In the Journal of Parallel and Distributed Computing, 2004

Performance Analysis of Methods that Overcome False Sharing Effects in Software DSMs*†‡

K. V. Manjunath
Dept. of Electrical and Computer Engineering
University of Michigan, Ann Arbor, MI 48109, USA
kvman@umich.edu

R. Govindarajan
Dept. of Computer Science and Automation
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560 012, India
govind@csa.iisc.ernet.in

*This work was partially funded by IBM's Shared University Research programme.

†A shorter version of this paper has appeared in the Proceedings of the 8th International Conference on High Performance Computing (HiPC-2001), Hyderabad, Dec. 2001 (http://www.hipc.org).

‡This work was done when the first author was at the Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 560 012, India.

Abstract

Page based software DSMs experience high degrees of false sharing especially in irregular applications with fine grain sharing granularity. The overheads due to false sharing is considered to be a dominant factor limiting the performance of software DSMs. Several approaches have been proposed in the literature to reduce/eliminate false sharing. In this paper, we evaluate two of these approaches, *viz.,* the Multiple Writer approach and the emulated fine grain sharing (EmFiGS) approach. Our evaluation strategy is two pronged. First, we use an implementation independent analysis that uses overhead counts to compare the different approaches. Our analysis show that the benefits gained by eliminating false sharing are far outweighed by the performance penalty incurred due to the reduced exploitation of spatial locality in the EmFiGS approach. As a consequence, any implementation of the EmFiGS approach is likely to perform significantly worse than the Multiple Writer approach. Second, we use experimental evaluation to validate and complement our analysis. The experimental results match well with our analysis. Also the execution times of the application follow the same trend as in our analysis, reinforcing our conclusions. More specifically, the performance of the EmFiGS approach is significantly worse, by a factor of 1.5 to as much as 90 times, compared to the Multiple Writer approach. In many cases, the EmFiGS approach performs worse than even a single writer lazy release protocol which experiences very high overheads due to false sharing.

The performance of the EmFiGS approach remains worse than the Multiple Writer approach even after incorporating Tapeworm — a record and replay technique that fetches pages ahead of demand in an aggregated fashion — to alleviate the spatial locality effect. We next present the effect of asynchronous message handling on the performance of different methods. Finally, we investigate the inter-play between spatial locality exploitation and false sharing elimination with varying sharing granularities in the EmFiGS approach and report the tradeoffs.

Keywords: False Sharing, Memory Consistency Protocols, Performance Evaluation, Software DSM

1 Introduction

Software Distributed Shared Systems [4, 19, 21], that rely on the virtual memory mechanism provided by the Operating System for detecting accesses to shared data, support sharing granularity of page size. The large page size results is excessive *false sharing* overheads, especially in fine grain irregular applications [28]. Different methods have been proposed in literature to reduce the effects of false sharing in page-based software DSMs. Two basic approaches followed in these methods are (i) allowing concurrent write accesses to a page and (ii) providing fine grain granularity through emulation without additional architectural support. We refer to these approaches as the Multiple Writer approach and the Emulated Fine Grain Sharing (EmFiGS) approach respectively. We call the latter *emulation* because it provides fine grain sharing over a coarse grain sharing system and as a consequence incurs higher cost even for a fine grain coherence miss.

Lazy Multiple Writer Protocol (LMW) [16, 18] implemented in most state-of-the-art software DSMs [2, 20, 27] is a typical example of the Multiple Writer approach. It allows concurrent writes to a page by providing mechanisms to merge the modifications at synchronization points. LMW is considered heavy-weight because of the Twin/Diff creation overheads incurred to maintain and update the modifications. Writer Owns Protocol [8] improves upon LMW by performing run time re-mapping of subpages such that all subpages in a page are written by the same process. The sharing is still performed at the page level, but by remapping parts of the page, false sharing is eliminated. Millipage/Multiview [12, 11] follows the EmFiGS approach. It decreases the sharing granularity to less than a page by allowing smaller size pages, called *millipages*. However, to avoid wastage of physical memory, multiple millipages are mapped to a single physical page (of larger size). Millipage is implemented with single-writer protocol for efficient operation. Compile-time methods [9, 14] have also been proposed, following the EmFiGS approach, wherein data structures susceptible to false sharing are identified at compile-time and various transformations are applied to them to place them in different

pages, thus eliminating false sharing. While the Multiple Writer approach *tolerates* false sharing, the EmFiGS approach *eliminates* false sharing.

While it is true that these approaches are successful in reducing false sharing, they do not sufficiently address the following questions: (i) what are the additional costs incurred, if any, in reducing false sharing? and (ii) is the reduction in false sharing overheads significantly higher than the additional costs, thereby leading to overall performance improvement of the application? Also, while there have been performance evaluation of individual implementations of these approaches, there has been been no complete comparative analysis of these approaches which is required to understand the inter-play of overheads. This motivates our performance analysis work on methods that overcome false sharing in software DSMs.

Our performance evaluation strategy is two pronged. First, we present an implementation independent analysis to obtain the counts of various overheads incurred under different protocols. In our analysis, we have taken into account all the overheads incurred in most page-based DSMs, including the round trip message overhead, synchronization overhead, and the page fault kernel overhead incurred in entering the OS kernel to transfer control to SEGV handler. The overhead counts provide a basis for comparison of the different methods. Second, we augment this comparison with execution time results of the benchmarks under an actual implementation of the methods. The experimental evaluation complements the manual analysis by reporting the contributions of different overheads to execution time. It also helps to validate the manual analysis and to attribute costs to the overhead components. Last, the time incurred by certain overheads such as synchronization overheads depend on the actual runtime conditions and the application behavior. In these cases, the performance results obtained from our experimental evaluation presents a better picture. The following discussion presents a more specific account of the performance study.

We have considered those protocols for study that sufficiently represent the dominant approaches to

overcome false sharing, *viz.,* the Multiple Writer approach and the EmFiGS approach. In addition, we have considered the Lazy Single Writer Protocol (LSW) as the base case, since it can incur high false sharing overheads. Our application suite consists of three SPLASH2 benchmarks [29], *viz.,* BARNES, WATER-SPATIAL and RADIX, all exhibiting high false sharing behavior. First, our manual analysis reveals that, despite eliminating false sharing completely, the EmFiGS approach still incurs significantly higher number of page faults and messages compared to the Multiple Writer approach. This is because, with a smaller sharing granularity in the EmFiGS approach, the amount of spatial locality is reduced, more true sharing faults ocur. Our analysis indicates that the overheads incurred by the EmFiGS approach are significantly higher by a factor of 4 or more as compared to LMW. As our analysis is independent of any specific implementation and captures the overheads that are intrinsic to the protocol, we conclude that any implementation of the EmFiGS approach is likely to incur more true sharing overheads than the savings in false sharing.

Second, we use experimental evaluation to validate the manual analysis. Our experimental evaluation also augments the analysis by attributing costs to the overheads counts. Our experimental platform is IBM's Scalable Parallel (SP) architecture, running open source CVM [20], a software DSM system. The experimental results on overheads counts match closely with those obtained from manual analysis and reinforces our conclusions. The decreased exploitation of spatial locality manifesting as increased page faults and messages in the EmFiGS approach, results in a degradation of 1.5 to as much as 90 times compared to the Multiple Writer approach. Further, contrary to the popular belief about the heavy-weightedness of Multiple Writer protocols, the overheads of Twin/Diff creation in LMW are insignificant and contribute to less than 1% of the total overheads.

To alleviate the effects of spatial locality, we used Tapeworm [17] — a record and replay scheme to fetch pages ahead of demand in an aggregated fashion — in conjunction with the EmFiGs approach. While Tapeworm decreases the spatial locality effects, increased message sizes due to aggregation limits

its performance; *i.e.,* EmFiGs with Tapeworm performs only 20% better than plain EmFiGs, which is still worse than LMW. We also analyzed the tradeoffs between the reduced exploitation of spatial locality and the elimination of false sharing by varying the millipage size in the EmFiGs approach. We observe that at higher millipage sizes the effects of false sharing dominate while at lower millipage sizes, the effects of spatial locality dominate the overheads. A break-even between these two effects occur at millipage sizes of 1024-2048 bytes for the applications studied.

The rest of this paper is organized as follows. Section 2 discusses the protocols studied and the overheads considered in our analysis. Section 3 describes the manual analysis in detail and also presents the results of the analysis. Section 4 deals with the experimental results. In Section 5, we discuss the related work and provide concluding remarks in Section 6.

2 Protocols and Overheads

This section discusses the protocols studied and the overheads considered in our analysis.

2.1 Page-based Software DSMs

Page-based software DSM systems use the Virtual Memory (VM) hardware to detect accesses to shared locations and to maintain consistency. They use the `mprotect()` system call to set the access protection bits appropriately for a page, depending on whether the data in the page is currently consistent or not. Inconsistent pages are invalidated (read/write protected) with the `mprotect()` system call in a barrier or lock acquire synchronization call. Read/write access to a shared page that does not have the appropriate permission raises a *page fault* exception. The context is then switched to the OS kernel exception handler, which in turn raises the segmentation violation (SIGSEGV) signal.

Software DSM systems capture this signal by providing a signal handler. The handler initiates actions to make the faulting page consistent. This typically involves sending messages to other processes,

and getting responses. Once the data in the faulting page is made consistent, the `mprotect()` system call is used to make the page readable or writable depending on the shared access. The term `page fault kernel overhead` refers to the involvement of OS in transferring the execution control to the SIGSEGV handler while accessing a protected page. Further, throughout the discussion, the term page fault refers only to accesses to shared memory locations that do not have the appropriate permission.

2.2 Sequential Consistency (SC) Protocol

The SC protocol implements Sequential consistency [1], which guarantees a single sequential order among memory operations from all processors. The SC protocol is a single-writer, multiple-reader protocol, *i.e.,* it allows a shared page to have either a single writer or one or more readers, but readers and writers never co-exist at the same time. Before a write to a shared page (by a process) can take place, the readable copy of the page in other processes must be invalidated. SC incurs high overheads if two processes access non-overlapping data which lie within the same shared page, and one of the processes is modifying the data. In this case, the page shuttles back and forth between the two processes. However, SC is a light weight protocol, requiring very little protocol actions on a page fault or on a barrier. Last, SC does not have to maintain large data structures such as twins or diffs as in the case of multiple writer protocols (to be discussed in Section 2.4).

2.3 Lazy Single Writer (LSW) Protocol

The LSW protocol guarantees Lazy Release Consistency [18] (LRC). With LRC, the modifications of process A become visible to process B only after B synchronizes with A. A simple example of this is when process B acquires a lock originally acquired by process A, either directly from A or through a set of processes. LSW allows only one writer for a page at any given time. The processor holding the writable copy of the page is called the *owner*. Each page has a *version number*, which is incremented

every time ownership changes. On an acquire, the releasing process sends (i) a list of pages (*write notice*) it has modified along with their version numbers and (ii) a list of pages modified by other processes for which it has received write notices, to the acquirer. The acquirer invalidates the local copies of the pages that are in the write notice by disabling read/write permission through `mprotect()` system call. On a subsequent read access to the page, a read fault occurs and the process looks at all the write notices for the page and requests the page from the process that has sent the highest version number. Once the most recent version is received, it is copied into the appropriate location is given read permission.

A write access to a page without write permission, causes a write page fault. The faulting process sends ownership request to a statically assigned *manager* of the page, which forwards the request to the actual *owner*. In case the page has no *owner*, *i.e.,* there is no current writer to the page, the *manager* sends the ownership to the faulting process. Otherwise, the *owner* changes the permission for the page to read-only and sends the page along with the ownership to the faulting process directly. The faulting process, upon receiving the page, increases its version number. When two processes write to non-overlapping parts of a shared page repeatedly between two synchronization points, the ownership could be transfered a number of times between the processes. Since there is no true sharing of data between the two processes — as the writes are to non-overlapping parts of a shared page — this is referred to as *false sharing*. The transfer of ownership back and forth between the processes is called the *ping-pong* effect. The *ping-pong* effect causes high overheads in applications having a high degree of false sharing.

2.4 Lazy Multiple Writer (LMW) Protocol

Lazy Multiple Writer (LMW) protocol also guarantees Lazy Release Consistency [15]. However, LMW allows concurrent write accesses to a page and merges the modifications at synchronization points.

These modifications made by a process are maintained by twinning and diffing [6]. On the first write to a shared page by a process, a write fault occurs and a SIGSEGV is raised. In the SIGSEGV handler, which is a part of the software DSM layer, an identical copy of the page (called a *twin*) is created. At the next synchronization point, the twin is compared with the modified copy of the page to generate a *diff*, a record of modifications to the page. The LMW protocol is homeless in the sense that there is no *home node* assigned for a page where modifications could be flushed at every synchronization. In the homeless LMW protocol, a page fault causes page requests to be sent to all processes that modified the page before synchronization. The diffs are returned in response to the page request. The requesting process applies all the diffs in the order specified by their timestamps to reconstruct the consistent copy of the page. As LMW allows concurrent write accesses, once a process has a writable copy of a page, future accesses (read/write) to the page by other processes will not cause additional page faults until the next synchronization point. This way, LMW tolerates false sharing. We chose LMW protocol to represent the Multiple Writer approach in our analysis.

2.5 Millipage

Multiview/Millipage [12] is a software DSM system which overcomes false sharing effects following the EmFiGS approach. It is capable of manipulating the shared memory in variable sized blocks called *millipages*, whose sizes could be smaller than the operating system page size. The basic idea behind Millipage is illustrated in Figure 1. Consider an array of 1024 integer elements spanning one 4 kilobytes page. Suppose the application program is such that Process 1 writes to the first 512 elements and Process 2 writes to the last 512 elements. When the array is allocated in a single shared virtual page, writes by the two processes, under a single writer protocol, cause the page to be transferred back and forth between them. Millipage provides two *views* of the array by allocating the two halves of the array in two non-overlapping virtual address regions that fall in two different virtual pages as shown

in Figure 1. Since access permission is controlled through the virtual memory mechanism, protection and fault handling can be done independently for the two pages. Access to the different halves of the array will cause page faults for different virtual pages and consistency can be maintained for these pages individually. The two virtual pages are however mapped to the same physical page, so that no physical space is wasted. The non-contiguous virtual address regions are mapped to data structures by the use of indirection. Thus Millipage effectively reduces the sharing granularity to less than a page size and eliminates false sharing. Millipage uses light weight SC protocol for efficient operation.

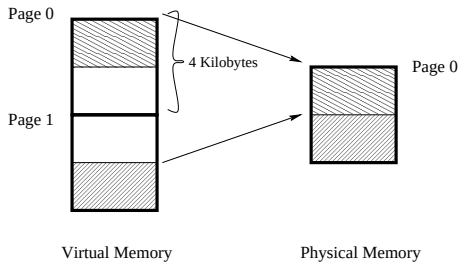


Figure 1: Illustration of Millipage

2.6 Array Bloating

Array Bloating is our alternative method for implementing the EmFiGS approach. In this method, we *pad and align* the elements of shared arrays so that a virtual page holds fewer elements (possibly only one) than it can accommodate, and non-overlapping regions of the array accessed by two different processes do not lie on the same virtual page. Virtual to physical page mapping is done in a manner similar to Millipage technique as shown in Figure 1. The example is same as the one described in Section 2.5. The array is bloated twice so that accesses to the first 512 elements (2 kilobytes) lie in virtual page k and those to the next 512 elements lie in virtual page $k + 1$ as shown in Figure 1. Thus we achieve an effective sharing granularity of 2 kilobytes. We use the term *bloat factor* to refer to the number times the data structure is bloated. To be precise, *bloat factor* refers to the number

of virtual pages that are mapped to the same physical page (refer to Figure 1). The *bloat factor* in this case is 2. The *bloat factor* also indicates the amount of data allocated per page. Higher the *bloat factor*, lesser the data allocated per page and hence lesser the false sharing. Access to appropriate elements of the array is accomplished by modifying the *sh_malloc* statements and instrumenting the array indices in the application program. Thus by reducing the sharing granularity to less than a page size, array bloating eliminates false sharing. Array bloating is less generalized than Millipage because it cannot be applied to irregular data structures with pointer references. However, the indirection overhead incurred by millipage is avoided in array bloating; but array bloating incurs overhead of array index calculation which is typically less costlier. In our analysis, we have considered array bloating implemented with LSW (ABLSW) and Sequential consistency protocol (ABSC) as methods following the EmFiGS approach.

2.7 Overheads

The overheads considered in our analysis are listed below. Each of these overheads would be incurred a number of times (overheads counts) and hence contribute certain time (overheads time) to the execution time. Our manual analysis deals with overheads counts, while the experimental evaluation reports overheads time.

Page fault kernel overhead: This is the overhead due to OS kernel involvement in page fault exception as discussed in Section 2.1.

Mprotect: This refers to the number of `mprotect()` system calls invoked.

Buffer copy: This is the overhead incurred in copying data from message data structures to the page and vice versa. This is over and above the various levels of copying that might be required by

the underlying messaging layer which is implementation dependent.

Twin creation, Diff creation, and Diff apply: These overheads are specific to LMW and are incurred in maintaining modifications to a shared page.

Message Overhead: A process sends messages to other processes on various events like a page fault, a lock acquire, or a barrier synchronization. A message arriving at the process causes consistency protocol actions to be performed and subsequently a reply to be sent to the requesting processor. Further, since messages arrive asynchronously at the receiver, some overheads are incurred at the messaging layer due to polling/interrupt handling. We refer to the time elapsed between the sending of the request and receiving the reply as *message time*, which includes the round trip message latency, the message processing time at receiver and other messaging layer overheads.

Synchronization/Barrier Overhead: This includes the calls to lock acquire/release and barrier synchronization and the associated overheads involved in performing the consistency protocol actions.

The above overheads represent all overheads that occur in page-based DSM systems. That the list is exhaustive can be inferred from our experimental results (described in Section 4) where the fraction of execution time of the benchmarks contributed by these overheads accounts for rest of the execution time other than application time.

3 Manual Analysis

In this section we describe our manual analysis of the overheads incurred in Multiple Writer and EmFiGS approaches.

3.1 The Method

We illustrate our analysis with a simple example. We are interested in those statements of the application program that can possibly cause any of the overheads listed in Section 2.7. These are the statements containing shared memory references and synchronization points. In software DSM programs, synchronization is achieved by an explicit function call to the underlying software layer. We inspect the benchmark source code to separate these two kinds of statements, *viz.,* shared memory accesses and synchronization points, and also the control structures of the program using an approach similar to program slicing.

```
//begin and end mark the rows
//each process is responsible
//for computing
while (cond) do
        for i := begin to end do
            .. = shared_array[i]
            shared_array[i] = ..
            shared_array[i] = ..
        od
        Barrier()
        if pid == 0
          for i := 0 to 1024 do
              .. = shared_array[i]
          od
        fi
        Barrier()
end
```

(a) Sliced program

Proc 0	Proc 1
while (*cond*) **do**	**while** (*cond*) **do**
for *i* := *begin* **to** *end* **do**	**for** *i* := *begin* **to** *end* **do**
Read Page 0	*Read Page* 0
Write Page 0	*Write Page* 0
Write Page 0	*Write Page* 0
od	**od**
Barrier()	*Barrier*()
if *pid* == 0	**if** *pid* == 0
for *i* := 0 **to** 1024 **do**	
Read Page 0	
od	
fi	**fi**
Barrier()	*Barrier*()
end	**end**

(b) Page access sequence

Figure 2: Sliced program and page access sequence

Figure 2(a) shows such a sliced program. Assume we have two processes. In this program, we have a read followed by two writes to a shared array by both processes. After a *Barrier*, process 0 reads

the entire shared array and finally, there is a Barrier synchronization. These accesses and Barriers are executed repeatedly in a loop. Assume that the array spans one shared page, and that process 0 accesses the first half of the page and process 1 accesses the next half. The portions of the array accessed by the two processes are specified by *begin* and *end* (local) variables in the program. Since both processes *write* to non-overlapping parts of the same page, there is a write-write false sharing in this case. By analyzing the source code, we arrive at a sequence of accesses to actual shared pages in different processes. Figure 2(b) shows such a sequence on 2 processors. For this step, the mapping of shared data structures to shared pages is to be known. This can be easily obtained from the source code, by knowing the order of *sh_malloc* function calls. We assume that the *sh_malloc* for the array is aligned to a page boundary, and *sh_mallocs* are contiguous in the virtual address space. We describe the analyses for LSW and ABLSW in the following subsections. The analyses for LMW and ABSC are similar and can be found in [22].

3.1.1 Analysis for LSW

Figure 3 illustrates the overheads calculation for LSW for a single iteration of the `while` loop shown in Figure 2. Different temporal interleavings of the sequences of shared accesses will cause different amounts of overheads. In our analysis, we calculate the minimum and maximum overheads that would be incurred due to these interleavings. Let us start with the case where both the processes have readable copy of the shared page and Proc 1 be the current owner of the page. (The reasons for this initial condition will be explained towards the end of this discussion.) The minimum overhead occur when only the first access to an invalidated shared page causes a page fault and the process finishes all its accesses to the shared page before it gets invalidated due to another process's access. Such a case is shown in Figure 3(a). Let us assume Proc 0 completes all its accesses before Proc 1 [1].

[1] If the accesses of Proc 1 take place before that of Proc 0, then Proc 1 will have the ownership and overheads incurred in this case are symmetric.

Proc 0	Proc 1
Read page 0	
Write page 0	
// Page fault for page 0	
Message to Proc 1	
	Ownership to Proc 0
Mprotect(write) page 0	
Write page 0	
// Read/Writes to page 0	
// without any pagefault	
// in the next 511 iterations	
	Read Page 0
	Write page 0
	// Page fault for page 0
	Message to Proc 0
Mprotect(read) page 0	
Buffer copy	
Ownership to Proc 1	
	Mprotect(write) page 0
	Buffer copy
	Write page 0
	// Read/Writes to page 0
	// with no pagefault in
	// the next 511 iterations
Barrier	Barrier
	Message to Proc 0
Mprotect(none) page 0	Mprotect(read) page 0
Read Page 0	
// Page fault for page 0	
Message to Proc 1	
	Buffer copy
Mprotect(write) page 0	
Buffer copy	
Mprotect(read) page 0	
Barrier	Barrier
	Message to Proc 0

(a) Best case

Proc 0	Proc 1
Read page 0	
	Read page 0
Write page 0	
// Page fault for page 0	
Message to Proc 1	
	Ownership to Proc 0
Mprotect(write) page 0	
	Write page 0
	// Page fault for page 0
	Message to Proc 0
Mprotect(read) page 0	
Buffer Copy	
Ownership to Proc 1	
	Mprotect(write) page 0
	Buffer Copy
Write page 0	
// Page fault for page 0	
Message to Proc 1	
	Mprotect(read) page 0
	Buffer Copy
	Ownership to Proc 0
Mprotect(write) page 0	
Buffer Copy	
	Write page 0
	// Page fault for page 0
	Message to Proc 0
Mprotect(read) page 0	
Buffer Copy	
Ownership to Proc 1	
	Mprotect(write) page 0
	Buffer Copy
⇑ Repeated 512 times ⇑	
Barrier	Barrier
	Message to Proc 0
Mprotect(none) page 0	Mprotect(read) page 0
Read Page 0	
// Page fault for page 0	
Message to Proc 1	
	Buffer copy
Mprotect(write) page 0	
Buffer copy	
Mprotect(read) page 0	
Barrier	Barrier
	Message to Proc 0

(b) Worst case

Overheads (in Proc 0)	Overhead counts	
	Best Case	Worst Case
No. of Page faults	2	1025
No. of Messages	2	1026
No. of Mprotects	5	1537
No. of Bcopy	2	1537

(c) Overheads

Figure 3: Overheads calculation for LSW

The first read access to page 0 by Proc 0 does not cause a page fault because it has a readable copy. However, the first write access by Proc 0 causes a write page fault. In the general case, a write page fault causes a message to be sent to the *manager* of the page which will forward ownership request to the current *owner* of the page. Let us assume that Proc 0 is the *manager* of the page 0, and hence it knows Proc 1 is the current *owner*. Proc 0 sends ownership request to Proc 1 and on reply from Proc 1, Proc 0 becomes the *owner* for page 0. This incurs a page fault kernel overhead and a message as shown in Figure 3(a). On receiving the ownership, page 0 is `mprotected` with write permission and Proc 0 proceeds to write to it. Hence, the next write access by Proc 0 does not cause a page fault. Note that there are no intervening accesses to page 0 from Proc 1 until this point in the best case situation.

Next the read access by Proc 1 takes place without a page fault as Proc 1 still it has a readable copy of the page. However, the next write access causes a page fault. Since Proc 1 is not the current *owner* of the page, it sends a page request to the *manager*, which is Proc 0. Since Proc 0 is the current owner also, it transfers the page along with the ownership to Proc 1, and removes the write permission from its own copy of page 0, making it read-only. The page is `mprotected` with write permission at Proc 1. The subsequent writes take place without any page fault at Proc 1.

At a barrier synchronization, a process assigned as the *barrier manager* has to collect the write notices from other processes. Let us assume that Proc 0 is the *barrier manager*. At the first barrier in the code, Proc 1 sends a write notice for page 0 to Proc 0. Consequently, the page is invalidated at Proc 0 and made read-only at Proc 1. The next read access at Proc 0 causes a page fault, resulting in a page request being sent to the current *owner*, *viz.,* Proc 1. Upon receiving the page, Proc 0 first makes the page writable, just in order to copy the page data from the message. This write does not require ownership as it is only for making the shared page consistent. Then the page is made read-only at Proc 0. All the other reads to the page in the loop proceed without a page fault. At the next barrier, there are no write notices sent because there were no writes in the previous interval. All overheads

incurred during each of the above steps are shown in Figure 3(c). Note that Proc 1 is still the *owner* of the page. Given this access sequence in an iteration, Proc 1 will be the *owner* for the page at the beginning of next iteration. This is the rationale behind our assumptions on initial conditions.

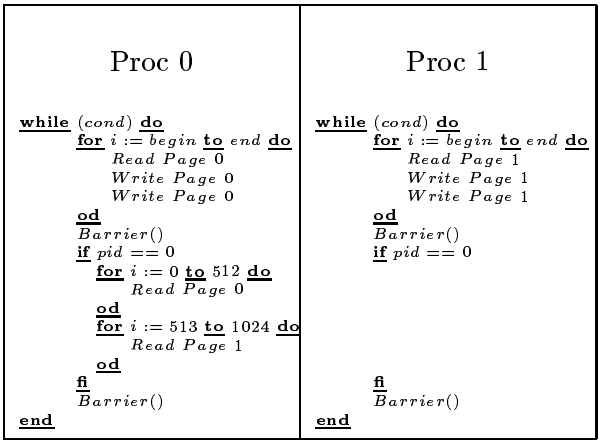
The maximum overheads counts occur when the accesses from different processes are completely interleaved. Such a complete interleaving for a single iteration of the `while` loop is shown in Figure 3(b). We observe that the maximum overheads occur when after every write access by Proc 0, there is an intervening write access by Proc 1, causing the ownership to be transferred back and forth between the two processes.[2] This sequence is repeated for all the iterations of the inner `for` loop in the worst case. We calculate the overheads following steps similar to the ones described above. The overheads incurred in the best LSW and the worst case situations for LSW are summarized in Figure 3(c).

3.1.2 Analysis for ABLSW

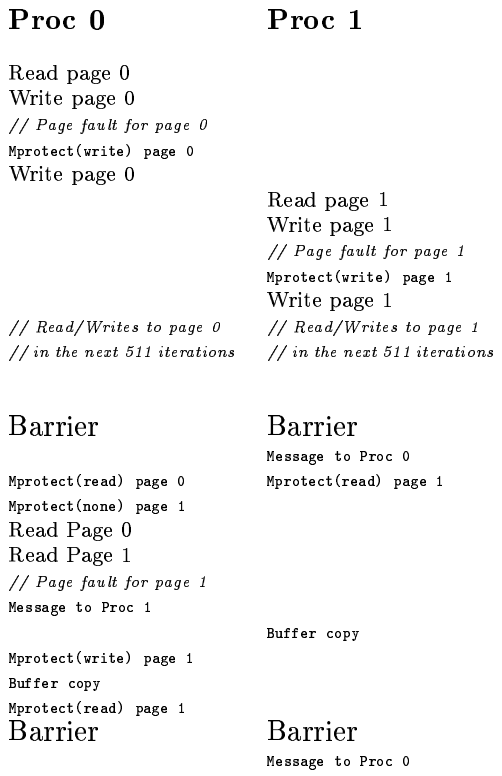
The overhead analysis for ABLSW is shown in Figure 4. The shared array in this case is bloated twice so that the first half gets allocated in virtual page 0 and the next half in virtual page 1. Hence the accesses to the shared array by the two processes fall in two different pages as shown in Figure 4(a).

After the first iteration, Proc 0 becomes owner for page 0 and Proc 1 for page 1 because of the write accesses to the respective pages. After the first barrier, the pages become read-only at the respective processes. Hence the first write access in the next iteration of the `while` loop causes a page fault. However there are no messages and ownership transfers in this case. The only page transfer occurs when Proc 0 tries to read page 1 in the final step of the iteration. The overheads calculation for ABLSW is shown in Figure 4(b) and summarized in Figure 4(c).

[2] A formal proof establishing that such a sequence causes the maximum overhead is beyond the scope of this paper.



(a) Page access sequence



(b) ABLSW Analysis

Overheads (in Proc 0)	Overheads counts
No. of Page faults	2
No. of Messages	1
No. of Mprotects	5
No. of Bcopy	1

(c)Overheads

Figure 4: Overheads calculation for ABLSW

3.1.3 Remarks

We conclude our discussion on the manual analysis with a few remarks. As was discussed in Sections 3.1.1 and 3.1.2, different temporal ordering of accesses to shared pages will cause different amounts of overheads. Our analysis estimates the minimum and maximum overheads that would be incurred in those cases. Similarly, when shared accesses occur inside a conditional statement, it is not possible to calculate the exact overhead counts. Our manual analysis can incorporate probabilistic assumptions about such shared accesses, using profile information, and can estimate the overhead counts

based on the assumed probabilities. Although such an approach has not been incorporated in our manual analysis, this is a simple extension to our analysis. In fact, in two of the benchmarks programs used, there are shared memory accesses inside conditional statements. However, since the conditional statements were true only in a small percentage of the cases in our analysis, we ignored the shared accesses within the conditional.

3.2 Benchmarks

This section presents a brief description of the benchmarks we have analyzed. The benchmarks analyzed and their problem sizes are listed in Table 1.

BARNES: This application simulates the interaction of a system of bodies in three dimensions over a number of time steps using the Barnes-Hut hierarchical N-body method [10]. The bodies are represented in an octree and the application spends most of the time traversing the octree to compute forces on the bodies. The resulting access pattern is quite irregular. Further the data structure representing a body, which is the basic unit of memory access in the algorithm, has a size of 96 bytes. Thus multiple bodies (about 42) gets allocated in a single shared page of 4 kilobytes size. When these bodies are accessed by different processes, significant false sharing behavior is exhibited in page-based DSMs. There is also significant true sharing in each iteration, when process 0 reads all the bodies, once the forces have been calculated on them, to compute their global positions. The domain decomposition in this application is spatial and across iterations bodies can move from one region of space to another, resulting in a different process accessing the body. However, such an occurrence is less probable, and in our analysis, for simplicity, we have assumed that bodies do not move over to different process in any iteration. Also our experimental results (reported in Section 4.1) indicate that the inaccuracies due to this assumption are minor. In our ABLSW and ABSC analysis, the data structure for a body is padded to 128 bytes and we consider an allocation of one body per shared page, *i.e.,* a *bloat factor* of 32. We

consider other bloat factors and their effects on the performance of the application in Section 4.5.

WATER-SPATIAL: This application evaluates the forces and potential that occur over time on a system of water molecules using a $O(n)$ algorithm [29]. It imposes a uniform 3-D grid of cells on the problem domain and the domain decomposition is spatial in this application. The molecules are allocated in an array and each process gets a group of elements to operate on. It happens that many shared pages have elements accessed by 2 or more processes that update them, causing write-write false sharing. This happens in the intra-molecular forces calculation phase. Also, when processes read the neighboring molecules' data to compute inter-molecular forces, read-write false sharing[3] occurs. Molecules move over to neighboring spatial domain occasionally in few iterations. Since such an occurrence is rare, in our analysis, we have assumed molecules do not move over. The data structure representing a molecule is 384 bytes in size and our ABLSW and ABSC analysis assumes padding of molecule data structure to 512 bytes and an allocation of one molecule per shared page, *i.e.,* a *bloat factor* of 8.

Benchmark	*Problem size*
BARNES	1024 bodies, 40 iterations
WATER-SPATIAL	64 molecules, 400 iterations
RADIX	16384 keys, 256 radix

Table 1: Benchmarks and problem sizes

RADIX: The radix sort is based on the method described in [5]. The algorithm is iterative, performing one iteration for each radix r digits of the keys. In each iteration, the histograms calculated locally are accumulated into a global histogram, using which the keys are permuted into a new array which is used in the subsequent iteration. We have considered a key size of 64 bytes. The permutation

[3]Read-write false sharing occurs when two processes access non-overlapping parts of a shared page and one of the processes writes to the page and the other reads the page. The first read by the reading process would cause the page to become read-only at the writing process in the case of LSW. Hence the next write access at the writing process will cause a page fault. However the rest of the accesses do not cause page faults

step exhibits highly irregular access pattern and high degree of false sharing. In the ABLSW and ABSC implementation, we bloat each key (64 bytes) to a shared page (4096 bytes), *i.e.,* a *bloat factor* of 64. This eliminates false sharing completely in the permutation step.

3.3 Why Manual Analysis?

Before presenting the results of our manual analysis, we address the following two questions : (i) Why perform (manual) analysis if experimental evaluation of the different approaches can be conducted on some platform? (ii) Do these overheads costs reported here directly translate into execution time? We counter the first question with the argument that if the underlying implementation used for experimental evaluation incurs certain overheads that are specific to that implementation of the protocol, rather than being intrinsic to the protocol itself, then the experimental evaluation is somewhat biased. We note one such overhead in Section 4.1. Our implementation independent analysis presents a true picture by considering only the overheads that are intrinsic to the protocol. However, an important caveat here regarding the manual analysis is that one should apply certain caution in comparing overheads counts, especially when the contribution to the execution time could be different under different protocols. For example, the overheads time for synchronization and message overheads can be different for two methods, even when the overheads counts are identical. Nonetheless, comparing overheads counts, obtained from an implementation independent analysis, does provide a fair basis for comparing different methods.

The above caveat is essentially what is referred to by question (ii) above. We address this question by performing the experimental evaluation of all the methods on the same platform which not only complements our manual analysis results, but also attributes costs (overheads time) for the various overheads. We present the results of the experimental evaluation in Section 4. Last, certain limitations of our manual analysis are discussed in Section 3.5.

3.4 Results

The results of our analysis on the overheads counts in different protocols for the three applications are presented in Tables 2, 3 and 4. We have considered a 4-process version of the applications and the overheads shown are incurred by process 0. The overheads incurred on other processes exhibit a similar behavior. Each row in the table indicate the number of times a particular overhead is incurred in the different protocols. For example, 3362 page fault kernel overheads are incurred by BARNES in LMW protocol. It can be observed from the discussion in Section 3.1 that this overhead analysis is independent of any specific implementation. For LSW, we have reported the best case and the worst case overheads. It should be noted that the results reported are based on the analysis of the entire application.

Overheads	LMW	LSW		ABLSW	ABSC
		Best case	Worst case		
Page Faults	3362	4797	34809	54407	58507
Messages	3684	3567	35055	31693	58589
Barriers	82	82	82	82	82
Mprotect	6683	9512	54120	108732	116932
Twin/Diff	7339	-	-	-	-

Table 2: Overheads in BARNES

Overheads	LMW	LSW		ABLSW		ABSC	
		Best case	Worst case	Best case	Worst case	Best case	Worst case
Page Faults	19200	18400	140000	34800	73200	66000	104400
Messages	18000	22000	143600	41200	79600	194000	347600
Barriers	3602	3602	3602	3602	3602	3602	3602
Mprotect	38800	35600	208400	83200	147200	153594	236800
Twin/Diff	32400	-	-	-	-	-	-

Table 3: Overheads in WATER-SPATIAL

A consistent trend observed in all the applications is that the overheads in ABLSW and ABSC are significantly higher than in LMW or in best case LSW. This is somewhat surprising given that false sharing is completely eliminated in ABLSW and ABSC, by reducing the sharing granularity to the

Overheads	LMW	LSW		ABLSW	ABSC
		Best case	Worst case		
Page Faults	1049	1372	4194652	28000	28000
Messages	1050	1053	3146032	28708	60000
Barriers	22	22	22	22	22
Mprotect	2754	2680	4195960	61496	65536
Twin/Diff	2734	-	-	-	-

Table 4: Overheads in RADIX

level of basic units of memory access in the applications; i.e., each shared page consists of only one body, one molecule, or one key. The number of page fault kernel overheads and the message overheads incurred in ABLSW are higher at least by a factor of 2 in WATER-SPATIAL and by a factor of 10 in other applications compared to LMW.

Why should ABLSW or the EmFiGS approach incur such high page fault overheads given that it eliminates false sharing completely? To answer this question, first let us note that the applications we have analyzed do contain true sharing behavior wherein a single process (such as Proc 0) reads the entire shared array after force calculation on all bodies/molecules were complete (similar to the read in Proc 0 between the two barriers in the example shown in Figure 2). Also, more than one body or molecule accessed by a process are co-located in the same shared page in the case of LMW, and a single page fault causes the entire page to be validated in the Multiple Writer approach, *i.e.,* a single page fault brings in the entire page (several bodies/molecules/keys). Whereas in the EmFiGS approach, due to the emulation of fine grain sharing, each page fault brings in a lesser amount of data (only one body, molecule, or key) on a true sharing miss. This results in a lower granularity system requiring more page faults than a higher granularity system to bring in the same amount of data. In other words, the amount of *spatial locality* exploited also reduces with reduction in sharing granularity. In the above cases, the Multiple Writer approach can gain by exploiting spatial locality, whereas the EmFiGS approach exploits significantly less spatial locality. This reduction in exploitation of spatial locality is the principal reason for ABLSW and ABSC having more page faults and consequently more

associated overheads like messages and `mprotects` than LMW or the best case LSW.

By tolerating false sharing, LMW incurs less page faults, messages, and mprotects than other methods. However, it incurs additional overheads of twin/diff create and diff apply which are not incurred in other protocols. It can be seen that, if the costs of the overheads of twin/diff create and diff apply are not significant, LMW is likely to perform better than other methods.

In WATER-SPATIAL, the inter molecular forces computation phase exhibits read-write false sharing, *i.e.,* a molecule's data structure is read by other processes while a different part of it is being updated by its owner. Any such intervening read by other processes would cause the molecule to become read-only in its owner, causing the subsequent write to result in a page fault. However, if the reads do not intervene during the writes, these page faults would not occur. Due to this behavior, different interleaving of reads/writes incur varying overheads in ABLSW and ABSC in the case of WATER-SPATIAL. As a consequence, we report the best and the worst case overheads for ABLSW and ABSC too for WATER-SPATIAL. Whereas BARNES and RADIX, which do not have read-write false sharing, are not affected by the interleaving reads/writes under the ABLSW and ABSC protocols.

Except in BARNES, the worst case overheads of LSW are significantly higher than the overheads in ABLSW and ABSC. This indicates that the *ping-pong* effect could occur to a large extent in these applications. If this happens, then ABLSW and ABSC, which completely eliminate false sharing, could possibly perform better than LSW. However, if the actual overheads fall closer to the best case in LSW, as our experimental results indicate, then LSW will perform better than ABLSW and ABSC in all the three applications.

In conclusion, we observe that the EmFiGS approach (ABLSW and ABSC) incur significantly higher overheads (by a factor of as much as 90) than the Multiple Writer approach. Since our analysis is implementation independent, we conclude that *any* implementation of EmFiGS such as ABLSW, ABSC, or even Millipage/Multiview will incur more true sharing overheads than the savings in false

sharing.

3.5 Remarks

Even though our manual analysis presents a true picture by considering the overheads that are intrinsic to the protocol, the analysis is not without limitations. First of all, it should be noted that the manual analysis is performed by inspection of the source code of the benchmarks. Therefore, availability of the source code is a primary requirement for our manual analysis. The manual analysis requires the knowledge of how the accesses to shared data structures translate to accesses to shared pages. This is easy to obtain if the accesses are straightforward, as in the case of array accesses. When the accesses to shared data structures are through pointers, then the manual analysis becomes complex and tedious. Next, it should be remarked here that our manual analysis takes into consideration only the major overheads incurred in the applications, like page faults from a software DSM perspective. However, it does not take into account other system related (finer) overheads like TLB misses and cache misses. Further, we can compare the counts of overheads incurred by two different methods directly, only if the overheads themselves are identical. For example, the twin/diff overheads which are only incurred in LMW, or the message and synchronization overheads which incur different amounts of execution time in different methods are difficult to compare. In such cases, the conclusions we can draw from the overheads counts are somewhat limited. Therefore, care should be taken in avoiding the caveats of manual analysis described above, while interpreting the results of the analysis. To address some of these issues and to attribute costs, in terms of execution time, for the different overheads, we present the experimental evaluation of the different methods in the following section. The experimental evaluation complements our manual analysis by presenting a complete picture and helps draw a better comparison of the different methods. Thus the manual analysis and the experimental evaluation go hand-in-hand in presenting a complete comparative performance evaluation of the methods.

4 Experimental Evaluation

This section presents our experimental evaluation of the benchmarks. Our experimental platform is a 12 node IBM SP2 connected by a high performance switch. The SP2 system consists of a number of POWER2 Architecture RISC System/6000 processor nodes each with its own main memory and its own copy of AIX operating system. The page size is 4 kilobytes. We used CVM [20], an open source software DSM system for our experiments. CVM uses MPI as the underlying messaging layer. We compiled CVM with MPL, which is IBM's proprietary implementation of the MPI standard. CVM supports LMW, LSW and Sequential Consistency protocols. We implemented the array bloating technique in CVM. Although in principle array bloating can be achieved using compiler instrumentation, we manually instrumented the benchmarks to support array bloating. Also, we have suitably modified CVM to obtain detailed break-ups of the overhead times.

Throughout this section, we present and discuss the results of the three benchmarks run on 4 processors for the input sizes shown in Table 1. Each of these benchmarks was also run on 2, 8, and 12 processors, and for a larger input size. These results show a similar trend and are not presented here due to space limitations.

4.1 Validation of Manual Analysis

Tables 5, 6, and 7 show the overheads incurred by the applications in the 4 processor case under different protocols. The column marked "Expt." shows the measured values of the overheads incurred by process 0 when the applications were run on CVM.

We see that the experimental values of all the overheads, except `mprotect`, match well and fall within 10% of those obtained from our manual analysis. A part of this (minor) deviation is due to the simplifying assumption — molecules/bodies do not move from one process to another (refer to Section 3.2) — in our analysis. The difference in the `mprotect` overhead can be reasoned as follows. In

Overheads	LMW		LSW			ABLSW		ABSC	
	Analysis	Expt.	Best case	Worst case	Expt.	Analysis	Expt.	Analysis	Expt.
Page Faults	3362	3425	4797	34809	15393	54407	55833	58507	59104
Messages	3684	3963	3567	35055	14654	31693	31202	58589	89126
Barriers	82	82	82	82	82	82	82	82	82
Mprotect	6683	12004	9512	54120	37219	108732	148077	116932	154817
Twin/Diff	7339	8002	-	-	-	-	-	-	-

Table 5: Overheads in BARNES — Comparison of Analysis vs. Experimental

Overheads	LMW		LSW			ABLSW			ABSC		
	Analysis	Expt.	Best case	Worst case	Expt.	Best case	Worst case	Expt.	Best case	Worst case	Expt.
Page Faults	19200	20333	18400	140000	27791	34800	73200	54535	66000	104400	82104
Messages	18000	19435	22000	143600	28536	41200	79600	42443	194000	347600	315469
Barriers	3602	3602	3602	3602	3602	3602	3602	3602	3602	3602	3602
Mprotect	38800	51288	35600	208400	70048	83200	147200	147200	153594	236800	226869
Twin/Diff	32400	43359	-	-	-	-	-	-	-	-	-

Table 6: Overheads in WATER-SPATIAL — Comparison of Analysis vs. Experimental

Overheads	LMW		LSW			ABLSW		ABSC	
	Analysis	Expt.	Best case	Worst case	Expt.	Analysis	Expt.	Analysis	Expt.
Page Faults	1049	1193	1372	4194652	9770	28000	28102	28000	28102
Messages	1050	1284	1053	3146032	9558	28708	18359	60000	60426
Barriers	22	22	22	22	22	22	22	22	22
Mprotect	2754	7855	2680	4195960	25568	61496	129342	65536	86404
Twin/Diff	2734	3628	-	-	-	-	-	-	-

Table 7: Overheads in RADIX — Comparison of Analysis vs. Experimental

CVM, the `mprotect()` system call is used to invalidate a page at a barrier, and it is called once for each write notice. Suppose a page has k write notices, then the `mprotect()` system call is invoked k times, even though the first call is sufficient to invalidate the page. In our manual analysis, we have have correctly accounted only one `mprotect` for an invalidation. Consequently, the experimental values of `mprotect` deviates by 50% –100% from the values obtained from our analysis. This is one example of an implementation dependent overhead which is very specific to the way CVM implements invalidation. Note that by not including such implementation dependent overheads, our manual analysis presents a true comparison of the different protocols.

The overheads in LSW measured in the experiments fall within the minimum and maximum values calculated in our analysis, and are closer to the best case overhead values. The overheads in the worst case LSW, calculated from our manual analysis are approximately 2, 5 and 400 times the measured experimental values in BARNES, WATER-SPATIAL, and RADIX respectively.

Comparing the overheads measured from our experimental evaluation, we can observe that LMW incurs significantly less overhead than ABLSW (by a factor of 2.5 to 15) or ABSC (by a factor of 4 to 20). LSW, despite experiencing considerable false sharing, incurs less overheads, less by a factor of 1.5 to 3 than ABLSW in all applications.

4.2 Execution Time Results

In this section, we address the question, "how do the observed overhead counts translate into execution times?".

4.2.1 The Method

Figure 5 shows a detailed split-up of the execution time of the applications in the 4 processor case. The performance results for 2 and 8 processor cases are reported in Appendix A. Our discussion here concentrates on the performance results of 4 processors, although they are also applicable to 2 and 8 processors. The total execution time of an application is broken down into the computation time (*i.e.,* time spent in the computation steps of the application — shown as "Computation" in graph legend) and the overheads time. In a separate experiment, we measured the computation time for each application running on a single processor (with $\frac{1}{4}th$ of the problem size) and found that it matches closely with computation time reported here. Hence we say that overheads considered in our study account for all time other than the computation time.

The overheads time is in turn broken down into the various overheads listed in Section 2.7. Except for the page fault kernel overhead time, all other overhead times are actually measured by running

the application on CVM. The page fault kernel overhead was measured by timing an empty SIGSEGV handler. The page fault kernel overhead per page fault was found to be 100 microseconds. Note that in our discussions, the page fault kernel overhead neither includes the actions performed to make the shared page consistent, nor the time taken to set up the appropriate read/write permissions. All timings are normalized with respect to the total execution time of the applications running under the LMW protocol in respective cases. The graphs show the timings on node 0. Timings on other nodes a follow similar trend.

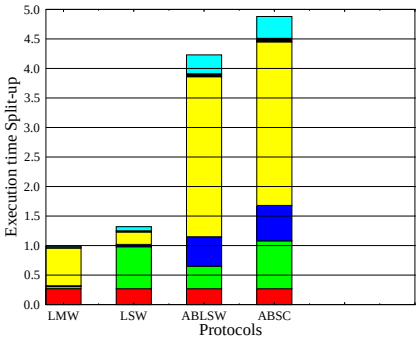
4.2.2 Results

All the applications follow the same trend as in the manual analysis, *i.e.,* ABLSW and ABSC perform significantly worse than LMW by a factor of 1.5 to as much as 90 times. Further, contrary to the popular belief about the "heavy-weightedness" of multiple writer protocols, we observe that the Twin/Diff overheads (not visible in the graphs) contribute to less than 1% of the total overheads. Decreased false sharing overheads leveraged by the negligible overheads in supporting multiple writers cause LMW to perform better than ABLSW and ABSC.

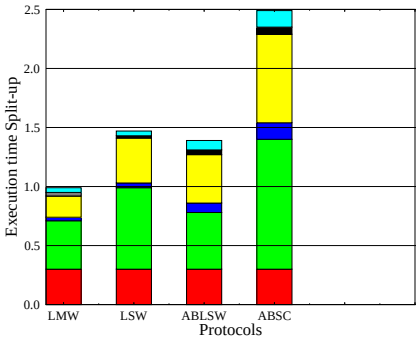
As the overheads in LSW fall closer to the best case (as seen from Tables 5, 6, and 7), we see that LSW also performs better than ABLSW and ABSC by 1.2 to 20 times. As remarked earlier, the poor performance of EmFiGs methods is due to the decrease in spatial locality. To further explain this, we first classify the page faults in the applications into two categories:

True sharing fault: The first page fault to a shared page between two synchronization points is considered a true sharing fault.

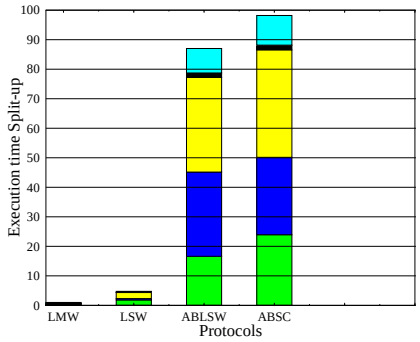
False sharing fault: All page faults other than the first page fault to a shared page between two synchronization points are considered false sharing faults. These faults occur only if some other process writes to the page in the same interval, causing the permission of the page to be changed to read-only in this process. Since the applications we have considered are free of data-races, the writes by the two



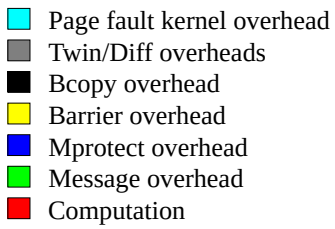
(a) BARNES



(b) WATER-SPATIAL



(c) RADIX



(d) Legend

Figure 5: Normalized Execution time split-up

processes are to non-overlapping regions. Therefore the subsequent page faults are solely due to false sharing.

We measure the true sharing and false sharing faults in the applications using the above classification criteria and report them in Table 8.

Comparing the total number of page faults in the applications from Tables 5 – 7 with the the

Benchmark	Number of False Sharing (FS) and True Sharing (TS) Faults							
	LMW		LSW		ABLSW		ABSC	
	FS	TS	FS	TS	FS	TS	FS	TS
Barnes	0	3425	3967	11426	0	55833	0	59104
Water-Spatial	0	19892	7373	17864	2280	52899	22407	60825
Radix	0	1193	5519	1197	0	28102	0	28102

Table 8: False Sharing (**FS**) and True Sharing (**TS**) Faults

number of true sharing and false sharing faults in Table 8, we see that all the page faults are true sharing faults in ABLSW and ABSC. In Water-Spatial, a few false sharing faults also occur due to the read-write false sharing of the individual molecules themselves. Thus we observe that the reduced exploitation of spatial locality causes high true sharing faults in ABLSW and ABSC. This increase in the number of true sharing faults necessitates more page request messages. Therefore we observe higher message overheads in ABLSW and ABSC.

LSW and SC protocols require that a page that is written by a process be made read-only at that process and invalidated at other processes at the next synchronization point. In ABLSW and ABSC, due to bloating of shared arrays, more pages have to be made read-only or invalidated at synchronization points than LMW or LSW. This leads to more work during a barrier synchronization in ABLSW, which in turn results in a larger barrier overheads. In the case of ABSC, the action performed at a barrier synchronization is minimal. However, it happens that one of the processes (typically process 0) arrives at a barrier earlier. This is because process 0 has read/write permission for all pages that it is modifying and read permission for all other pages. Hence this process completes all its computation until the next synchronization point. During this period, the process does not see asynchronous messages sent by other processes. This happens because in CVM asynchronous message are received using polling, and polling occurs only when the application process invokes a DSM layer routine. This is another example of implementation dependent overhead. We discuss this further in Section 4.4.

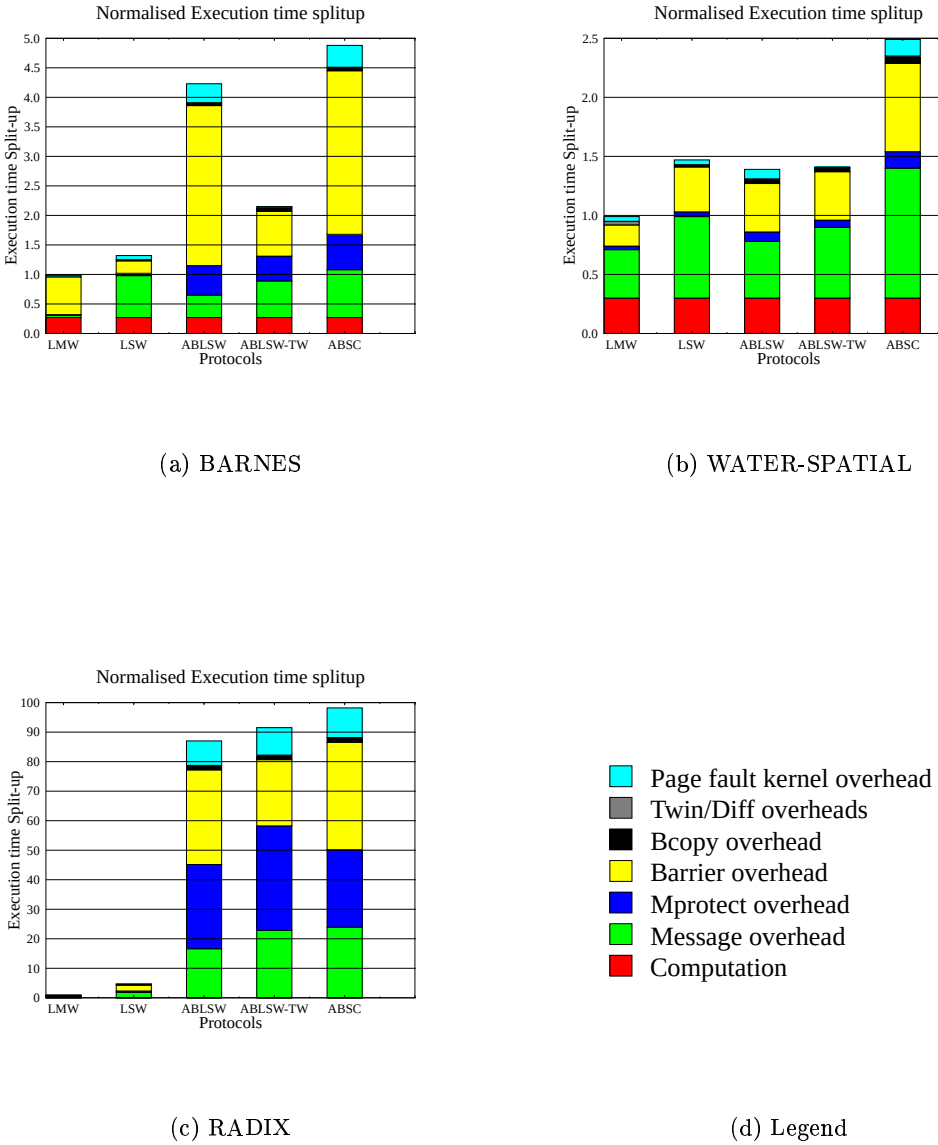
As a consequence of the above, process 0 enters the DSM layer only upon executing the next synchronization call. Subsequently all page/diff requests of other processes were satisfied by process 0, and the other processes arrive at the barrier. In our performance measurements, the barrier time also includes the waiting time at the barrier. Hence the barrier overhead is larger even in ABSC. Thus we see that the decreased exploitation of spatial locality is the main reason for poor performance of ABLSW and ABSC, the dominating overheads being message and barrier overheads. We address the spatial locality effects in the following section.

4.3 Overcoming Spatial Locality Effects using Tapeworm

In the previous section, we have seen that the principal reason for the poor performance of ABLSW and ABSC is the high true sharing faults which is the result of reduced exploitation of spatial locality in these protocols. This raises the question that if the effects of reduced spatial locality are eliminated, will ABLSW and ABSC perform better than LSW or LMW? In other words, can the EmFiGS approach be somehow made useful? In this section, we propose to incorporate Tapeworm [17], a record-replay scheme which helps overcome spatial locality effects, in ABLSW. We refer to this combined scheme as ABLSW-TW, and evaluate its performance in this section.

The basic idea behind Tapeworm is to "record" accesses to a shared page in an interval between two synchronization points (not necessarily two consecutive ones). The recorded accesses can be "replayed" at the beginning of the next instance of the same interval, say the next iteration. Replaying means validating those pages early. The pages can be validated in an aggregated fashion speculatively. The read accesses to the *shared* array by the processes will not cause page faults in the current iteration, if the accesses to shared page follow the recorded information. Thus, iterative applications that tend to access same set of shared pages across iterations benefit from the tapeworm mechanism by incurring fewer page faults and messages. However, if the accesses to shared memory are irregular and non-repetitive, then these applications do not benefit from tapeworm and might even perform poorly because of the

wasted effort in fetching unnecessary pages.



(a) BARNES

(b) WATER-SPATIAL

(c) RADIX

(d) Legend

Figure 6: Normalized Execution time split-up

Figure 6 shows the normalized execution time of applications under ABLSW-TW protocol. The normalized execution time of applications under other protocols are also shown for comparison. The execution times are as observed on node 0 when the application were run on 4 processors. The execution time results for 2 and 8 processor cases are shown in Appendix A.

Comparing the execution times of ABLSW and ABLSW-TW, we observe that the execution time reduces by a factor of 2 in Barnes. Note that while the message time has increased slightly, the barrier and page fault kernel overhead times have decreased significantly in this application, causing ABLSW-TW to perform better than ABLSW. However, in the remaining two applications, the execution time under ABLSW-TW is marginally higher than ABLSW. In Water-Spatial, even though the page faults are completely removed, and the page fault kernel overhead is nearly zero, due to the increased message overhead, ABLSW-TW performs worse than ABLSW by about 2%. In Radix, Tapeworm is not very beneficial as (i) the number of iterations is only 4, (ii) the recorded information can only be used in the last two iterations, and (iii) the record/replay scheme is not accurate for RADIX, as the writes were to different sets of pages in each iteration. As a consequence, the reduction in the number of page faults and messages in Radix under ABLSW-TW is not as significant as in Barnes. We also observe an increased message overhead in Radix under ABLSW-TW, which contributes to its poor performance.

Next, we observe that the barrier synchronization overhead is lower in ABLSW-TW than in ABLSW. This is because, the actions performed in a barrier vary between the protocols (although the number of barriers executed under different protocols is the same). Also, we observe that the message time under ABLSW-TW has increased by 15 – 30% compared to ABLSW in all three application. This is partly because of the increased message sizes in ABLSW-TW due to aggregation. Further the increase is also due an the implementation specific detail, namely, asynchronous message handling in CVM. We analyze the message overhead in greater detail in the following section.

Last, comparing ABLSW-TW with LMW, we observe that the execution under LMW is significantly lower, by a factor of 1.4 to 90, in all three applications. Thus, even after removing all the true sharing fault overheads caused by the reduced exploitation of spatial locality in array bloating or the EmFiGS approach, ABLSW-TW performs poorly due to the increased message overhead, increased barrier wait time, and the large `mprotect` overheads due to array bloating. Thus LMW seems to be a clear winner

and tolerates false sharing effects.

4.4 Analysis of Message Overhead

In CVM, a process sends messages to other processes on various events like page faults, lock acquire, or barrier. These messages are handled asynchronously, *i.e.,* a message is processed only when the receiving process polls for the message, even though the message might have been received by the underlying messaging layer at an earlier point of time. This is because the receiving process is typically unaware of the message arrival. In CVM, polling for messages is performed using the `MPI_probe` function at the beginning of the page fault handler or in any explicit function calls to *cvm_barrier*, *cvm_lock*, or *cvm_unlock*. As a result of this asynchronous message handling, some time is elapsed between the instant the message arrives at a node and the instant the receiving process actually "sees" it. We refer to this time as "holdup" time (Keleher [24] uses the term "responsiveness").

We measure the holdup time for a request message by taking the difference between the time when the message is sent from the sender and the time when the receiver "sees" the message[4]. This difference actually gives the sum of latency of the message and the holdup time. In CVM, request messages are small as they contain only a flag indicating the type of request, *e.g.,* page request and diff request. Therefore, we approximate the above time to the holdup time of the message. Table 9 shows the average holdup time per message in the three applications under the different protocols.

Benchmarks	Average Holdup time (in μ seconds)				
	LMW	LSW	ABLSW	ABLSW-TW	ABSC
Barnes	156	691	232	7214	153
Water-Spatial	1196	870	449	2701	288
Radix	526	375	3482	4115	2203

Table 9: Average Holdup Time

We see that the average holdup time for ABLSW-TW is highest in all cases. In fact, the average

[4]We use IBM's proprietary implementation of the MPI standard in which `MPI_Wtime` function returns the global time.

holdup time in ABLSW-TW is 2 – 8 times larger than the average holdup time in LMW. In ABLSW-TW, since pages are fetched ahead of demand, the number of page faults incurred is significantly lower. Hence, the number of calls to `MPI_probe` also becomes lower. Therefore, we see a high holdup time in ABLSW-TW. Consequently, the message overhead in ABLSW-TW is higher and ABLSW-TW performs poorer than LMW and LSW. Further, we observe that the average holdup time among other protocols do not follow a consistent trend. This may be due to the fact that the holdup time is strongly influenced by the frequency of `MPI_probe` calls, which, in turn, depends on the application behavior like how often synchronization primitives (`cvm_barrier`, `cvm_lock`, and `cvm_unlock`) are called. These are the occasions when DSM layer gets control and the `MPI_Iprobe` is issued. As a consequence, we see a somewhat inconsistent trend in the holdup times under LMW, LSW, ABLSW, and ABSC.

Thus, the decreased number of page faults in ABLSW-TW causes an indirect effect of increased holdup time and consequently higher message overheads. This effect is mainly due to the implementation of message handling in CVM, *i.e.,* asynchronous message handling. This raises the question "Would ABLSW-TW perform better with application-level polling [24] or interrupt-based message handling?". To address this question, we employed application level polling in the benchmarks. We inserted the `cvm_probe` call, which checks for new messages and processes the outstanding messages if any, in the innermost loops of the benchmarks. We report the execution times of the benchmarks in the 4 processes case with and without `cvm_probe` in Table 10.

	Time in seconds									
	LMW		LSW		ABLSW		ABSC		ABLSW-TW	
Benchmark	Async. Message	Appln. Polling	Async. Message	Appln. Polling	Async. Message	Appln. Polling	Async. Message	Appln. Polling	Async. Message	Appln. Polling
Barnes	32.83	31.72	43.63	43.54	141.08	140.55	162.32	164.11	71.74	71.12
Water-Spatial	59.67	51.07	88.19	92.80	85.04	78.02	151.01	147.21	84.25	72.94
Radix	4.16	3.03	21.11	21.32	364.03	328.74	410.54	343.47	358.21	346.10

Table 10: Execution Times with Application Level Polling

We observe that even though application level polling improves the performance by upto 20% in

some cases, we do not see any change in the general trend with respect to the relative performance of the different methods in any of the applications. Therefore, we conclude that the experimental results presented in this section, especially with respect to the relative performance of the different methods, will hold good even when application level polling is used.

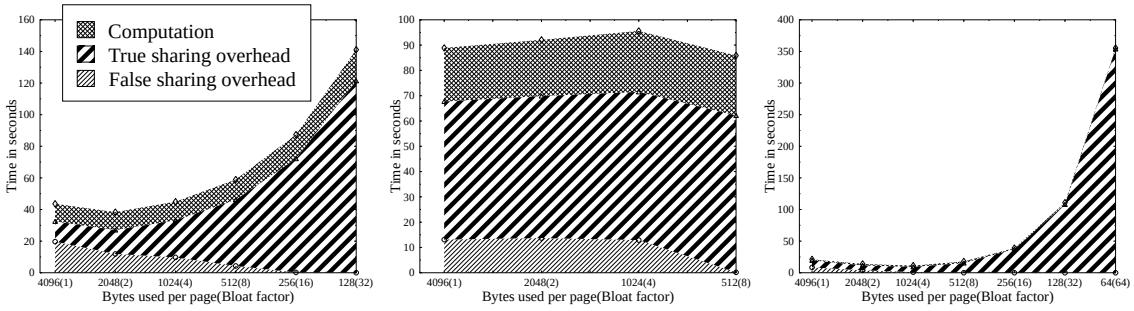
4.5 Effects of Bloat Factor

In our discussion so far on the array bloating implementation, we have assumed the allocation of only one body, molecule or key per shared page, *i.e.,* we have considered *bloat factors* of 32, 8 and 64 in BARNES, WATER-SPATIAL and RADIX respectively. From the discussions in Section 3.4, it is evident that the above *bloat factors* can eliminate false sharing completely. However, at the same time, they also reduce the exploitation of spatial locality in the applications. Thus the *bloat factor* parameter has a two fold effect: increasing it reduces false sharing but decreases exploitation of spatial locality, whereas decreasing it introduces more false sharing but increases exploitation of spatial locality. Hence, a natural question that arises is where is the tradeoff point for these two effects. To investigate this, we present the execution times of the three applications running under ABLSW with different *bloat factors.*

The execution times of the applications are broken down into false sharing overheads, true sharing overheads, and the computation time. All the overheads due to false sharing faults[5] (including the associated messages, operations in SIGSEGV handlers, and `mprotect`s) are included in false sharing overheads. The overheads due to true sharing faults are included in true sharing overheads. We include the only other overhead, *viz.,* the barrier overhead, also in true sharing overheads. This is because the amount of work done at barrier (like invalidation of pages) is directly proportional to the number of shared pages. The computation time is the time spent in computation steps in the application.

Figure 7 presents the break-up of the execution time into false sharing overhead, true sharing

[5]Refer to Section 4.2.2.



BARNES WATER-SPATIAL RADIX

Figure 7: Effect of Bloat Factor

overhead, and computation time for the three applications running under ABLSW protocol with varying *bloat factors*. On the x-axis, we plot the actual number of bytes that are used per shared page. This is equivalent to the size of a millipage in Millipage/Multiview. The corresponding *bloat factors* are also shown in brackets. Note that, higher the value of bloat factor, fewer bodies, molecules, or keys are allocated per page. Thus, the left most point on the x-axis corresponds to a bloat factor of 1, which represents pure LSW protocol and the right most point represents the full ABLSW protocol as discussed in Section 3.2. The y-axis represents of execution time in seconds. The graph shows 3 regions representing false sharing overheads, true sharing overheads, and the computation time.

In all cases the false sharing overhead decreases with increasing *bloat factors*. When the *bloat factor* is such that there is only one body, molecule, or key per shared page, there is no false sharing in the application and we see that false sharing overhead is 0. The true sharing overhead increases for all applications with increasing *bloat factor*. A break-even between these two overheads occur at a bloat factor of 2 (2048 bytes per page) in the case of BARNES and at a bloat factor of 4 (1024 bytes per page) in the case of RADIX. As can be observed at least in BARNES and RADIX, the increase in true sharing significantly dominates over the decrease in false sharing. Hence the total execution time is seen to be increasing at higher bloat factors. In WATER-SPATIAL too, the false sharing overhead decreases and the true sharing overhead increases with increasing *bloat factors*. the break-even occurs

at a bloat factor of 8 (512 bytes per page) Because of the less steep increase in the true sharing overhead (with increasing bloat factors), the effects of increased true sharing overhead is not as evident as it is in the other two applications. Nonetheless, the true sharing overhead does increase with bloat factor; its contribution to the execution time at bloat factors 1, 2, 4 and 8 are 54.5, 56.1, 58.6 and 62.0 seconds respectively.

Thus we see that the inter-play between the false sharing and true sharing overheads produces best performance at a lesser value of bloat factor, which in general is not equal to the value where we have only one body, molecule, or key per page. We call the ABLSW with this bloat factor as best case ABLSW. Table 11 shows the execution time of best case ABLSW along with LMW. We see that even with the best case bloat factor, ABLSW performs worse than LMW by an a factor 1.15 to 4.06. Last, we see that having a value of *bloat factor* lesser than the maximum value required to eliminate false sharing completely is beneficial in ABLSW even at the cost of some false sharing overheads.

	Exec. time in seconds	
Benchmark	LMW	Best Case ABLSW
BARNES	32.83	38.34
WATER-SPATIAL	59.67	85.87
RADIX	4.16	11.22

Table 11: Comparison of best case ABLSW vs. LMW

5 Related Work

There is a large body of literature in the area of software distributed shared memory [4, 19, 21]. A specific concern in page-based software DSM systems is the effect of false sharing on the performance. The LMW approach which uses the relaxed memory consistency model [1, 18] with multiple writers [6, 15] has been widely used to overcome the effects of false sharing. Recently, Itzkovitz and Schuster in their work on Millipage/Multiview [12, 11] present a different technique to eliminate false sharing in page-based software DSMs by emulating fine grain sharing. Their work only reports performance

speed-ups obtained from their technique, but does not present a quantitative comparative analysis of the multiple writer approach and Millipage. The work presented in this paper addresses this problem both by an implementation independent analysis and thorough experimental evaluation.

The successor to Millipage [13, 23] extends the technique to adapt the sharing granularity across variables and code sections dynamically. This is a much recent work and is not addressed elaborately in our work. Nonetheless, our manual analysis can be extended directly to adaptive Millipage by considering different sharing granularities across different iterations and different sections of the iteration. Also, the maximum speedup improvement that is reported in this work [23] is 1.8 for the adaptive Millipage over original Millipage. But we observe that the speedup improvements of LMW over ABSC (following the EmFiGS approach, representative of Millipage) are 4.9, 2.5, and 99 for the three applications. Further, the Tapeworm technique more or less captures the adaptiveness by validating multiple pages dynamically. However, our results establish that even ABLSW-TW does not perform better than LMW.

Orthogonal approaches based on compiler techniques have been presented by Scales *et al.,* [25], Jeremiassen [14], and Granston [9] to overcome false sharing effects in software DSM system. While the later two works [9, 14] are specifically for fine grain hardware DSM, the first work proposes a software DSM that supports fine grain sharing. In [7], the performance of Shasta [25], a fine grain software DSM system, and Cashmere [26], a software coherent shared memory on a cluster. Although this work presents quantitative performance results, it does not focus on false sharing. Our paper, on the other hand, compares false sharing removal and false sharing tolerance on page-based DSM systems with no additional hardware support.

Our Array Bloating is somewhat similar to the *pad and align* transform approach followed in [14]. Freeh and Andrews [8] have proposed the "Writer-owns" protocol which improves upon the multiple writer protocol by dynamically re-mapping sub-pages (called "atoms") to the processes that write to

them at run time, to reduce false sharing. While the focus of these works are on developing efficient DSM architecture, our work concentrates on comparative performance evaluation of the different approaches. To the best of our knowledge, there is no such comparative study reported in the literature.

In the work [3] by Amza *et al.,* it is claimed that the performance of many applications running under relaxed consistency models improve when sharing granularity is increased. This work also discusses tradeoffs between increased false sharing and better exploitation of spatial locality in LMW systems, when bigger consistency units (sizes bigger than a page) are used in software DSMs. They attribute the performance improvements to the fact that the benefits of aggregation usually outweigh false sharing penalty. This is in agreement with our results. Further our work adds more insight to this observation by comparing LMW with systems having lower granularity (sizes smaller than a page), but running SC and LSW protocols. Last, while their performance results are based on experimental evaluation on Treadmarks software DSM [19] under LMW protocol, our work presents both analytical and experimental results for LMW, LSW, ABLSW, and ABSC

Zhou *et al.,* [30] present the performance tradeoffs of relaxed consistency models and true fine grain sharing. They claim that the performance of hardware fine grain sharing systems with sequential consistency tends towards the performance of relaxed consistency models. But in our work, we have considered emulated fine grain sharing without any architectural support.

6 Conclusions

In this paper, we have presented the performance evaluation of two approaches to overcome false sharing, *viz.,* the Multiple Writer approach and the EmFiGS approach, in page-based software DSM systems. Our evaluation is two pronged : (i) an implementation independent approach which reports overheads counts in different methods and (ii) an experimental evaluation of the methods implemented on top of CVM, a page-based software DSM running on IBM-SP2. Our results indicate that even

though false sharing is eliminated completely in the EmFiGS approach, its overheads are found to be significantly higher by a factor of 1.5 to 90 times the overheads in multiple writer approach. The high overheads in the EmFiGS approach are due to the decreased exploitation of spatial locality. By emulating finer granularity of sharing, the EmFiGS approach incurs more true sharing overheads than the savings in false sharing. In contrast, the overheads incurred by the multiple writer approach to support multiple writers is insignificant, which is contrary to the popular belief. Further, the EmFiGS approach performs worse than LSW (Lazy release Single Writer) protocol, which experiences the false sharing overheads in full. The performance of the EmFiGS method, namely ABLSW doesn't improve significantly even after incorporating Tapeworm, which is a record-replay technique to fetch pages ahead of demand in an aggregated fashion. Thus, ABLSW-TW, ABLSW with Tapeworm, performs worse than LMW. Next, we have analyzed the effects of asynchronous message handling on the performance of the methods. Last, we have also investigated the effect of sharing granularity in the EmFiGS approach. The tradeoffs between higher false sharing at higher sharing granularity and reduced exploitation of spatial locality at lower sharing granularity are reported.

The overhead analysis presented in this paper is restricted to software DSMs with no hardware support and no OS modification. It can be extended to software DSMs that include hardware support and/or OS modification. We have reported experimental evaluation of various overheads in a public domain software (CVM) on IBM's Scalable Parallel (SP) architecture with a high performance switch. It is possible to study the effects of various overheads under different messaging layers and newer network interface architecture as well as a different mechanism, such as interrupt based scheme, for handling asynchronous messages. Last, as discussed in Section 5 our manual analysis can be extended directly to adaptive granularity schemes by considering different sharing granularities across different iterations. These are possible future directions.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs between false sharing and aggregation in software distributed shared memory. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 90–99, June 1997.

[4] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors. Technical Report COMP TR89-91, Dept. of Computer Science, Rice University, April 1989.

[5] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, , and Marco Zagha. A comparison of sorting algorithm for the Connection Machine CM-2. In *Proc. of the 3rd Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA'91)*, pages 3–16, July 1991.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

[7] S. Dwarkadas, H. Lu, A. L. Cox, R. Rajamony, and W. Zwaenepoel. Combining compile-time and run-time support for efficient software distributed shared memory. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):476–486, March 1999.

[8] V. W. Freeh and G. R. Andrews. Dynamically controlling false sharing in distributed shared memory. In *Proc. of the Fifth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-5)*, pages 403–411, August 1996.

[9] E. D. Granston. Towards a compile-time methodology for reducing false sharing and communication traffic in shared virtual memory systems. In *Proc. of the 6th Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 273–289, June 1994.

[10] Chris Holt and Jaswinder Pal Singh. Hierarchical n-body methods on shared address space multiprocessors. In *7th SIAM International Conference on Parallel Processing for Scientific Computing*, pages 313–318, February 1995.

[11] A. Itzkovitz and A. Schuster. Distributed shared memory: Bridging the granularity gap. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, June 1999.

[12] A. Itzkovitz and A. Schuster. MultiView and Millipage: Fine-grain sharing in page-based DSMs. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation (OSDI'99)*, pages 215–228, February 1999.

[13] Ayal Itzkovitz, Nitzan Niv, and A. Schuster. Dynamic adaptation of sharing granularity in DSM systems. In *Proc. of the 1999 Int'l Conf. on Parallel Processing (ICPP'99)*, September 1999.

[14] T.E. Jeremiassen and S.J. Eggers. Reducing false sharing on shared memory multiprocessors through compile-time data transformations. In *Symposium on Principles and Practice of Parallel Programming*, pages 179–188, July 1995.

[15] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Department of Computer Science, Rice University, December 1994.

[16] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems (ICDCS-16)*, pages 91–98, May 1996.

[17] P. Keleher. Tapeworm: High-level abstractions of shared accesses. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation (OSDI'99)*, pages 201–214, February 1999.

[18] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[19] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[20] Pete Keleher. The CVM manual. Technical report, University of Maryland, November 1996.

[21] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC'86)*, pages 229–239, August 1986.

[22] K. V. Manjunath and R. Govindarajan. Performance Analysis of Methods that Overcome False Sharing Effects in Software DSMs. 2001.

[23] Nitzan Niv and A. Schuster. Transparent adaptation of sharing granularity in multiview-based software DSM systems. In *15th International Parallel and Distributed Processing Symposium*, April 2001.

[24] D. Perkovic and P. Keleher. Responsiveness without interrupts. In *Proc. of the 13th ACM-SIGARCH Int'l Conf. on Supercomputing (ICS'99)*, June 1999.

[25] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 174–185, October 1996.

[26] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and Michael Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

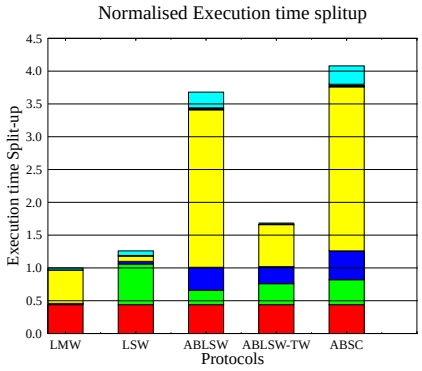
[27] M. Swanson, L. Stroller, and J. B. Carter. Making distributed shared memory simple, yet efficient. In *Proc. of the 3rd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'98)*, pages 2–13, March 1998.

[28] J. Torellas, M.S. Lam, and J.L. Henessey. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 6(43):651–663, June 1994.

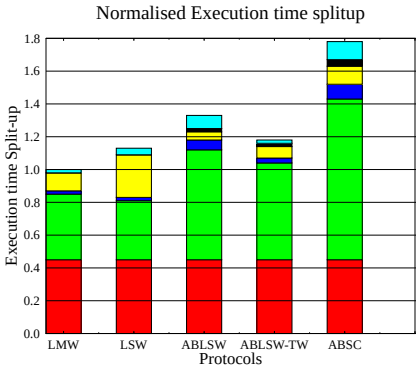
[29] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological consideration. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

[30] Y. Zhou, L. Iftode, K. Li, J. P. Singh, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed consistency and coherence granularity in dsm systems: A performance evaluation. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 193–205, June 1997.

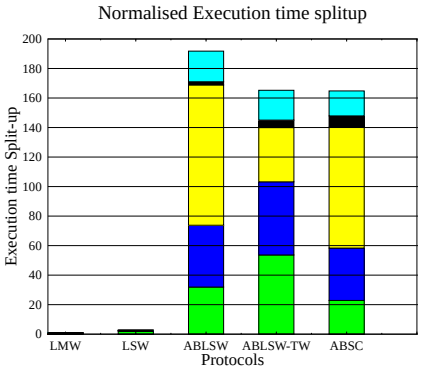
A Normalized execution times



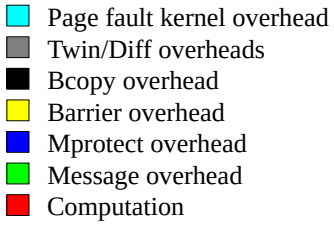
(a) BARNES



(b) WATER-SPATIAL

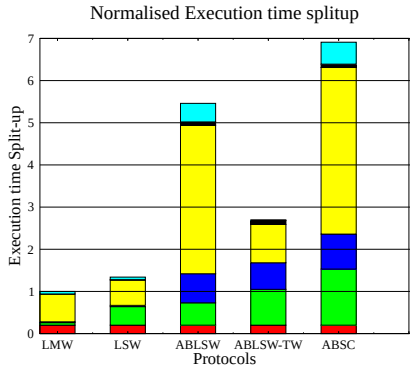


(c) RADIX

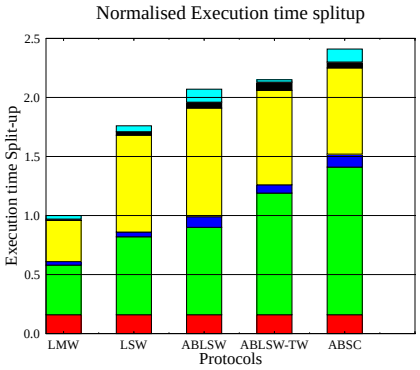


(d) Legend

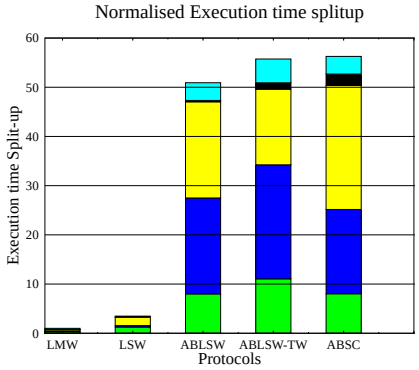
Figure 8: Normalized Execution time split-up — 2 processors case



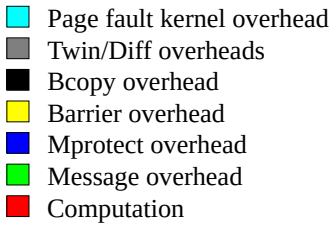
(a) BARNES



(b) WATER-SPATIAL



(c) RADIX



(d) Legend

Figure 9: Normalized Execution time split-up — 8 processors case