate of page faults.

3.3.4 Varying Page Sizes

Table 3.1 compares per-process statistics for the two protocols as page size is varied from 512 up to 16k bytes. The rows for each application are time (in millions of cycles), remote misses, and kilobytes communicated. Conventional wisdom has long held that one of the main problems with page-based DSMs is that the large consistency granularity, i.e. the size of virtual memory pages, limits performance. This limitation is seemingly exacerbated by the current trend towards larger page sizes (usually to 8192 bytes), which clearly increase false sharing, and therefore consistency overhead as well. This is true for single-writer protocols, as increasing false sharing leads to increasing spurious contention for shared memory pages. However, multi-writer protocols such as LRC handle such situations without network communication. Hence, they do not suffer from the so-called *ping-pong* effect [15]. The remaining disadvantage of large page sizes in the context of multi-writer protocols is the extra overhead during the creation of small diffs.

Set against this disadvantage are several advantages. First, shared data adjacent to requested data arrives without explicit requests. If this data is later needed (i.e. the application has good spatial locality), the extra fetch is avoided. In addition to avoiding subsequent faults, this data aggregation avoids the high message startup costs that are common in the environments in which software DSMs are implemented.

Second, fewer diffs are created and less overhead is wasted in dealing with them. LRC implements a *lazy diffing* protocol, meaning that diffs are not actually created until they are requested by remote processors. Given sufficient spatial locality, processors often modify multiple objects on the same page, especially as page sizes increase. If these modifications are made prior to the arrival of requests for the earlier modifications, all of the modifications can be combined into a single diff.

Finally, larger page sizes usually reduce OS overhead. Changing page protections and calling signal handlers are relatively expensive operating system primitives. Larger page sizes mean that fewer of each occur.

Table 3.1 reveals the rather unintuitive fact that not only can increasing page size increase performance, but that this can happen even at page sizes beyond the 4k that is currently typical. The best performance for all four applications is seen with 8k pages. Information about performance at page sizes larger than 4k is particularly important, because software DSMs can easily use page sizes *larger* than the system page size merely by passing appropriate arguments to operating system calls. We have since confirmed this result by adding a command-line page size parameter to CVM, and other researchers have observed the same phenomenon [16]. Performance improves by an average of 25% between the 512 byte page size that is more typical of object size and 8k. Over the same span, 83% of remote misses are eliminated, while bandwidth requirements remain roughly the same.

Table 3.3 shows the results of an experiment that can be used to measure the efficiency of the prefetching effect more precisely. Rather than changing page sizes, we modified page fault behavior so that adjacent pages are validated at the same time as the page that caused the fault. This entails fetching diffs for adjacent pages at the same time as the target page, effectively achieving the same savings in messages and remote latency as increasing page sizes. The two methods differ in that the adjacent-page method allows the prefetched pages to be re-invalidated individually.

For example, if we have set a `prefetch_factor` of 4, and a read access to page 41 causes a segmentation violation, we also fetch diffs for the invalid pages among page 40, 42, and 43. If all three of these pages are initially invalid, then we say we have prefetched three pages. More importantly, we then track what happens to those pages next. A prefetch is useless if the page is re-invalidated before the next local access, or never used at all. The prefetch is considered useful if the page is accessed at least once before being invalidated.

The columns of Table 3.3 show remote misses and prefetches for three prefetching factors: 1 (control case), 2, and 4. The prefetches are divided into useless and useful categories as above. Our base page size is 4k bytes, so a prefetch factor of four corresponds to a prefetch of a 16k page.

Remote misses are divided into normal misses and *false misses.* A false miss is one that is induced by a larger page size. For example, if the system uses 4k pages and pages 40 through 43 are valid, an invalidation to page 43 will not affect subsequent accesses to some object *X* that resides on page 40. However, if we instead use 16k pages, the corresponding invalidation will also invalidate the portion of the shared space that contains object *X*. These *false misses* are one measure of the negative consistency implications of increasing page sizes. However, Table 3.3 shows that false misses occur only infrequently for our applications as effective page size moves from 4k to 16k bytes.

		LRC - Page Size						Region
		512	1024	2048	4096	8192	16384	Acquires
Barnes	time	580	517	477	457	439	457	849
	misses	2,446	1,718	1,244	915	655	482	2,699
	kbytes	1,131	1,161	1,160	1,170	1,178	1,201	786
LU	time	817	685	622	603	594	600	569
	misses	1,067	534	267	167	100	66	68
	kbytes	1,173	1,097	1,059	1,216	1,421	1,805	551
SOR	time	205	192	186	182	182	190	188
	misses	252	126	63	32	24	28	32
	kbytes	160	147	140	136	135	190	258
Water	time	413	353	320	298	291	304	481
	misses	1,627	953	494	268	154	114	1,528
	kbytes	990	902	800	832	732	848	1,290

Table 3.1: Varying Page Size

	1	Factor = 2				Factor = 4			
	Remote	Remote Misses		Prefetches		Remote Misses		Prefetches	
	Misses	Normal	False	Useful	Useless	Normal	False	Useful	Useless
Barnes	915	646	14	256	228	497	35	411	490
LU	167	160	0	67	6	157	0	98	34
SOR	32	32	16	0	11	32	24	0	28
Water	268	156	0	107	2	109	2	156	13

Table 3.3: Prefetch Factors (base page 4k bytes)

Overall, we find that the prefetching effect is extremely us e-ful. With a prefetch factor of four, nearly 42% of remote misses in Barnes, LU, and Water are eliminated. Even at a prefetch factor of four, 46% of all prefetched pages are useful. Prefetc h-ing is a complete loss for SOR, however, because only edge rows are communicated, and each row in our current array takes exactly one page. Prefetching would be a win for SOR if the rows overlapped more than a single page.

3.3.5 Message Traffic

Figure 1 shows that communication cost is the most signif i-cant source of overhead for REGION. Table 3.2 shows that the probable source of this overhead is that REGION uses far more messages than LRC. Since the cost of messages in our enviro n-ment is dominated by the startup cost, the number of messages is usually more important than the total bandwidth co nsumed.

In addition to showing the total messages and kilobytes sent for each application and protocol, Table 3.2 breaks messages down by type. For REGION, the types are barrier messages (“bar”), reduction messages (“reduc”), messages requesting read (“read”) or write (“write”) access to a region, and invalidate messages (“inval”). For LRC, the types are barrier messages (“bar”), lock messages (“lock”), and diff request messages (“diff”).

One of the primary issues that we wished to investigate was the bandwidth advantage of object-based protocols. In fact, Table 3.2 shows that this bandwidth advantage is not one-sided. While REGION benefits from sending only the data in the r e-quested object, LRC benefits by only sending the modified po r-tions of the objects that are sent. Barnes’ regions average only 102 bytes, and it appears efficient in this case to send entire objects, since REGION sends less data than LRC. In contrast, the average region size in Water is 668 bytes, and REGION sends 75% more data than LRC. This can be explained by noting that each Water phase modifies only a portion of each molecule, sometimes as little as 12 bytes. Sending the rest of the 600+ byte object is pure overhead.

However, sending the entire object is not intrinsic to the o b-ject approach. Systems such as Midway [2] and Shasta [14] use software dirty bits to create essentially the same diffs used by LRC. Such systems will send strictly less application data than LRC, although consistency overhead in messages may be larger.

Since synchronization messages dominate for LRC, it is u n-likely that the total number of messages consumed by LRC can be reduced significantly without changing the programming model, or using optimistic techniques.

3.3.6 Aggregating Regions

We ran a series of experiments using a modified version of REGION in which regions are aggregated into groups. Our tent was to investigate a way in which we might give REGION some of the benefits that large page sizes give to LRC, These groups are treated as units for the purposes of synchronization and co n-sistency. Requests for individual regions are transformed into requests for all regions in the same group.

This approach is not general; it may introduce deadlocks into applications that were previously free of them. This is because synchronization acquires to different regions, which wouldn’t have conflicted in the original protocol, will conflict in the mod i-fied version if they are to regions in the same group. For this re a-son, two of our four applications, LU and Barnes, could not be run under the modified prot ocol.

SOR slowed down when regions were aggregated. This is to be expected, since as mentioned in section 3.3.4, only edge rows are communicated, and a region is exactly one row. Water showed a significant improvement when small numbers of r e-gions were aggregated: with groups of ten regions, performance improved almost 36%. Performance began to degrade when more regions than this were grouped, but remained better than perfor m-ance with no aggregation until group size was increased to b e-tween 40 and 50 regions.

The regions allocated by Water average 668 bytes in length, so the group size of ten that showed the best performance gives a granularity of approximately 6k. At least two factors contribute to the fact that this is lower than the page size that performed best with LRC, which was 8k (with performance only slightly worse at 16k). First, REGION sends entire regions even when Water changes only a few bytes, and second, that there cannot be mult i-ple concurrent writers to different regions in a group whereas LRC allows multiple writers to a page. These factors also co n-tribute to the fact that the best performance of REIGON with aggregation was still 6% worse than the best performance seen with LRC.

4. Conclusions

This paper has investigated performance tradeoffs between object-based and page-based approaches to software distributed shared memory under a variety of simulated conditions. The r e-sults of our experiments are somewhat surprising: on average, the page-based approach out-performs the object-based approach unless message passing is very inexpensive. This is despite the fact that the object approach provides the underlying system with

	Region Messages						Kilo-	LRC Messages				Kilo-
	bar	reduc	write	inval	read	total	Bytes	bar	lock	diff	total	Bytes
Barnes	10	14	2,196	1,202	4,483	7,904	786	24	165	1,159	1,347	1,178
LU	44	2	0	0	103	148	551	46	0	106	152	1,421
SOR	34	0	54	0	54	142	258	34	0	24	58	135
Water	12	4	1,055	598	1,903	3,573	1,290	16	1,136	171	1,323	732

Table 3.2: Message Traffic

more application-specific information and is able to use an update protocol to combine data movement and synchronization.

The primary reason for this performance advantage is that the page-based approach has lower communication requirements, which are the dominant form of overhead on distributed computing platforms. As expected, we found that the page-based approach uses far fewer messages, as our applications all have significant locality. On the other hand, we found that the object-based approach (at least when sending whole objects, as in our implementation) did not have the expected bandwidth advantage. While REGION sends significantly less data than LRC for Barnes and LU, the converse is true for Water and SOR. REGION would have a bandwidth advantage over most page-based protocols because object validation entails sending only the requested object, rather than an entire page. However, LRC uses diffs to send only the modified portions of objects. These modifications may come from a superset of the objects sent by REGION, but some of the objects in the superset are later useful. In cases where the prefetching effect is small, the bandwidth advantage would unequivocally go to an object protocol that employed some form of diffing, such as Midway. As expected, the object protocol sent far more discrete messages than the page-based protocol.

Spatial locality has a large impact on both communication requirements and overall performance. Average performance improves by 25% as page size increases from 512 to 8k bytes. Remote misses decrease by 83%, while total data transferred remains roughly the same. Table 3.3 shows that the reason for the lack of increase in data traffic is that a large percentage of prefetched data is useful, and hence would have would be faulted across later anyway.

Beyond a simple comparison of two protocols, this paper has identified the major performance advantages of each style of protocol. For the object approach, the major advantages are the combination of data and synchronization into single messages, and the potential bandwidth advantages if complete objects are not sent. For the page-based approach, the clear win is reduced message traffic because of data that is effectively prefetched by the large page sizes.

However, these advantages can be combined. Various studies have shown that page-based algorithms can use heuristics to mimic the update characteristics of the object approach by predicting future accesses. The difference is that the heuristics do not always predict correctly, leading to lost opportunities to combine messages, and wasted bandwidth when unneeded data is sent.

The prefetch effect is not necessarily limited to data. Optimistic synchronization acquisition could potentially improve performance in page-based systems as well.

As discussed in Section 3.3.6, region aggregation could be performed for object-based protocols. This is demonstrated by our simple experiment in aggregating regions, in which we improved performance by 36% on one application by making our unit of consistency a group of ten regions. Region-based protocols require explicit associations to be made between data and synchronization. Hence, the underlying DSM can not blithely change region sizes without potentially violating assumptions made by the programmer. Any such violation could potentially cause deadlocks. However, the underlying system could make tentative acquisitions of "nearby" data. Correct region applications should not deadlock under this discipline as long as tentatively acquired regions are immediately released upon request. We will be exploring these ideas in future work.

5. Bibliography

[1] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," in *ACM Transactions on Computer Systems*, February 1988.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway Distributed Shared Memory System," in *Proceedings of the '93 CompCon Conference*, February 1993.

[3] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," in *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

[4] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," in *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[5] W. Yu, C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, pp. 18--28, February 1996.

[6] 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*, October 1991.

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

[8] A. Srivastava and A. Eustace, "ATOM: A system for Building Customized Program Analysis Tools," in *Proceedings of the SIGPLAN `94 Conference on Programming Language Design and Implementation*, May 1994.

[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] 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*, May 1992.

[11] 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*, May 1990.

[12] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558--565, July 1978.

[13] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[14] D. Scales and K. Gharachorloo, "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.

[15] B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[16] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory," in *Proceedings of the Principles and Practice of Parallel Programming*, 1997.