International Journal of Parallel Programming, Vol. 32, No. 2, April 2004 (© 2004)

CAS-DSM: A Compiler Assisted Software Distributed Shared Memory

N. P. Manoj,[1] K. V. Manjunath,[2] and R. Govindarajan[3]

Traditional software Distributed Shared Memory (DSM) systems rely on the virtual memory management mechanisms to detect accesses to shared memory locations and maintain their consistency. The resulting involvement of the OS (kernel) and the associated overhead which is significant, can be avoided by careful compile time analysis and code instrumentation. In this paper, we propose such a Compiler Assisted Software support approach (CAS-DSM). In the CAS-DSM implementation, the involvement of the OS kernel is avoided by instrumenting the application code at the source level. The overhead caused by the execution of the instrumented code is reduced through several aggressive compile time optimizations. Finally, we also address the issue of reducing certain overheads in polling-based implementation of receiving asynchronous messages. We used SUIF, a public domain compiler tool, to implement compile time analysis, instrumentation and optimizations. We modified CVM, a publicly available software DSM to support the instrumentation inserted by the compiler. Detailed performance evaluation of CAS-DSM is reported using a set of Splash/Splash2 parallel application benchmarks on a distributed memory IBM SP-2 machine. CAS-DSM achieved moderate to good performance improvements for most of the applications compared to the original CVM implementation.

[1] Hewlett-Packard India Software Operations, 29 Cunningham Road, Bangalore 560 052, India. E-mail: manojnp.mnp@alumnus.csa.iisc.ernet.in

[2] Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109, USA. E-mail: kvman@umich.edu

[3] Department of Computer Science and Automation, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560 012, India. E-mail: govind@csa.iisc.ernet.in

0885-7458/04/0400-0077/0 © 2004 Plenum Publishing Corporation

Reducing the overheads in polling-based implementation improves the performance of CAS-DSM significantly resulting in an overall improvement of 12–52% over the original CVM implementation.

KEY WORDS: Coherent Virtual Machine (CVM); software distributed shared memory; Stanford University Intermediate Form (SUIF); performance evaluation

1. INTRODUCTION

Shared Memory and the *Message Passing* models are two major parallel architecture models. A Shared Memory system has a global memory accessible to all the processors in the system.[1] There are two models available in shared memory systems based on the nature of sharing of this global memory across processors. They are the Uniform-Memory-Access (UMA) architecture, wherein the access times to a memory word from any two processors are equal and the Non-Uniform-Memory-Access (NUMA) architecture, wherein the access time varies for different processors.[1] Examples of NUMA architectures, also known as hardware Distributed Shared Memory (DSM) systems include Stanford DASH,[2] SCI,[3] DDM,[4] and KSR1.[5] Programming in shared memory systems is relatively simpler as updating shared data is considered as a natural extension to programming in uniprocessor environment. This is a major advantage of shared memory systems. However, with increase in the number of processors, these systems typically suffer from increased contention and longer latencies in accessing the shared memory. This degrades the performance and limits scalability. Also, data access synchronization, cache coherency, and memory consistency are other issues in shared memory systems. Although sophisticated memory consistency models[1, 6] facilitate higher performance, they also increase the burden on programmers to ensure appropriate memory access order through synchronization primitives. Lastly, implementing the shared memory abstraction in hardware increases the cost of the system.

A Distributed Memory system consists of multiple independent processing nodes with local memory modules, connected by a general interconnection network.[1] Unlike shared memory systems, the local memories are private and are accessible only to the local processors. Hence these systems also have the nomenclature No-Remote-Memory-Access (NoRMA) machines. Since there is no centralized shared memory, increasing the number of nodes does not result in any of the problems mentioned for shared memory systems. Thus, the main advantage of these systems is that they are highly scalable and do not require expensive and complex hardware for memory controllers. A major drawback of distributed memory

systems is that the responsibility of partitioning the data and managing the communication falls on the application programmer. Parallel Virtual Machine (PVM)[7] and Message Passing Interface (MPI)[8] are standard application program interfaces used to program distributed memory systems.

The Distributed Shared Memory (DSM) paradigm evolved from the shared memory model and the distributed memory model, attempts to combine the advantages of shared memory and distributed memory systems. In other words, DSM systems attempt to achieve both ease of programming and scalability. A distributed shared memory system (DSM) logically provides a single global address space over a physically distributed system. A DSM system can also be realized in software. In a software DSM, a software layer provides the shared memory abstraction over a distributed memory machine. This abstraction in software can be provided in several ways such as by user-level libraries, modification of OS, using compiler support, binary program instrumentation, or by a combination of these. Ivy,[9] TreadMarks,[10] Munin,[11] Midway,[12, 13] CVM,[14] Shasta,[15] Tapeworm,[16] and Millipede[17] are examples of software DSMs.

Software DSM systems are attractive from the viewpoint that they are cost effective and can run either on a distributed memory machine or on a cluster of workstations. Further, software DSM systems can employ more sophisticated memory consistency models to achieve higher performance. Although the performance of software DSMs is somewhat lower compared to hardware DSMs, Lu, *et al.*, demonstrate that software DSM programs could achieve a performance comparable to efficient distributed memory programs.[18] Thus software DSMs not only serve as a platform for shared memory program development but also as systems where moderate to high performance can be achieved in a distributed memory machine or a cluster of workstations/PCs at moderate to low cost. Recently OpenMP[19] is gaining increasing acceptance as a shared memory programming model. In order to run OpenMP programs on distributed memory machines or cluster of workstations, a software DSM is used as an inexpensive base framework in Refs. 20 and 21.

1.1. Overview of Our Work

Typically, a user-level DSM detects accesses to the shared location through the support of the virtual memory (VM) system. Such a software DSM is also known as Distributed Virtual Shared Memory System (DVSM). If a shared location is not in a consistent state as per the followed memory consistency model, then an access to the page in which this location resides will result in raising the segmentation violation (`segv`) signal

which will be caught by the DSM layer using a segmentation violation handler. This handler will make the page consistent by taking appropriate actions. The steps are detailed in Section 2.1.

One of the main overheads occurring in a DVSM is due to the reliance on VM mechanism for identifying shared memory location. The motivation for our approach comes from the fact that while the consistency steps themselves are unavoidable, the generation of the `segv` signal and the resulting involvement of the OS for handling the `segv` signal, can be avoided by careful compile time code instrumentation. We discuss methods by which this dependence on the VM mechanism can be reduced significantly or eliminated by performing compile time analysis of the application source code. In this paper, we propose an implementation of a Compiler Assisted Software DSM (CAS-DSM). A number of compile-time optimizations have been proposed and implemented to reduce the overheads incurred by the instrumented code, and thus improve the performance of CAS-DSM. These optimizations include, aggregation and hoisting (above nested `for` loops) of the instrumented code for ensuring consistency, selective discarding of consistency checks, and function-inlining and constant propagation to increase the scope of our compile-time analysis and optimization. We evaluate the performance improvement due to these optimizations on a set of Splash/Splash-2 shared memory applications.[(22)] CAS-DSM achieves 5–15% performance improvement over Coherent Virtual Machine (CVM),[(14)] a public domain software DSM on which CAS-DSM is implemented.

Software DSMs typically use the polling-based approach to receive asynchronous messages. This is because the polling-based approach incurs less overhead than the interrupt-driven approach. However, the time at which the messages are polled relative to when they arrive, informally referred to as the *holdup time* time,[4] can have significant impact in software DSMs. In this paper, we propose to reduce the holdup time by the use of compile-time instrumentation in source code and present a simple approach for this. Preliminary performance results of our approach reveals a significant reduction in the holdup time which in turn leads to overall performance improvement of 10–15% on the average, and upto 52% in certain application.

Section 2 discusses the motivation behind our work. In Section 3, we present the details of a basic implementation of CAS-DSM and its performance. Sections 4 and 5 discuss the proposed compile-time optimizations and their implementation respectively. We report the performance of

[4] Refer to Section 2.3 for a detailed discussion and a clear definition of holdup time.

CAS-DSM and the benefits of different optimizations in Section 6. Section 7 discusses works related to ours. We conclude in Section 8.

2. MOTIVATION

This section is divided into three parts. The following subsection provides the necessary background for understanding our work. Next, we discuss the pagefault overhead and motivate the need for reducing this. Last, we illustrate certain overheads involved in the polling-based approach for receiving asynchronous messages and the need to reduce them.

2.1. Background

Our work is in the area of DVSMs. Memory consistency models play a very important role in DSMs. A memory consistency model defines the legal ordering of memory references issued by a processor as observed by other processors.[23] A commonly assumed memory consistency model for shared memory multiprocessors is *sequential consistency*, which gives programmers a simple view of the system. Informally, sequential consistency requires that memory operations from all processors appear to execute one at a time and interleaved in an arbitrary manner, with the memory operations within a process maintaining program order. The sequential ordering of reads and writes, however, limits the performance in a multiprocessor system. To overcome this, several relaxed memory consistency models were proposed, such as *processor consistency*,[24] *weak consistency*,[25] and *release consistency*.[23] In the (eager) release consistency model,[23] a consistent view of memory is guaranteed only at (lock) release or barrier synchronization points. In our work, we follow the lazy release consistency protocol,[26] in which the consistent view is further delayed to the subsequent lock acquire, and the consistent view is guaranteed only at the acquiring process. The relaxed consistency models allow more asynchrony to be exploited, resulting in lower execution time for the shared memory application. However, programming under weaker consistency models requires additional efforts from the programmer, as it is the responsibility of the programmer to ensure the required memory access ordering through appropriate use of synchronization primitives.

As mentioned earlier, many of the DVSMs detect accesses to shared locations with the help of the VM support through `SIGSEGV` signal. On a `SIGSEGV` signal, with the involvement of the OS kernel, the segv handler installed in the DSM layer gains control over the execution. First, this segv handler checks that the page fault was caused due to an access to a shared location and not by a program error. Once it confirms that the signal was

indeed caused by a shared memory access, steps are initiated to make the shared data consistent, consistent under the supported memory consistency model.

Typically in a software DSM, the size of the shared data accessed/fetched from a remote node, referred to as the granularity of sharing, is equal to the size of a virtual page. This is mainly due to involvement of the virtual memory management in maintaining consistency of shared data, and partly due to the overhead involved in sending short messages in distributed memory systems. A side effect of the large granularity of sharing is *false sharing*. False sharing occurs when two processors repeatedly access, at least one of the accesses being a write, to different parts of a shared page. This results in the page being sent back and forth between the two processors. If the accesses by the two processes are to non-overlapping parts of the page, the incurred cost of sending pages back-and-forth is an overhead, as the data is not truly shared. In order to reduce the overheads due to false sharing we follow a multiple writer protocol.[11] A multiple writer protocol will allow more than one processor to write into the same page and thus reduce false sharing. As per the lazy release multiple writer protocol, the modifications to the pages have to be merged at a subsequent synchronization point to obtain a consistent copy of the page. In order to achieve this, a copy of the page, called *twin*, will be made when the page is first written. At a synchronization point, the differences between the original copy and the written copy is determined. This is called a *diff*. The diffs by all the processors are collected at the synchronization point for making the page consistent.

In our work, we assume that the shared memory programs are written for a system supporting lazy release consistency model. Further the shared programs are written under the SPMD (Single Program Multiple Data) program model, and all shared variables have an explicit "shared" keyword as prefix.

2.2. Pagefault Overhead Reduction

As mentioned earlier, many of the software DSMs detect accesses to shared locations and maintain consistency through `SIGSEGV` signal. While the steps taken by the `segv` handler themselves are unavoidable (in the sense that they are needed for ensuring the consistency of data), the involvement of the OS (kernel) and hence the associated overhead due to the page fault is high and avoidable. Henceforth we refer to this overhead as the *page fault overhead*. In this overhead we do not include the time taken for the steps that make a shared page consistent. Instead of invoking the consistency steps implicitly through the segv handler, one could insert

Table I. Page Fault and Function Invocation Overheads

Platform	Page fault overhead	Empty function call overhead
	in μ seconds	
IBM SP Power2 Processor @ 77MHz	104.1	0.89
IBM Power3 Processor @ 375MHz	22.5	0.02
Sun UltraSparc II Processor @296 MHz	78.9	0.01
Sun UltraSparc II Processor @450 MHz	48.9	0.07

explicit function calls in the application program. The latter approach would only incur the overhead of invoking a function call. We compared the overheads due to page fault and function invocation on a number of hardware platforms.[5] Our results are summarized in Table I.

Thus the page fault overhead due to the kernel involvement is at least two orders of magnitude higher. This overhead can be avoided by careful compile time analysis and code instrumentation. We explain this with the help of an example.

Consider the example code given in Fig. 1. All the variables prefixed by "`shared_`" are shared variables. The function call `barrier()` represents the call to the barrier synchronization routine. Assume at the start of the code all shared pages are invalid at a particular node. For the code shown in Fig. 1, a traditional DSM will cause a page fault in statement 3, when each page of shared memory is accessed for the first time. The subsequent access in statement 4 will not cause a page fault. However the write access in statement 5, will again raise a SEGV to ensure the twin of the page is created when an element in the shared page is written for the first time. Finally, the subsequent read access do not cause any page fault.

Our compiler support essentially captures this idea by inserting appropriate code before statements 3 and 5 and, thereby avoiding the possible page fault at these accesses. The inserted code is henceforth referred to as *instrumented code*. The inserted API function `make_page_readable()` ensures that the page in which this shared address lies is made consistent and read permission is set through the `mprotect()` call. Similarly, the API function `make_page_writable()` ensures a consistent page is both readable and writable, and the creation of the twin for the

[5] Since the experimental results presented in this paper is based on a somewhat older hardware platform, we conducted this study on a number of modern workstations to see if there is a significant difference between these two overheads in modern platforms as well.

```
(1) barrier();
(2) for (i = 0; i < N; ++i)
    {
(3)   x = shared_array[i];
(4)   y = shared_array[i];
(5)   shared_array[i] = z;
(6)   shared_array[i] = 1;
(7)   x = shared_array[i];
    }
(8) barrier();

(a) Original Code
```

```
(1) barrier()
(2) for (i = 0; i < N; ++i)
    {
      make_page_readable(addr(shared_array[i]),
                   ,access);
(3)   x = shared_array[i];  /* no code reqd */
(4)   y = shared_array[i];
      make_page_writable(addr(shared_array[i]));
(5)   shared_array[i] = z;   /* no code reqd */
(6)   shared_array[i] = 1;    /* no code reqd */
(7)   x = shared_array[i];
    }
(8) barrier();

(b)Instrumented Code
```

Fig. 1. Motivating example.

page. The resulting code is shown in Fig. 1(b). In comparison to the code shown in Fig. 1(a), our approach does not incur any page fault overhead. However the cost of checking whether the page is consistent is incurred for each array access. In contrast, in the conventional software DSM systems, once a page is made consistent, further accesses to the same page do not incur any overhead until the page is invalidated at a synchronization point.

What is the granularity of sharing that is to be supported in our approach? While supporting granularity at shared variable (or object) level (e.g., as done in Ref. 27) makes compile-time analysis easier, it is wasteful in terms of both communication bandwidth and execution time, to make a full shared object consistent, especially if the object is a large area spanning multiple pages. Further, object level granularity would aggravate false sharing problem for large objects. On the other hand, supporting a lower level granularity less than a pagesize, e.g., a size of 64 bytes as in Ref. 15, in our approach would require *exact* compiler analysis for ensuring consistency of shared data. As will be discussed later, in the absence of exact compile-time analysis, instrumenting an application program conservatively would increase the overhead due to the instrumentation. Hence we support pagesize granularity in our approach and follow an *optimistic* approach in our analysis and code instrumentation. This implies that at places where the shared data can potentially be consistent, our approach optimistically avoids instrumentation. Whenever the optimistic assumption

fails and the accessed page is inconsistent, we rely on the underlying virtual memory support to make the data consistent.

Next we address the issue of the overhead incurred by the inserted code. It should be noted that the instrumented code by itself is an overhead. While the efficiency of inserted code can be ensured and improved by careful hand-tuning of the code, the overhead incurred due to frequent execution of the instrumented code can significantly degrade the performance. This is best explained with the help of our motivating example shown in Fig. 1.

From Fig. 1(b), it can be seen that the overhead of the instrumented code is incurred at every iteration of the `for` loop, although only once in a number of iterations it is required for ensuring consistency. Assuming a pagesize of 4 Kbytes, and the size of each of array element as 4 bytes, the consistency check is required roughly once in 1024 iterations. Thus the frequent invocation of the consistency check function may result in the instrumented version performing worse than the VM based systems. In order to reduce this overhead, accesses to shared data are analyzed. Determining which pages of the shared array will be accessed before the barrier at statement 8, and inserting a single function call to get those pages consistent before entering the `for` loop would reduce the overhead drastically.

Further, following an optimistic approach, we combine multiple consistency function calls for the same variable in order to reduce the overhead incurred at runtime due to the consistency function calls. Various standard compile time optimizations such as inlining and constant propagation were made use of to accomplish the above goal. It should be noted here that our instrumentation as well as the optimizations discussed in Section 4, including the aggressive optimistic discard optimization, always ensure the correctness of the application program under the given memory consistency model. That is, any access to shared memory is ensured to return a consistent data, under the lazy release memory consistency model,[26] either by our instrumentation or through the underlying virtual memory mechanisms.

Last, we briefly explain how our approach works in the presence of lock acquire and release synchronization primitives. In the example shown in Fig. 1, if the first barrier (in line (1)) is replaced by a `lock acquire (l)` and the second barrier (in line (8)) is replaced by a `lock release (l)`, then the consistency check calls inserted in the code would be exactly same as the ones shown in Fig. 1(b). That is, our analysis assumes that at the `lock acquire (l)` synchronization primitive, all shared pages could be modified by other processes and hence their consistency need to be checked. Our analysis does not consider or determine

which shared pages are modified by other processes before the synchronization step.[6] Last, for the release consistency model,[6, 23] a memory read following a `lock release (l)` does not introduce any consistency check, if the shared page has been accessed at least once since the last acquire or barrier. For a memory write access following the `lock release (l)`, a consistency check is introduced only if the diff creation is *eager*.

2.3. Reducing Holdup Time

The DVSM layer relies on the underlying message passing system for communication between nodes, and hence incurs a significant amount of time in communication, roughly 30 to 60% of the total execution time of the application. Although this varies considerably from application to application, it is still a significant portion of the total execution time. In a software DSM, many consistency related messages such as getting a page or diff from a remote node, or synchronization related communication for a barrier or lock acquire, wait for a response to arrive from a remote node. All these messages will arrive at the remote process asynchronously. Due to the overheads involved in the interrupt mechanism,[7] many software DSMs use polling for handling these messages. The polling-based approach is very efficient especially when a processor sends a request, such as a request for a shared page or lock acquire, and waits for its response. However, in receiving asynchronous message, the response to the requesting processor is delayed if the processor is busy doing computation and deferring polling. We illustrate the scenario with the help of an example.

Consider the following scenario shown in Fig. 2. The application program is running in two processors, `P0` and `P1`. Assume that an access to a shared location in `P0` caused the DSM layer to get control and initiate the steps for making the page consistent. At time t_0 processor `P0` sends a message m to `P1` for getting the modifications of this page. `P0` may continue to do certain computation after t_0. This computation may belong either to the application (if the DSM supports multithreading[28]) or to the DSM layer. Let this computation overlap with communication up to time t_2 and let process `P0` start waiting from time t_2. At time t_1 the message reaches the

[6] Such an analysis, in the first place, could be quite involved. Second, in lazy release consistency model, the analysis needs to consider only the pages modified by the process releasing the lock. However, it is not possible to determine this exactly at compile time.

[7] The interrupt overhead time, defined as the average time to process an interrupt with an empty interrupt service routine, is roughly 20 microseconds for the hardware platform used in this paper. On faster (more recent) workstations, the interrupt overhead is lower (5 to 10 microseconds) but is still in the order of a few microseconds.

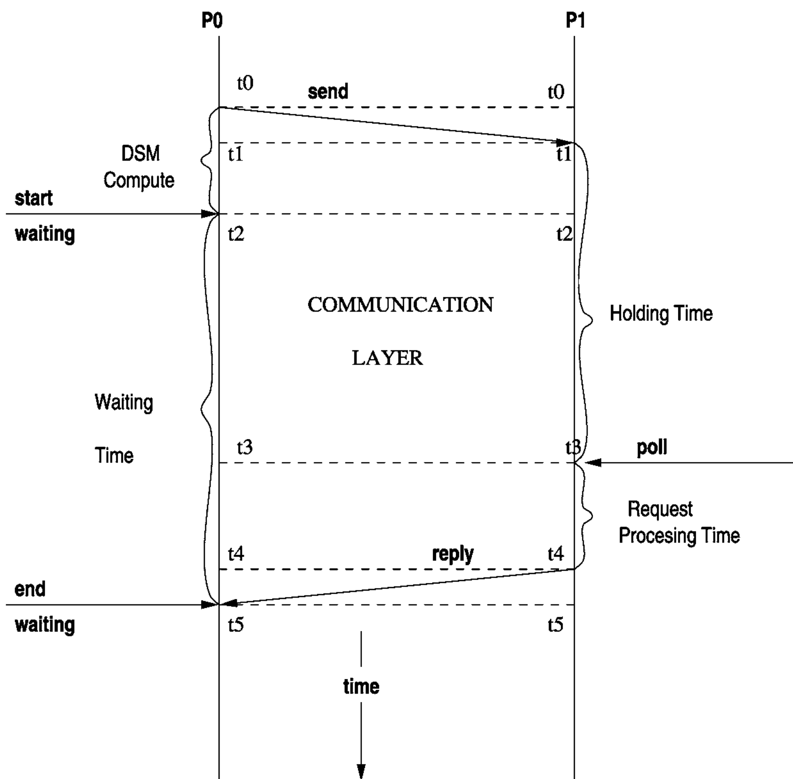


Fig. 2. Request-reply scenario in our DSM.

receiver and is held in a buffer until the receiver polls for messages at t_3. P1 polls for the message only when the DSM layer in P1 gets control. Let P1 poll at time t_3.[8] To process the message, P1 initiates the related computation, if any, and sends the reply to P0 at t_4. At t_5, the reply reaches P0 and is handled immediately, as P0 will be continuously polling for this reply.

The time from t_0 to t_5 is divided into different groups as shown in Fig. 2. Of these, the send and receive latencies (time delays $(t_1 - t_0)$ and $(t_5 - t_4)$ respectively) depend on the communication layer and are not in the direct control of the application program. Also, the *request processing time*,

[8] If there are other messages that have arrived in P1 before *m*, then there is also a waiting time in P1 involved before P1 can process the message. We can associate this with the request processing time.

$(t_4 - t_3)$ at P1 depends on the type of the request and the related consistency actions and is not in the control of user application. This leaves us with the holdup delay $(t_3 - t_1)$ of which $(t_2 - t_1)$ is overlapped by user/DSM computation at P0. The remaining time $(t_3 - t_2)$ is called the holdup delay. Our method targets the reduction of this holdup delay through careful insertion of polling functions in the application code.

To summarize, in this paper we explore a number of compile time techniques to reduce the overheads incurred by a software DSM. We have implemented CAS-DSM, a Compiler Assisted Software DSM and evaluated its performance. The following sections deal with the details of CAS-DSM, the compiler optimizations and performance evaluation.

3. CAS-DSM: BASE FRAMEWORK

An overview of the implementation of CAS-DSM is shown in Fig. 3. We perform a source-to-source translation of application programs. This transformation is based on analyzing the application program and instrumenting the application with code that ensures consistency of shared data. Consistency is achieved through a set of calls to API functions in the DSM layer. For our work, we chose CVM, a public domain DSM implementation,(14) as our base software DSM. CVM is an efficient implementation and supports multiple memory consistency models, which makes CVM a good choice for experimental work. One other reason for using CVM is that it can be easily ported to IBM-SP, our experimental platform. Last, CVM has also been used by a few other work (refer, e.g., Refs. 29, 30, and 31). In our work, we modified CVM to provide the necessary API for our implementation. For performing compile time analysis and instrumentation of code, we make use of the SUIF compiler framework.(32) The instrumented code is compiled and linked with the modified DSM libraries.

The executable is run on IBM-SP2,(33) a distributed memory machine. 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 the AIX operating system. The SP2 system used in our experiments consists of sixteen IBM RS/6000 591 wide nodes, each having 256 MB main memory, running at a clock speed of 77 MHz. The nodes are interconnected by a high-performance, multistage, packet-switched network for interprocessor communication.(34) The high-performance switch, TB2, is a low-latency, high-bandwidth switching network. We have used the *User Space* path for communication which gives a two-way latency of 75 microseconds and a bandwidth of 40 MB/seconds.

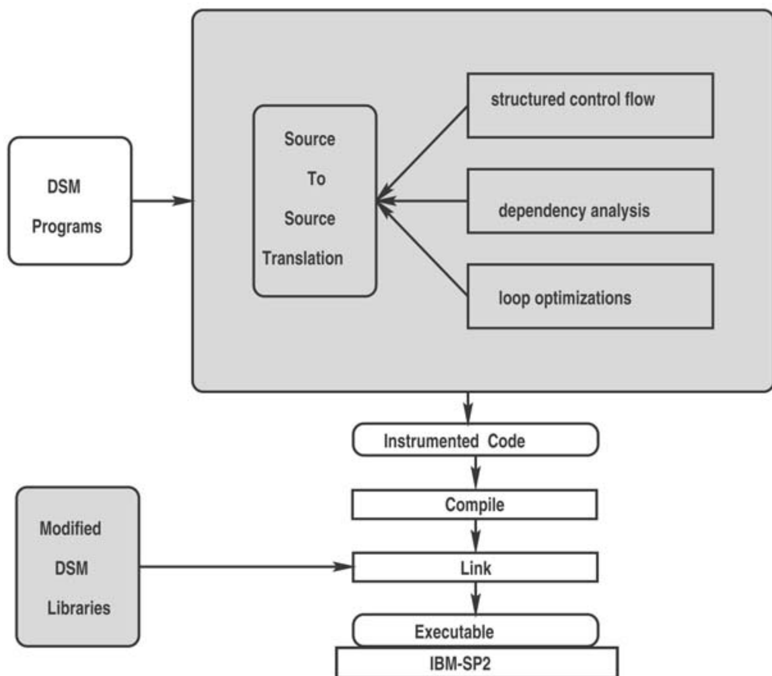


Fig. 3. Overview of our work.

3.1. The Compiler Framework: SUIF

SUIF (Stanford University Intermediate Format) compiler system is a platform for research on compiler-techniques for high-performance machines.[35] SUIF is a research compiler used for experimenting and developing new compiler algorithms. The compiler is structured as a small *kernel* plus a *toolkit* consisting of various compilation analysis and optimizations built using the kernel. The SUIF kernel provides an object-oriented implementation of the SUIF intermediate format. The intermediate format is a mixed-level program representation. Besides the low-level constructs such as SUIF instructions, this representation includes three high-level constructs: loops, conditional statements and array access operations. The high level compiler passes typically represent a procedure as a list of language-independent abstract syntax trees (ASTs). An AST includes block-structured constructs such as loops and conditionals. At the leaves of an AST are expression tress, composed of SUIF instructions.

The SUIF system consists of a set of compiler passes implemented as separate programs. Each pass typically performs a single analysis or

transformation and then writes the results out to a file. New passes can be inserted freely at any point in a compilation. Different compiler passes interact with one another either by updating the SUIF representation directly or by adding annotations to the SUIF program. Each kind of annotation is defined with a particular structure so that definitions of the annotations serve as definitions of the interface between passes. We have made use of the following passes for our work:

- `scc` is the driver for the SUIF ANSI C and FORTRAN 77 compiler.
- `porky` makes various transformations to the SUIF code.
- `s2c` reads the specified SUIF file and prints out its translation into the Standard C language.

3.2. The DSM System: CVM

CVM[14] is a software DSM that supports multiple coherence protocols and consistency models. CVM is written entirely as a user-level library and runs on most UNIX-like systems. CVM provides a multiple-writer protocol to reduce false sharing. CVM's primary protocol implements a multiple-writer version of lazy release consistency model. The UNIX `mprotect` system call is made use of to control access to shared pages. In particular, the inconsistent shared pages are read or write protected. An attempt to perform an access, to a location in these shared pages generates a `SIGSEGV` signal. The `SIGSEGV` signal handler, initiates the steps necessary to make the page adhere to the lazy release multiple writer protocol.

3.3. Basic Implementation

In our work, we start from a basic implementation. We used the high-level intermediate representation, called high-SUIF, and our analysis traverse through this abstract syntax tree (AST) representation to identify and mark shared accesses. Using these markings (or annotations), our compiler pass inserts the consistency function before every shared memory access in the AST. Finally the annotated AST is converted to C using `s2c`.

Our analysis is restricted to only array and scalar references in `for` loops. It does not deal with dynamic structures or references in `while` or `do` loops. In the base implementation, the consistency check is added before the first shared reference (of a specific type – read or write) after a synchronization point, exactly as shown in Fig. 1. Further, our analysis is between two successive synchronization points in the code and is limited to

accesses within a process. It is possible to extend our analysis to consider accesses made by different processes in between two barrier points, and hence obtain information on the pages that are (likely to be) modified by other processes, as done in the work of Lu *et al.*[36] However, we do not attempt this here.

3.4. Performance

We evaluated performance of the basic implementation. We call this implementation CAS-Basic. We used some of the Splash/Splash-2 benchmark programs.[22, 37, 38] The benchmarks used are listed in Table II with a brief description for each. All the benchmarks were run on an IBM SP2[33] system using *User Space* Communication mode. The benchmarks were run for different configurations ranging from 1 to 8 processors.

We observed that the performance of the applications on CAS-Basic is significantly poorer than in the original CVM. This is because, while the overhead due to page fault is almost eliminated in CAS-Basic, the overheads due to the instrumented code results in a significant slowdown, which more than offsets the gain achieved by eliminating page fault overhead.

For all the benchmarks we measured, less than *one* percent of the CVM_Basic_consistency calls actually translated into consistency related calls (`cvm_make_page_consistent()`). This implies that on the remaining calls, the instrumented function found the page to be already consistent. In these cases the checking for consistency merely causes heavy run-time overheads. The original virtual memory based approach does not

Table II. Benchmarks

Name	Brief description	Input Size
SOR	Models natural phenomena like determining the temperature gradients over a square area.	2000×500 for 10 K iterations
FFT	3-Dimensional Fast Fourier Transform which numerically solves partial differential equations.	$2^6 \times 2^6 \times 2^4$ integers
LU	Factors a dense matrix as the product of upper triangular and lower triangular matrices.	1024×1024 matrix block size = 16
TOMCATV	Solves tridiagonal system of equations by computing residuals using 2-dimensional matrices.	257×257 for 1000 iterations
RADIX	Implements an integer sort by generating histograms in an iterative method.	$2 \times 1024 \times 1024$ radix = 1024

incur any overhead for this, since the SEGV signal is generated only if the page does not have the desired access permissions. Thus our experimental results clearly indicate that the basic implementation is highly inadequate and results in a significant slowdown. Aggressive compiler techniques are needed to reduce this overhead. In the following section, we discuss various such optimizations.

4. OPTIMIZATIONS

In this section we discuss various compile time optimizations which are motivated by the significant overhead incurred by the consistency checking code in our basic implementation. Our analysis targets shared memory array accesses, within `for` loops. We restrict our analysis to array accesses with linear indices—linear with respect to the indices of the `for` loop that contain this access. Also, using `porky` we convert as many `while` loops as possible to appropriate `for` loops. This increases the scope of our analysis. For ease of analysis, all `for` loops are *normalized* using `porky`. A *normalized* `for` loop is the one in which the lower bound is zero, step size is one and condition is a "less than or equal to" test. In the following sub sections we describe the various optimizations and their implementation.

4.1. Aggregation and Hoisting

The idea behind hoisting the instrumented code is best explained with an example. Consider the program segment shown in Fig. 4(a). Statements are labeled for easy reference in the discussion. The instrumented version is shown in Fig. 4(b), with the inserted code given in italics. The variable `shared_a` and `shared_b` are integer arrays. Consider the first access to `shared_a` inside the `for` loop. We see that the array is accessed from location 0 to `N-1`, consecutively. So, instead of instrumenting code once for every shared access, we can aggregate the instrumentation for all the iterations of a shared access and hoist this aggregate outside the `for` loop. This will cause the instrumented code to be executed less frequently, thereby reducing the associated overheads substantially. The pseudo code for this function, CAS_AgHoist is shown in Fig. 5. Henceforth, this optimization will be referred to as *AgHoist*.

The input parameters to the instrumented function are the base address of the concerned array, the size of each array element, the boundary indices of the array access, and the type of access. First, the CAS_AgHoist function will determine the addresses of the *start* and the *end* elements accessed as shown in Fig. 5. Next, the page numbers of the

```
S0:  for (i = 0; i <= N-1; ++i)
     {
S1:       x = shared_a[i];
S2:       y = shared_b[i + 1];
S3:       shared_a[i + 1] = z;
     }
```

(a) Original Code

```
S4:  CAS_AgHoist(shared_a, elemsz, 0, N - 1, READ);
S5:  CAS_AgHoist(shared_a, elemsz, 1, N, WRITE);
S6:  CAS_AgHoist(shared_b, elemsz, 1, N, READ);
S0:  for (i = 0; i <= N - 1; ++i) {
S1:       x = shared_a[i];
S2:       y = shared_b[i + 1];
S3:       shared_a[i + 1] = z;
     }
```

(b) Instrumented Code after AgHoist

Fig. 4. Original code and instrumented code after aggregation and hoisting.

shared page in which these addresses fall are calculated. Finally, the pages are checked for the corresponding consistency, and are made read/write accessible through `cvm_make_page_consistent`.

As discussed above, AgHoist instruments the consistency code and hoists it above the `for` loop where shared access occurs. As a logical extension to this, the aggregation and hoisting can be done above the outer

```
CAS_AgHoist(shared_addr, elemsz, begin, end, access) {

    addr_begin = shared_addr + (begin × elemsz);
    addr_end   = shared_addr + (end × elemsz);

    page_begin = shared_page(addr_begin);
    page_end   = shared_page(addr_end);

    if ((page_begin not a shared_page) and
            (page_end not shared page))
            return;

    if (access == READ) {
        for (page = page_begin to page_end)
            if (page not readable)
                cvm_make_page_consistent(page, READ);
    } else {
        for (page = page_begin to page_end)
            if (page not writable)
                cvm_make_page_consistent(page, WRITE);
    }
}
```

Fig. 5. Pseudo code for CAS_AgHoist.

`for` loops also until the next synchronization primitive is encountered. The maximum hoisting and aggregation phase is henceforth referred to as MaxAgHoist phase.

It should be noted here that the `cvm_make_page_consistent` call is *blocking*; i.e., the process waits for responses to the `diff` requests which are received through polling. Once the process receives the `diffs`, it applies them and to make the page consistent. Also, the calls to make pages consistent in `CAS_AgHoist` are done one at a time as shown in Fig. 5. Further discussion on this is deferred to the following subsection.

4.2. Discarding

Discarding some of the instrumentation code is another technique used to improve the performance. The idea behind discarding is to group accesses due to different statements in the source program and insert only a single CAS_AgHoist call. The intuition behind this optimization is that the different shared accesses inside a loop have a large overlapping section between them. For example in Fig. 4 consider the accesses to `shared_a[i]` and `shared_a[i+1]` in statements `S1` and `S3` respectively. Except for the extreme elements, i.e., `shared_a[0]` and `shared_a[N]` all the other elements in between these two are touched in both the accesses. Thus even if we discard one of the two calls to `CAS_AgHoist` it may still be possible to ensure consistency of the elements accessed. Discarding is done after the AgHoist pass. Henceforth the term AgDiscard will also refer to discarding after AgHoist. This is because the non-overlapping accesses are also likely to fall in the same virtual page as those of overlapping pages. However if they do not, then the underlying VM based mechanism ensures consistency. Thus, discarding one of the AgHoist calls results in reducing the overhead due to the call.

An important issue in discarding is, given a number of choices, what are the guidelines to be followed to select the `CAS_AgHoist` calls to be discarded? One of the guidelines is the type of the access. If we are given a choice between two AgHoist calls, one for read and another for write access to shared data, then the former is discarded. This because the consistency related operations done for a write operations form a superset of those done for a read. Another criterion which influences this choice is the amount of information available about an access. For example, assume that for one access we know the stride is less than a page size and in other we do not know the stride at all. In this case, we instrument code for which the stride is known at compile time while discarding the other. In such a case, by instrumenting the code for an access for which the stride is

known, the application will *not* incur the additional overhead of stride calculation during runtime.

Finally, it should be noted that the technique of discarding is based on *heuristic* and not on exact analysis. This is because exact analysis is expensive, resulting in run-time overhead, and doesn't give any significant advantage compared to our heuristic approach. If discarding is performed after MaxAgHoist, we refer to it as MaxAgDiscard.

As shown in Fig. 5, the calls to make pages consistent in `CAS_AgHoist` are done one at a time. However it is possible to aggregate this which amounts to aggregating the `diff` requests sent for various pages to different processes. We have implemented a new function in CVM, `cvm_make_all_pages_ consistent`, which sends a single message to each relevant process requesting the `diffs` for all the pages that need to be made consistent. Each process, in response, sends a single aggregated message, of the `diffs` for the various pages. It should however be noted here that the number of pages which are made consistent in a single `cvm_make_all_pages_consistent` call varies from application to application. We study the impact of message aggregation by applying this optimization with MaxAgDisard. We refer to the resulting implementation as MaxAgDiscard+MsgAggr. Thus when we refer to MaxAgHoist and MaxAgDiscard optimizations, messages are not aggregated in these implementations.

4.3. Inlining and Constant Propagation

It should be noted that our analysis is intra-procedural. Performing inter-procedural analysis will improve the scope for MaxAgHoist and MaxAgDiscard as well as enable better compile time analysis such as identifying a constant in the function body which was passed as a parameter. We motivate the discussion with an example.

Consider the function `fn` shown in Fig. 6(a). This function is accessing a shared variable inside a loop. Also, it can be seen that the function is called within a loop from another function `main`. If we instrument the code using the optimizations discussed so far, the resulting code is shown in Fig. 6(b). Since the MaxAgHoist and MaxAgDiscard are limited to function boundaries, only single level aggregation is achieved in this code. As a result, the consistency function will be executed 101 times, once for every loop within the function and once in main. But if we inline the function in main, as shown in Fig. 7(a), and then with MaxAgHoist, we get the instrumented code as shown in Fig. 7(b). The consistency code will get executed just *once* in this case, leading to a substantial reduction in the overhead. Further, constant propagation of the inlined code leads to

```
void fn(shared_a, n, base) {
for (i = 0; i <= n-1; ++i)
   shared_a[base+i] = x;
}

int main() {
base = 50;
for (j=0; j<=99; ++j)
   fn(shared_a,j,base);
}
```

(a) Original Code

```
void fn(shared_a, n,base) {
CAS_AgHoist(shared_a,base,(base+n-1),W);
for (i = 0; i <= n-1; ++i)
   shared_a[base+i] = x;
}

int main() {
base = 50;
CAS_Basic_consistency(shared_a,0,W);
for (j=0; j<=99; ++j)
   fn(shared_a,j,base);
}
```

(b) Instrumented Code

Fig. 6. Original version.

determining exact end points of the shared accesses at compile time itself. Hence the overhead in calculating some of the runtime boundaries is also avoided. Such a code segment occurs in one of our benchmarks, namely LU decomposition. This motivates the need for inter-procedural analysis. Since our compiler framework does not support such an analysis, we manually inline and perform constant propagation to enhance the scope of

```
void fn(shared_a, n, base) {
for (i = 0; i <= n-1; ++i)
   shared_a[base+i] = x;
}

int main() {
base = 50;
for (j=0; j<=99; ++j)
   for (i = 0; i <= n-1; ++i)
       shared_a[50+i] = x;
}
```

(a) Inlined Original Code

```
void fn(shared_a, n, base) {
CAS_AgHoist(shared_a,base,(base+n-1),W);
for (i = 0; i <= n; ++i)
   shared_a[base+i] = x;
}

int main() {
base = 50;
CAS_AgHoist(shared_a,50,149,W);
for (j=0; j<=99; ++j)
   for (i = 0; i <= n-1; ++i)
       shared_a[50+i] = x;
}
```

(b) Inlined Instrumented Code

Fig. 7. Inlined version.

MaxAgHoist and MaxAgDiscard. Once the function body is manually inlined, our compiler framework is able to perform constant propagation, and the instrumentation along with the optimizations. We observe significant improvement in performance in the case of LU due to manual inlining and constant propagation. Automating selective function inlining requires inter-procedural analysis in our SUIF framework. This is left for future work. Once inter-procedural analysis is included in the framework, it enables other compiler optimizations as well which could be beneficially applied in our framework.

4.4. Reducing Holdup Delays

Next we propose a method to reduce the *holdup time*. Referring to Fig. 2, it can be seen that the holdup time arises essentially due to the asynchronous nature of the message arriving in `P1`. In the CVM implementation, the polling for the message starts only when the DSM layer takes control (of execution). To reduce the holdup time, we must invoke the polling actions as often as needed. CVM provides an API function `cvm_probe`. In our approach, we instrument explicit polling calls at the application source code itself to moderately increase the frequency of polling. Since most of the shared accesses in our programs occur within `for` loops, we decided to introduce calls to polling function above the `for` loops. Calling the `cvm_probe` too often would increase the overhead, and hence, we combined the optimizations discussed in the previous sections with the polling function and instrumented both of them together.

Multithreading can be used to overlap computation with communication in order to the mask the communication overhead in processor `P0`. Essentially, multithreading masks the waiting time seen at the sender's side (`P0`). Whereas our approach, reduces the holdup time at the receiver's end. It is possible that when there is sufficient computation available for overlapping, i.e., sufficient enough to cover the holdup time, multithreading will perform better than our method. On the other hand, if the computation overlap is less, then multithreading can be used to complement our method. As mentioned in Ref. 28, multithreading comes with its own overheads such as thread switch cost, cache pollution, and local contention of resources between threads in the same node. In contrast, the overhead associated with our method is the overhead of polling and thus the total overhead depends on the total number of times the polling function is called. By performing more compile time analysis, it is possible to reduce this overhead in our method.

5. IMPLEMENTATION

In this section, we discuss the implementation of the above optimizations in the compiler framework.

5.1. Compiler Analysis and Optimization Passes

The source code of the application is first converted into the SUIF intermediate format. The analysis and instrumentation parts, henceforth referred to as *aggregation phase*, has different passes associated with it. The details of different passes are shown in Fig. 8. Communication of information between successive passes in SUIF is through annotations in the intermediate format.

Pass 1, the shared array access identification pass, identifies the accesses to shared arrays, using the keyword *shared*. The type of access is also identified and the information is annotated on the array access instructions. Pass 2 identifies the innermost `for` loop. Simultaneously, the code is traversed checking for the presence or absence of synchronization constructs.

During Pass 3, variable write detection and linear access identification pass, for every `for` loop, which does not contain any synchronization construct, the list of all variables (shared as well as non-shared) which are written in that `for` loop are identified and pointers to each of these are stored in the annotations of the `mrk` node, a structure in SUIF where annotations can be stored. This linear access information is used in the next pass to determine whether or not aggregate analysis can be performed. Also, for all the shared array accesses within a `for` loop, checks are done to see whether the array indices are linear with respect to the `for` loop index. This information is annotated at the respective array instructions. Also, the pointers to the linear array accesses are kept at the `mrk` node so that the analysis can be made faster in the subsequent passes.

Pass 4 is the dependency checking pass. In this pass, the array indices are separated out and for each variable in the index, a check is done to determine whether that variable is being written inside the body of the `for` loop. This is done by looking at the variable write list generated in the previous pass. It should be noted that for non-linear array indices, or for an array access whose index variables are written within a loop, no consistency check instrumentation is done and we rely on the fall back option of VM mechanism. Address formation pass, Pass 5, generates some of the parameters to the CAS_AgHoist function call. This pass will find the start and end elements in the array that are accessed in the `for` loop.

The next pass (Pass 6) uses the annotations generated during the previous passes and identifies the API function to be inserted. If discarding is

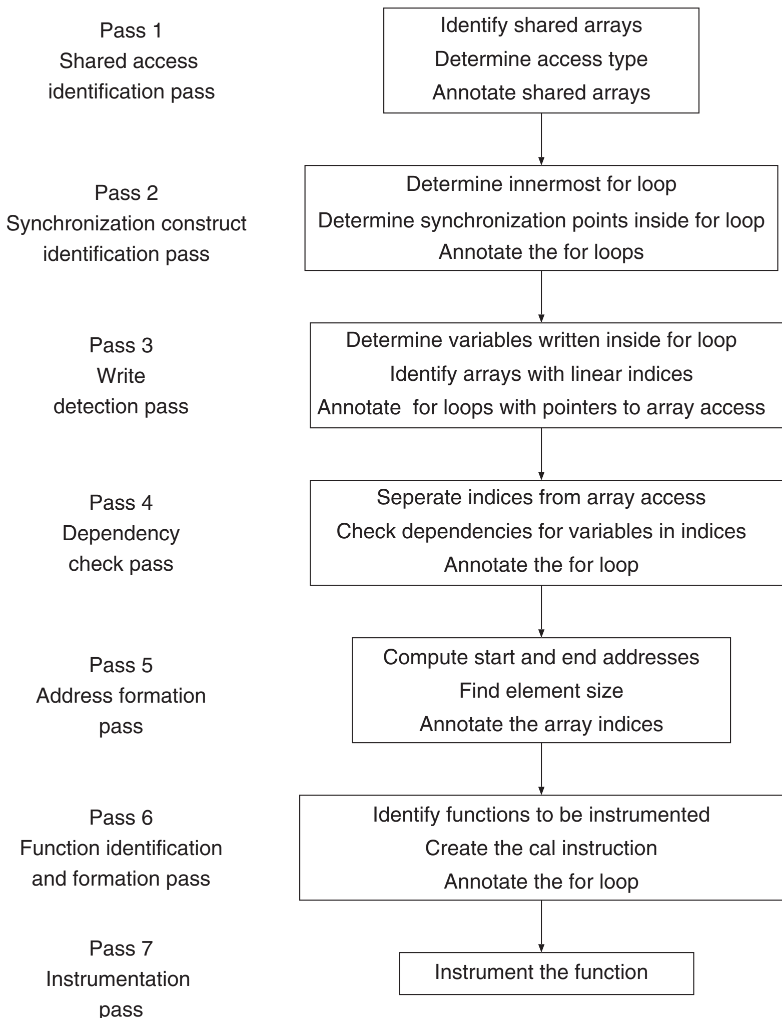


Fig. 8. Compiler passes.

done, then the function call to be discarded is decided based on the previously discussed criteria and marked as discarded. This selection is based on the amount of information available about the accesses as well as the stride of the access. The MaxAgHoist phase is implemented within the aggregate phase, but before the discarding pass. The MaxAgHoist is done iteratively, by calling this pass repeatedly for each level of `for`. Each pass does the aggregation and hoisting for the `for` loop at a particular level.

The AgDiscard, MaxAgHoist, and MaxAgDiscard passes (not explicitly shown in Fig. 8) are implemented one after another as shown in Fig. 9. The final pass (Pass 7) does the actual instrumentation of the code. In actual implementation, only very few passes exist since multiple passes are combined into one. Only for the sake of presentation we have organized them as so many different passes.

Currently, our compiler is not performing inter-procedural analysis. Hence, in order to study the performance improvements due to inter-procedural analysis, we inlined the functions manually and applied the MaxAgHoist and MaxAgDiscard optimizations. Also, we performed constant propagation, so that more information is made available for the compile time analysis. Inlining is useful if a particular function is called inside a loop and also if the function is accessing shared variables.

As discussed before, holdup time reduction is achieved by instrumenting explicit polling functions in the application programs. The polling function will check whether any messages are waiting and passed on to the message handler. The polling function returns after servicing all outstanding requests. The implementation of this optimization is preliminary, since we perform only very limited analysis for the instrumentation with polling function. The polling function is *piggy-backed* on the existing optimizations.

5.2. The global picture

In this section we discuss the order in which the optimization are applied to the application source code. All the passes are shown in the Fig. 9. In this figure, rectangular boxes with rounded corners represent original or translated code while other rectangular boxes represent compiler phases. The name of the compiler phase is shown in italicized font adjacent to the rectangular box, while the text within the rectangular box indicates the action performed by the phase.

The source code of the application program is run through the C preprocessor `cpp`. In case, inlining is done, it should be done before preprocessing. As mentioned before, inlining is done manually in the current implementation. After preprocessing, the source code is converted to the SUIF intermediate code using `snoot`. Next a series of optimizations such as dismantling the if loops, forward propagation, dead code elimination and find-fors are done. The find-for phase will build `for` loops out of loop nodes such as `while` ··· `do` for which a suitable index variable and bounds can be found. Since our analysis is based on for loops, this phase increases the scope of our optimization. Constant propagation is optionally invoked by `porky`.

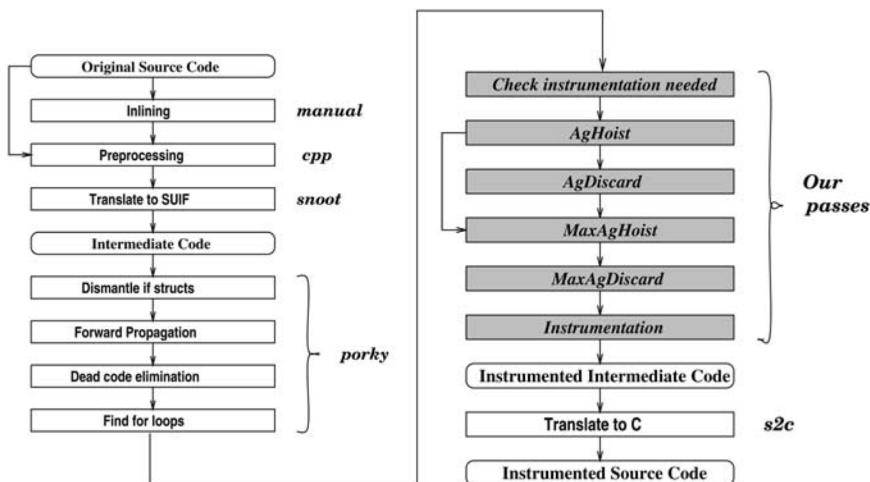


Fig. 9. The global picture.

The porky passes are followed by our optimization passes. It should be noted that our passes operate for a given procedure.

Our optimization passes AgHoist, AgDiscard, MaxAgHoist, MaxAgDiscard are applied in the order shown in Fig. 9. Then the program is instrumented with the API functions. The instrumented code is then translated back to C code, using `s2c`. The modified source code thus obtained is compiled using the `gcc` compiler in IBM-SP2, and linked with the enhanced DSM libraries.

5.3. Enhancements to CVM

In this section we discuss the implementation of the API. We have provided different API functions to be inserted in the application depending upon the amount of information available. The different API functions are summarized in Table III. The `CAS_Basic_consistency` is used in CAS-Basic implementation. With the optimizations discussed in this section, the `CAS_AgHoist_const` function is typically used in our implementation. This function is used when the index expressions of the *start* and *end* elements of the array are compile time constants and the stride of access is less than the pagesize. Two variants of this function, `CAS_AgHoist_expr` and `CAS_AgHoist_step`, are implemented and used respectively when the index expression do not evaluate to constants and the step size is greater than the pagesize. In these cases, the evaluation

Table III. Instrumentation API in CAS-DSM

Function name	Parameters	Brief description
CAS_Basic_consistency	`addr`, `type`	Makes a single page which contains `addr` consistent
CAS_AgHoist_const	`base`, `type` `pg_start`,`pg_end`	Makes multiple pages `pg_start` to `pg_end` consistent
CAS_AgHoist_expr	`base`,`type` `start`, `end`	similar to `CAS_AgHoist_const` except that `start` and `end` are address expressions evaluated at runtime
CAS_AgHoist_step	`base`, `type` `start`, `end`, `stride`	similar to `CAS_AgHoist_expr` except that the stride is greater than pagesize stride is evaluated at runtime

of index expressions takes place at run time and hence incur additional overhead. The assembly code of all four functions are hand-tuned to minimize the overhead incurred by these calls. Hand-tuning the assembly code has resulted in reducing 1 or 2 assembly instructions in the instrumentation functions.

6. PERFORMANCE EVALUATION OF CAS-DSM

In this section, we report the performance of CAS-DSM. We applied the optimizations in the source to source translation of benchmarks. We have compared the performance of CAS-DSM (with optimizations implemented) with that of original CVM implementation. Both were implemented on IBM-SP2. Although the platform used in our experimental work is a somewhat slower/older machine, the focus of our experimental results is on the relative advantages/benefits of the different optimizations discussed in Section 4. We use the metric *Relative Speedup* for performance comparisons, which is defined as

$$\text{Relative Speedup} = \frac{\text{CVM Exec. Time}}{\text{CAS-DSM Exec. Time}}.$$

We evaluate the performance improvement due to individual optimizations as well as their collective impact[9] Apart from the speedup we are also interested in the number of segvs eliminated. Finally, we are also interested

[9] The execution times reported in the paper are obtained by running each experiment a number of times (typically 3, and sometimes 5) and taking the average of these runs.

in how many `CAS_AgHoist` calls actually translated into consistency related steps. Such calls are referred to as *useful calls*. Other calls correspond to the overhead incurred by CAS-DSM. We measure the percentage of useful calls to the total calls in CAS-DSM. In the subsequent sections, we discuss the effectiveness of various optimizations based on these three metrics.

6.1. Performance of Aggregation, Hoisting and Discarding

The relative speedups achieved for different optimizations are shown in Table IV. For comparison purposes, we also report the execution time of all applications under different optimizations in Table V. Since our focus is to evaluate how the different optimizations (AgHoist, AgDiscard, MaxAgHoist, MaxAgDiscard, and MaxAgDiscard+MsgAggr) improve the performance of the application, we do not report the absolute speedup with increasing number of processors. It can be noted that some of these applications do have a poor speedup (with number of processors) even on the original CVM.

From Table IV, we can observe that the CAS-DSM approach with MaxAgHoist, MaxAgDiscard, and MaxAgDiscard+MsgAggr is able to achieve a small performance improvement over CVM. The performance improvement achieved by our approach is governed by three factors: (i) the reduction in number of segmentation violations, (ii) overhead incurred by the consistency check, measured using the percentage of useful calls, and (iii) reduction in message overhead due to fetching of pages in an aggregated fashion. The percentage reduction in the number of segmentation violations due to the instrumented code is shown in Table VI for 4 and 8 processor runs. Table VII reports the number of consistency check calls and the percentage of *useful calls*, those which necessitate a consistency action, for the 8 processor case. In the following paragraphs, we discuss the results of each benchmark and the reasons for performance improvement or degradation using these parameters for the 8 processor case.

In SOR, a relative speedup of upto 1.08 is achieved with AgDiscard on a 8 processor system. Also, we are able to get almost 100% reduction in the segmentation violations. However, we can see that the consistency check overhead is quite high as only less than 1% of the total checks are useful. One reason for the low percentage of useful calls is that, in introducing the consistency checks for a memory access following a synchronization primitive, our approach does not consider/analyze whether or not the shared data is modified before the synchronization. It conservatively assumes all shared data have been modified. Further, the absence of runtime information also causes a significant number of useless consistency check calls.

Table IV. Relative Speedups on CAS-DSM under Various Optimizations

No. of procs.	AgHoist	AgDiscard	MaxAgHoist	MaxAgDiscard	MaxAgDiscard + MsgAggr
			Relative Speedup (SOR)		
1	1.03	1.04	1.03	1.04	1.18
2	1.05	1.06	1.05	1.06	1.20
4	1.05	1.06	1.05	1.06	1.20
8	1.07	1.08	1.07	1.08	1.18
			Relative Speedup (FFT)		
1	0.92	0.96	0.99	0.99	1.01
2	0.97	1.02	1.04	1.04	1.04
4	0.94	1.01	1.00	1.04	1.04
8	0.99	1.03	1.05	1.04	1.06
			Relative Speedup (LU)		
1	0.31	0.31	0.31	0.31	0.31
2	0.38	0.38	0.39	0.39	0.39
4	0.45	0.45	0.46	0.46	0.46
8	0.55	0.55	0.56	0.55	0.55
			Relative Speedup (TOMCATV)		
1	0.88	0.95	0.94	0.95	0.98
2	0.91	0.98	0.97	0.98	0.98
4	1.08	1.15	1.10	1.22	1.23
8	1.41	1.46	1.45	1.47	1.47
			Relative Speedup (RADIX)		
1	1.01	1.01	1.01	1.01	1.01
2	1.02	1.02	1.02	1.02	1.08
4	1.01	1.01	1.01	1.01	1.03
8	1.01	1.02	1.01	1.02	1.04

In can be observed that discarding does not significantly reduce the number of SEGVs eliminated (refer to Table VI), yet improves the percentage of useful calls. This in turn reduces the consistency check overhead and improves the performance at least marginally. It should be noted that in SOR, maximum aggregation is the same as AgHoist and hence the columns MaxAgHoist and MaxAgDiscard represents the same figures as AgHoist and AgDiscard. The maximum improvement in performance is seen in the case of MaxAgDiscard + MsgAggr. Fetching pages in an aggregated fashion as against fetching them sequentially at a consistency check decreases the message overhead. This results in a speed-up of upto 1.20.

Table V. Execution Time on CAS-DSM under Various Optimizations

No. of procs.	Original CVM	CAS-DSM				
		AgHoist	AgDiscard	MaxAgHoist	MaxAgDiscard	MaxAgDiscard + MsgAggr
		Execution Time in Seconds (SOR)				
1	2974.31	2901.11	2865.01	2901.11	2865.01	2520.60
2	3086.82	2958.05	2921.07	2958.05	2921.07	2572.35
4	1607.89	1534.69	1516.18	1534.69	1516.18	1339.90
8	864.53	809.36	803.65	809.36	803.65	732.50
		Execution Time in Seconds (FFT)				
1	253.72	275.37	264.12	257.07	256.39	251.20
2	256.92	264.38	252.73	246.07	248.11	247.04
4	177.69	188.60	175.76	177.67	171.60	170.86
8	137.25	139.29	133.68	132.15	131.53	129.48
		Execution Time in Seconds (LU)				
1	18.13	59.25	59.23	57.72	57.89	57.89
2	24.61	64.00	63.94	62.47	62.69	62.69
4	17.73	39.28	39.31	38.36	38.78	38.78
8	13.58	24.90	24.58	24.28	24.81	24.81
		Execution Time in Seconds (TOMCATV)				
1	48.49	55.14	50.89	51.87	51.21	49.47
2	53.05	58.14	54.08	54.96	54.32	54.13
4	39.87	37.01	34.79	36.39	32.66	32.41
8	34.20	24.28	23.33	23.58	23.20	23.20
		Execution Time in Seconds (RADIX)				
1	12.18	12.03	12.07	12.03	12.07	12.06
2	19.83	19.38	19.39	19.38	19.39	19.39
4	13.68	13.54	13.49	13.54	13.49	13.49
8	10.56	10.40	10.38	10.40	10.38	10.38

FFT, on the other hand suffers a small performance degradation with only AgHoist optimization. The relative speedup ranges from 0.92 to 0.99. We can observe that the percentage reduction in segmentation violations, is only between 38 to 54%. The reason for the large number of segmentation violations, even in the instrumented case, is due to the fact that array accesses in FFT are with non-linear indices. Since we are not analyzing non-linear array indices, but chose the fall-back VM support for the same, these

Table VI. Percentage Segmentation Violations Eliminated

Optimizations	No. of procs.	Benchmark				
		SOR	FFT	LU	TOMCATV	RADIX
AgHoist	4	99.99%	54.42%	80.00%	47.24%	9.46%
AgHoist	8	99.99%	38.60%	85.75%	53.27%	8.76%
AgDiscard	4	99.99%	53.91%	80.00%	47.16%	9.46%
AgDiscard	8	99.99%	38.46%	85.75%	53.27%	8.41%

accesses result in segmentation violations. Further, in FFT, the percentage of useful calls is very low (0.1%), leading to high consistency check overhead. With AgDiscard optimization, we get a relative speedup of 1.03 with respect to the original. It can be seen that the percentage of useful calls have been increased by a factor of 10 when compared to AgHoist. This substantial reduction in the overhead has caused the performance gain. Also FFT achieved a relative speedup of 1.04 to 1.05 with MaxAgHoist and MaxAgDiscard.

Table VII. Statistics on Consistency Checks and Useful Calls

Benchmark	Performance metric	Optimization			
		AgHoist	AgDiscard	MaxAgHoist	MaxAgDiscard
SOR	# Consistency checks	200.000 M	160.000 M	200.000 M	160.000 M
	% Useful calls	0.358%	0.448%	0.358%	0.448%
FFT	# Consistency checks	68.891 M	6.664 M	2.059 M	0.773 M
	% Useful calls	0.212%	2.190%	7.090%	18.770%
LU	# Consistency checks	46.758 M	46.757 M	45.384 M	45.384 M
	% Useful calls	0.027%	0.027%	0.028%	0.028%
TOMCATV	# Consistency checks	2.212 M	1.507 M	1.507 M	1.262 M
	% Useful calls	0.462%	1.595%	1.584%	1.612%
RADIX	# Consistency checks	0.013 M	0.007 M	0.013 M	0.007 M
	% Useful calls	8.420%	15.560%	8.420%	15.560%

Even with aggressive optimizations, LU performs worse. We can see a slowdown by a factor of 2 to 3. Although, the reduction in number of segmentation violations ranges from 75 to 85% (refer to Table VI), the main overhead comes in the consistency checks. For the different number of processors, the useful calls is less than 0.02%, which is an order of magnitude lower when compared to other benchmarks (see Table VII). Also, there is little difference in the execution time with and with AgDiscard optimization. Lastly, the performance of LU does not exhibit much change even after MaxAgHoist or MaxAgDiscard optimizations, as can be seen from Table IV.

In the case of TOMCATV, a performance improvement of upto 41% is achieved with AgHoist. AgDiscard, MaxAgHoist, and MaxAgDiscard optimizations results in further improvement in the execution time as is evident from Table IV. The consistency overhead is comparable to that in SOR or FFT. The relative speedup also improves when the number of processors is increased from 2 to 4 or 8. In this benchmark, the reduction in holdup time plays an important role in the performance gains. We defer a discussion on this to Section 6.3. As will be shown in Table VIII, there is a significant improvement in the percentage of useful calls with 4 and 8 processors which also contributes to this improvement in relative speedup.

Lastly, in RADIX, the improvement in execution time is minor (1 to 2%). This lower performance improvement could be due to the fact that the percentage of segmentation violations reduced is very small (less than 10%). In radix a large majority of its shared references are of the form `a[b[i]]`, which is not amenable to compile-time analysis. Hence, we do not instrument this reference and rely on the underlying VM support to take care of this access. The consistency check overhead in RADIX is comparatively low, as the percentage of useful calls is roughly 8%. This is an order of magnitude higher when compared to other benchmarks. Further, the actual number of consistency calls in RADIX is also low resulting in relatively low overhead. Even with the AgDiscard optimization, there is not much difference in performance. However, it can be seen that the percentage of useful calls has increased roughly by 7% with discarding. Also, in RADIX, the performance of MaxAgHoist and MaxAgDiscard are same as AgHoist and AgDiscard respectively, as the consistency check calls cannot be moved out of the innermost loop due to synchronization primitives.

In all of the above applications except SOR, MaxAgDiscard + MsgAggr performs only marginally better than MaxAgDiscard. As explained in Section 4.1, we aggregate messages at a consistency check, i.e., if the check determines that n pages are inconsistent, and (say) Proc 1 is the last writer to all the n pages, then only one message is sent to fetch the n pages

Table VIII. Improvement in Percentage of Useful Calls with Number of Processors in MaxAgDiscard Optimization

No. of procs.	Percentage of useful calls in				
	SOR	FFT	LU	TOMCATV	RADIX
1	0.003	0.09	0.001	0.12	14.18
2	0.123	12.59	0.013	0.31	14.56
4	0.223	15.15	0.015	0.64	14.95
8	0.448	18.77	0.028	1.62	15.56

as against *n* individual messages (sent in the case of MaxAgDiscard). It can be easily seen that aggregation can improve performance only for large values of *n*. It was observed that the average value for *n* in FFT, LU, TOMCATV, and RADIX was between 1 and 2. As a result the performance improvements due to message aggregation is not significant in these applications.

In summary, it can be seen that the relative speedup shows reasonable improvement over the original CVM code. Also, the relative speedup improves from single processor configuration to 8 processor configuration. This is due to the fact that the percentage of useful calls increases with increasing number of processors, as shown in Table VIII.

6.2. Inlining Results

The effect of inlining and constant propagation on the execution time of LU is shown in Table IX. To make a fair comparison we also measured the execution time of LU with inlining on the original CVM implementation.

Table IX. Execution Time in Seconds for Inlined LU

	Execution time in seconds						
	CVM		CAS-DSM				
No. of procs.	Original code	Inlined code	AgHoist	AgDiscard	MaxAgHoist	MaxAgDiscard	MaxAgDiscard + MsgAggr
1	18.13	14.47	50.14	49.95	14.38	14.40	14.40
2	24.61	21.00	54.92	54.80	19.02	19.02	19.02
4	17.73	14.96	33.45	33.42	13.80	13.80	13.80
8	13.58	11.18	20.62	20.44	10.56	10.65	10.65

Table X. Relative Speedups for Inlined LU w.r.t. Inlined Original

No. of procs.	Relative speedup w.r.t. inlined original				
	AgHoist	AgDiscard	MaxAgHoist	MaxAgDiscard	MaxAgDiscard + MsgAggr
1	0.31	0.31	1.05	1.05	1.05
2	0.38	0.38	1.16	1.16	1.16
4	0.45	0.45	1.15	1.16	1.16
8	0.55	0.55	1.15	1.15	1.15

It can be seen that inlined original version itself performs slightly better than the original version with no inlining on CVM. The execution times after AgHoist as well as AgDiscard are still comparable, with the corresponding earlier versions, since there is no substantial reduction in the overhead of consistency calls. However after MaxAgHoist the overhead decreases substantially resulting in a performance improvement. Table X reports the relative speedup of LU with inlining. To quote the actual figures, the number of consistency calls in the inlined version reduced from 45 million to 0.2 million. This resulted in an increase in the percentage of useful calls to 4.76% and hence an improvement in execution time.

Fig. 10 summarizes the relative speedup of CAS-DSM due to the different optimizations for 2, 4, and 8 processors.

6.3. Impact of Reducing Holdup Time

In this section we evaluate the performance improvement due to the instrumentation of polling code in the application. In this study we assume that the application is instrumented with MaxAgDiscard optimization, and the `cvm_probe` function is inserted in `CAS_AgHoist`. We present both the relative speedups and reduction in the holdup time (refer to Fig. 2 in Section 2.3) due to the instrumentation of the polling function. It should be noted here that the holdup time reported in this discussion is a result of CVM's design decision to handle message reception through polling. As a consequence, the two-way latency of messages as seen by the application increase in CVM, even though the underlying high performance switch of IBM-SP2 can support a low two-way latency of 75 microseconds. Thus the high values of holdup time reported in this section are essentially due to this rather than due to the interconnection network in the underlying architecture.

In these experiments, the reduction in the overall execution time is a result of both MaxAgDiscard optimization and polling. Likewise, in addition to the consistency check overhead, the overhead due to the polling

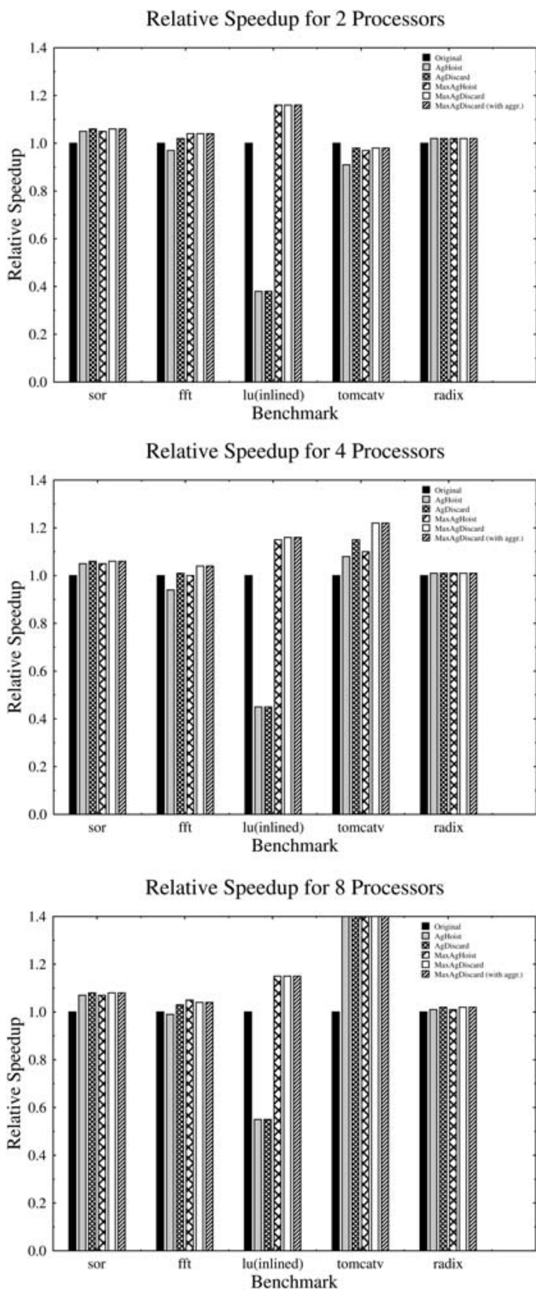


Fig. 10. Relative speedup of CAS-DSM.

Table XI. Relative Speedups with Polling

No. of procs.	Relative speedup of CAS-DSM with polling				
	SOR	FFT	TOMCATV	LU(inlined)	RADIX
2	1.34	1.12	1.23	1.40	1.01
4	1.30	1.21	1.20	1.32	1.01
8	1.22	1.19	1.52	1.17	1.01

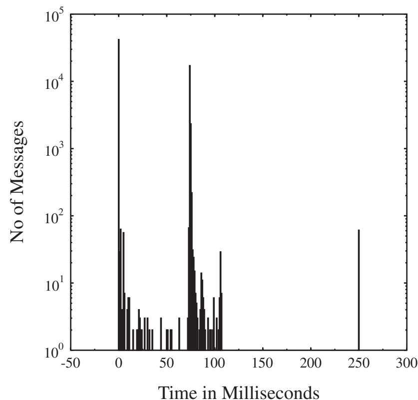
function is also incurred. To determine the overhead associated with polling, we ran a small program which just polls and returns. It was found that each call of the polling function takes 5.83 μ seconds.

The relative speedups for all the benchmarks with polling is shown in Table XI. It can observed that with the instrumentation of polling, the performance of all applications (except RADIX), has improved significantly (12 to 52%). This performance benefit is due to the reduction (through `cvm_probe`) in the holdup time. For illustrative purposes, we report the execution times for SOR with and without the instrumentation of polling in Table XII. For comparison purposes, the execution time of SOR in the original CVM implementation is also presented in the same table. The last column shows the relative speedup of the application after instrumentation with polling function to the execution time of the program in the original CVM implementation. From the last column it is clear that the improvement in the execution time is more than 25% over the original (once the polling function is instrumented).

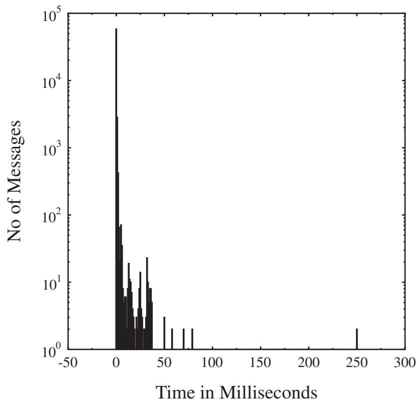
To substantiate our claim on the reduction in holdup time, we plot the histogram of the holdup time experienced by messages in Fig. 11. The x-axis represents the holdup time in milliseconds (ms). On y-axis the number of messages which take x amount of holdup time is plotted.

Table XII. SOR-Execution Time in Seconds

No. of procs.	SOR-Execution Time in Seconds			Relative speedup (with polling)
		Aggregation		
	Original	Without polling	With polling	
2	3086.82	2958.05	2309.11	1.34
4	1607.89	1534.69	1234.03	1.30
8	864.53	809.36	707.83	1.22



(a) Original 2 Procs



(b) Aggregation & Polling 2 Procs

Fig. 11. Holdup time for SOR.

It should be noted that the y axis is in the logarithmic scale. Due to space constraints, we present only the histogram for SOR (2 processors), but it is representative of all benchmarks that we considered. Figures 11(a) and 11(b) show the holdup time respectively for the original code and the instrumented (with MaxAgDiscard and polling) code. In the graphs, we plot the number of messages which experienced less than 1 ms of holdup time at time 0 ms. Similarly, the total number of messages whose holdup times are greater than or equal to 250 ms are plotted at time 250 ms. One can observe a shift in the histogram towards the lower holdup times in the graph shown in Fig. 11(b) compared to that in Fig. 11(a). Specifically, we see that previously, a large number of messages having a holdup time between 50 and 100 ms in Fig. 11(a). After the instrumentation most of the accesses incur a holdup time of less than 50 ms. To quote the actual numbers, if we take the two "peaks" in the graphs occur at 0 ms and at 74 ms in Fig. 11(a) with number of messages equal to 41,795 and 17,095 respectively. On the other hand, in Fig. 11(b) there is only one peak at 0 ms with the value of 58,317 messages. The histograms for other benchmarks also follow a similar trend and hence have not been included in the paper. This substantial reduction in the message holdup time contributes heavily toward the reduction in the execution time or the improvement in the relative speedup.

Why does the average holdup time reduce with polling (`cvm_probe` calls) in CAS_AgHoist function? In the original CVM implementation, `diff` request for updating a page is sent by a process only when it tries to

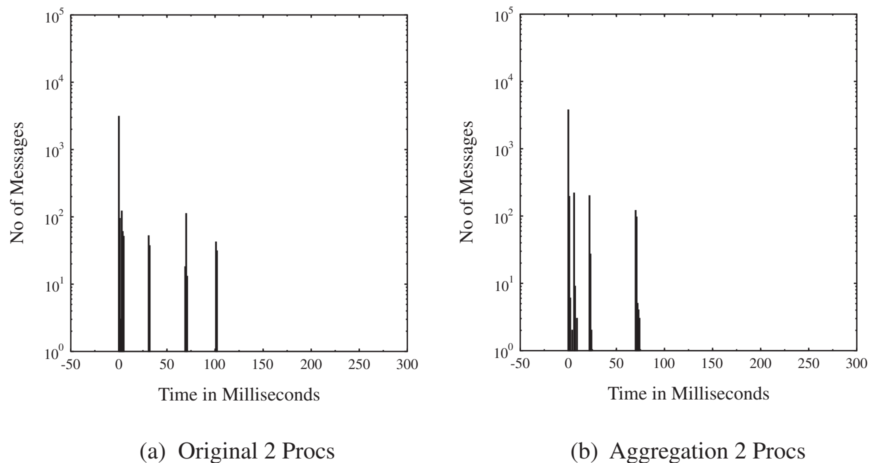
access some location in the shared page that is in a inconsistent state. Further requests for updating different pages are sent by the process at different times depending on when it accesses the shared pages. However, these messages are seen by the receiving processes only when they enter the DSM layer during its execution. As a result the holdup time experienced by asynchronous messages could be quite high in the original CVM implementation. As opposed to this, in our CAS-DSM implementation with polling calls inserted in the instrumented calls, the aggregation and hoisting of `cvm_make_page_consistent` calls enable sending the `diff` request messages for all pages in a clustered fashion by different processes more or less at the same time, often following a barrier synchronization call. Thus when all processes are making the required pages consistent, they also receive messages from other processes, which they see through `cvm_probe` call and service, resulting in smaller average holdup time.

Similar performance improvement due to the reduction in holdup time has also been reported by Keleher,[39] although he refers to holdup time and its effects as "responsiveness" to messages. In his work calls to `cvm_probe` are directly inserted in the application program, whereas in our approach they are automatically included as a part of `cvm_make_page_consistent` calls.

The performance improvement in the benchmarks (SOR, FFT, TOMCATV, LU and RADIX) with the insertion of polling are upto 1.34, 1.21, 1.52, 1.40, and 1.01 respectively. Thus except in RADIX we observe significant improvement in execution time due to the reduction in holdup time. The reason for no performance improvement in RADIX is that even in the original code, the average holdup time is less than one millisecond.

A careful observation of the relative speedups (in Table IV) reveals that the performance of TOMCATV is disproportionately large when compared to the rest of the benchmarks. The reason for this is that the optimizations implicitly reduce the holdup time, and hence the communication overhead. We plotted the graphs showing the holdup time reduction for the original benchmark as well as instrumented one with only AgHoist (but without polling function). The histogram for TOMCATV for 2 processor case is shown in Fig. 12. We observe that with the AgHoist optimization, message that experience a holdup time of 100 ms or more have reduced significantly.

To get a more quantitative view of the reduction in holdup time, we present the *average holdup time ratio* which is the ratio of the average holdup time of the instrumented code (AgHoist instrumentation but without polling) and the average holdup time of the original code (again, without any instrumentation). The average holdup time ratios for all the benchmarks are shown in Table XIII. It can be seen from the table that in



(a) Original 2 Procs (b) Aggregation 2 Procs

Fig. 12. Holdup time for TOMCATV for 2 processor.

all the benchmarks the average holdup time ratio is close to 1, for most of the number of processors, except in the case of TOMCATV. For TOMCATV, the average holdup time ratio ranges from 0.62 to 0.83. This plays a major role in increasing the relative speedup and hence the performance of TOMCATV under AgHoist and other optimizations.

Lastly, we evaluate the effect of increasing the frequency of polling in the original CVM itself. It should be noted that in the original code the polling function in CVM will get executed only when the DSM layer gets the control. Segmentation handler is one of the entry points into the DSM layer from the application program which is executed very often. Hence, we introduced this polling function in the segmentation handler. We refer to this implementation as DSM Poll as the polling is done inside the DSM

Table XIII. Average Holdup Time Ratio

No. of procs.	Average holdup time ratio				
	SOR	FFT	LU	TOMCATV	RADIX
2	1.17	1.41	1.00	0.83	0.75
4	0.97	0.95	0.73	0.69	1.00
8	1.05	0.94	1.00	0.64	1.00

Table XIV. Combined Effect of Instrumentation

No. of procs.	SOR Appln. poll	SOR DSM poll	SOR Combined	FFT Appln. poll	FFT DSM poll	FFT Combined
1	0.67	1.00	0.67	0.99	1.00	1.00
2	1.34	1.00	1.35	1.12	1.02	1.12
4	1.30	1.00	1.31	1.21	1.20	1.21
8	1.22	1.00	1.23	1.19	1.17	1.18

No. of procs.	TOMCATV Appln. poll	TOMCATV DSM poll	TOMCATV Combined	LU Appln. poll	LU DSM poll	LU Combined	RADIX Appln. poll	RADIX DSM poll	RADIX Combined
1	0.74	0.99	0.75	0.89	0.99	0.88	1.01	1.00	1.00
2	1.23	0.95	1.22	1.40	1.05	1.42	1.01	1.00	0.96
4	1.20	0.98	1.22	1.32	1.03	1.32	1.01	1.05	1.00
8	1.52	0.99	1.53	1.17	1.02	1.19	1.00	1.03	1.00

(All values: Relative speedup)

routine. In contrast, in the implementation discussed earlier, the poll function is invoked in the aggregation (`CAS_AgHoist` function). Hence we refer to that as Application Poll. The results of DSM Poll, and Application poll for all the benchmarks are shown in Table XIV. This table also shows the effect of instrumenting the calls both in the application program and in the DSM (referred to as *Combined*).

Except in FFT, DSM polling does not result in any significant improvement in performance. The improvement in FFT could be due to the sharing pattern of the application. Last, in all the applications combined polling, i.e., polling in DSM layer in addition to Application layer does not offer any additional performance improvements over the Application poll. In a few cases, combined polling reduces program performance owing to increased overhead in polling. From the above, we conclude that instrumenting the application code without introducing polling in the DSM layer performs better in general. Also, more analysis should be performed to determine the positions where the polling code should be introduced in the application code.

7. RELATED WORK

Several software DSM implementations have been proposed in the literature.[9–12, 14, 27, 40–43] Of these, a large majority[10–12, 14, 41] use VM support

for detecting shared accesses and maintaining consistency of shared locations while we use compile time support to reduce the reliance on VM. More recently, Keleher proposes a *tape* mechanism that records shared accesses and uses that to predict future access and prefetch them[16] in a software DSM. In Ref. 17, Itzkovitz and Schuster describe a page-based software DSM called Millipage that is based on Multiview, which enables variable sharing granularity. A survey of DSMs can be found in Ref. 44. In the following paragraphs, we compare other DSM implementations which are similar to CAS-DSM.

In Ref. 15, Scales *et al.*, present Shasta, a software DSM that instruments the application executable to insert code before every load and store. This work also describes some optimizations to reduce the overhead due to the instrumentation code. One major advantage of our approach over Shasta is that, more information about the shared accesses is available to our CAS-DSM, since we perform the analysis on application source as opposed to application binary. This helps us to incorporate more optimizations. A major advantage Shasta has over our method is that it provides support for setting the coherence granularity level. In our approach, the granularity is fixed to the page level. Support for varying coherence granularity in Shasta, helps application programmers to take advantage of fine grain granularity to increase the performance of the application. Lastly, as Shasta instruments executables/binaries directly, it is applicable to a larger range of applications, including commercial software where the source code is not available.

Another related work is RT-DSM[13] which also proposes compile time instrumentation. In RT-DSM, the application source is analyzed during compile time to instrument code after a write to a shared location for consistency maintenance. Consistency in RT-DSM is maintained in two phases: A write detection phase and a write collection phase. An important difference between CAS-DSM and RT-DSM is that RT-DSM instruments code after every write. RT-DSM relies on VM-support mechanism for read accesses to shared locations. Whereas in CAS-DSM the instrumentation is for both read and write. Further, RT-DSM supports fine grain sharing for writes while CAS supports page level granularity. A major advantage RT-DSM has over our approach is that it avoids the cost of *diffs and twins* by having a fine granularity of sharing. The main objective of CAS-DSM is to reduce the segmentation violation overheads.

Dwarkadas *et al.*,[45] eliminate synchronization overheads by compile time analysis. Their approach identifies modification to shared data by each processor, that would be required in other processors and *push* the data by inserting this code at appropriate parts in the program. In other words, their analysis will avoid a significant number of segmentation

violations without the overhead of explicit consistency check code. This work also addresses issues regarding communication overhead reduction using message aggregation mechanisms. Further in this work since the modified data is *pushed* to the receiver, it also reduces communication delays. In contrast our motivation was to eliminate the page fault by careful program instrumentation and is based on the *pull* data model. Our approach is restricted to compile-time analysis of possible segmentation violations and insertion of appropriate consistency check calls. Further, our analysis is limited to memory accesses and synchronization constructs within a process, as opposed to across processes as in Ref. 45. However, their work did not study the impact of holdup delays (in processing asynchronous messages) on the execution time.

Lu *et al.*, have proposed a method by which compile time analysis can be used for the reduction of communication overheads for irregular applications.[36] This work concentrates on aggregation of messages into a single message and thus reduce the overhead associated with sending more number of messages. In CAS-DSM, the messages are aggregated under the MaxAgDiscard + MsgAggr optimization. We reduce the waiting time of the messages in CAS-DSM by insertion of the polling function. This reduces the waiting time at the receiver's side. In contrast, their work[36] concentrates on reducing the overhead only at the sender side,

In Ref. 39, Keleher reports the *responsiveness* to asynchronous communication and reducing *notification delays* in distributed systems. In this work he studies the effect of introducing network polls judiciously in the application code and their impact on the performance. This is similar to our approach of instrumenting polling function. In our approach we instrument the polling function within the `CAS_AgHoist` function. Both work report the resulting performance gain due to reduction in holdup time, and the need for extensive analysis.

Our approach to making pages consistent (through `cvm_make_page_consistent`) ahead of time is different from software prefetching approaches.[46] The latter approach attempts to mask the latency by prefetching the pages/blocks that are likely to be accessed in future. These approaches, at least currently, do not take into account the presence of synchronization constructs and the memory consistency model supported. Whereas our approach is tuned to do exactly this. Secondly, while prefetching methods focus on hiding memory latency, our approach focuses on eliminating page fault kernel overhead. Further prefetching methods fetches only a few pages/blocks that are just enough to mask the latency; prefetching more pages/blocks than required may have other effects displacing useful information, especially in the case of hardware DSMs. In CAS-DSM, with MaxAgDiscard optimization, we aggregate and get all

pages that are accessed until the next synchronization point. Lastly, in the present implementation of CAS-DSM, the `cvm_make_page_consistent` and `cvm_make_all_pages_consistent` calls are *blocking*; i.e., execution control waits (polls for responses for `diff` requests) on these calls until the pages are made consistent. Although it may be possible to change this implementation, due to the aggregation of messages (`diff`s for all pages) and the small amount computation that is involved between the `cvm_make_all_pages_consistent` call and the first access to the shared page, we do not expect any significant improvement. As a result, our approach does not mask the latency involved in getting pages consistent.

8. CONCLUSION

This paper presents CAS-DSM, a Compiler Assisted Software DSM. The major goal of CAS-DSM is to reduce the involvement of OS kernel using compiler techniques. For achieving this, CAS-DSM performs a source to source translation of the application code. We made use of SUIF, a compiler tool, for performing the analyses and instrumenting the code. We modified CVM, a publicly available DSM, to provide the enhanced support required for instrumentation. We proposed and implemented various aggressive compile time optimizations to improve the performance of CAS-DSM. These optimizations include aggregation and hoisting of consistency checking code, optimistically discarding some of the instrumented calls, and standard compile time techniques such as inlining and constant propagation. Our methods achieve a performance improvement of upto 10 to 15% for most of the applications compared to the original CVM implementation.

Also, we proposed a method to reduce the overheads involved in the polling-based implementation for receiving asynchronous messages. We concentrate on reducing the waiting time of the messages by decreasing the delay associated in servicing a request. This is also accomplished by compile time analysis and instrumentation of polling code into the application source. Preliminary experiments were carried out in this direction and we were able to get performance improvements as high as 47% in some cases.

Future work could include extending the analysis to across procedures and performing constant propagation inter-procedurally. Selective function inlining can also be included in the our compiler framework. It is also possible to extend our analysis on shared accesses across different processes, which would help reduce unnecessary consistency check calls. Software

prefetching can also be incorporated to improve the existing performance by overlapping computation with communication. From the point of communication overhead reduction, further analysis can be carried out to determine the overheads associated with the inserted polling calls. Last, in order to reduce the effects of hold-up time, the software DSM can be implemented using the multithreaded approaches, where multiple computation and communication threads can be made responsible for computation and communications.

ACKNOWLEDGMENTS

The authors are thankful to the anonymous reviewers for their numerous suggestions which helped to improve the presentation of the paper. The support received from IBM (under IBM's Shared University Research Program), IBM Solutions Research Centre, India, and Tata-IBM, India, are gratefully acknowledged. The work by Manoj was supported by IBM's Shared University Research Program and a research grant from IBM Solutions Research Centre, India and Tata-IBM, India. A preliminary version of the work of Manjunath has appeared in the *Proceedings of the First Workshop on Software Distributed Shared Memory*, held in conjunction with the International Conference on Supercomputing (ICS-99), Rhodes, Greece, June 1999. The work of Govindarajan was done when the first and second authors were at the Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 560 012, India.

REFERENCES

1. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd Ed., Morgan Kaufmann Publishers, San Francisco, CA (1996).
2. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, The Stanford DASH Multiprocessor, *Computer*, **25**(3):63–79 (March 1992).
3. D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi, Distributed-Directory Scheme: Scalable Coherent Interface, *IEEE Comput.*, 74–77 (June 1990).
4. E. Hagersten, A. Landin, and S. Haridi, DDM—a Cache-Only Memory Architecture, *Computer*, **25**(9):44–54 (September 1992).
5. S. Frank, KSR1: High Performance and Ease of Programming, No Longer an Oxymoron, *Proc. of the 5th Ann. ACM Symp. on Parallel Algorithms and Architectures*, Velen, Germany, p. 335 (June 30–July 2, 1993).
6. Sarita V. Adve and Kourosh Gharachorloo, Shared Memory Consistency Models: A Tutorial, *IEEE Comput.*, 66–76 (December 1996).
7. V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing, *IEEE Concurrency: Practice and Experience*, **2**(4):315–339 (December 1990).

8. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press (1994).
9. K. Li. Ivy, A Shared Virtual Memory System for Parallel Computing, *International Conference on Parallel Processing*, pp. 94–101 (1988).
10. C. Amza, A. L. Cox, S. Dwarakdas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, Treadmarks: Shared Memory Computing on Network of Workstations, *IEEE Computer*, 18–28 (February 1996).
11. J. K. Bennett, J. B. Carter, and W. Zwaenepoel, Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence, *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Seattle, WA, pp. 168–176 (March 14–16, 1990).
12. B. N. Bershad and M. J. Zekauskas, Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, Technical report, School of Computer Science, Carnegie Mellon University, Pisstburgh, PA 15213 (1991).
13. M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, Software Write Detection for a Distributed Shared Memory, *The Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 87–100 (November 1994).
14. P. Keleher, *CVM: The Coherent Virtual Machine*, University of Maryland, College Park, MD (July 1997). `http://www.cs.umd.edu/projects/cvm`
15. D. J. Scales, K. Gharachorloo, and C. A. Thekkath, Shasta: A low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, *Proceedings of Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, pp. 174–185 (October 1996).
16. P. Keleher, Tapeworm: High Level Abstractions of Shared Access, *Proc. of the 3rd Symp. on Operating System Design and Implementation*, pp. 201–214 (February 1999).
17. A. Itzkovitz and A. Schuster, Multiview and Millipage: Fine Grain Sharing in Page Based DSMs, *Proc. of the 3rd Symp. on Operating System Design and Implementation*, pp. 215–228 (February 1999).
18. H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, Message Passing Versus Distributed Shared Memory on Networks of Workstations, *Proc. of the Supercomputing Conference SC-95* (December 1995).
19. OpenMP. `http://www.openmp.org`.
20. H. Lu, C. Lu, and W. Zwaenepoel, OpenMP on networks of workstations, *Proc. of the Supercomputing Conference SC-98* (November 1998).
21. K. Kusano, M. Sato, T. Hosomi, and Y. Seo, The Omni OpenMP Compiler on the Distributed Shared Memory of Cenju-4, *Proc. of the Intl. Workshop on OpenMP Applications and Tools, WOMPAT 2001*, West Lafayette, IN, USA (July 2001).
22. S. C. Woo, M. Ohara, E. Torrie, J. Pal Shingh, and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, pp. 24–36 (June 22–24, 1995).
23. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. of the 17th Ann. Intl. Symp. on Computer Architecture*, Seattle, WA, pp. 15–26 (May 28–31, 1990).
24. J. R. Goodman, Cache Consistency and Sequential Consistency, Technical Report 1006, Dep. of Comp. Sci., University of Wisconsin, Madison (February 1991).
25. M. Dubois, C. Scheurich, and F. Briggs, Memory Access Buffering in Multiprocessors, *Proc. of the 13th Ann. Intl. Symp. on Computer Architecture*, Tokyo, Japan, pp. 434–442 (June 2–5, 1986).

26. P. Keleher, A. L. Cox, and W. Zwaenepoel, Lazy Release Consistency for Software Distributed Shared Memory, *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture*, Gold Coast, Australia, pp. 13–21 (May 19–21, 1992).
27. K. L. Johnson, M. F. Kaashoek, and D. Wallach, CRL: High-Performance All-Software Distributed Shared Memory, *Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors* (June 1995).
28. K. Thitikamol and P. Keleher, Multi-Threading and Remote Latency in Software dsms, *International Conference on Distributed Computer Systems*, Baltimore, Maryland, USA (May 1997).
29. D. A. Bader and J. JaJa, SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs), *Journal of Parallel and Distributed Computing*, **58**(1):92–108 (1999).
30. H. Han and C-W. Tseng, Compile-Time Synchronization Optimizations for Software DSMs, *Proc. of the International Parallel Processing Symposium* (1998).
31. T. Park and H. Y. Yeom, An Efficient Logging and Recovery Scheme for Lazy Release Consistent Distributed Shared Memory Systems, *Future Generation Computer Systems*, **17**(3):265–278 (2000).
32. Stanford SUIF Compiler Group, *The SUIF Parallelizing Compiler Group*, Technical report, Stanford University (1994).
33. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, Sp2 System Architecture, *IBM Systems Journal*, **34**(2):152–184 (1995).
34. C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker, The sp2 High-Performance Switch, *IBM Systems Journal*, **34**(2):185–204 (1995).
35. Stanford SUIF Compiler Group, *An Overview of the Suif Compiler System*, Technical report, Stanford University (1994).
36. H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel, Compiler and Software Distributed Shared Memory Support for Irregular Applications, *Sixth Symposium on Principles and Practices of Parallel Programming*, pp. 48–56 (1997).
37. G. E. Blelloch, C. E. Lieserson, B. M. Maggs, C. G. Plaxton, and S. J. Smith, and M. Zagha, A Comparison of Sorting Algorithms for the Connection Machine CM-2, *Symposium on Parallel Algorithms and Architectures*, pp. 3–16 (July 1991).
38. S. C. Woo, J. P. Singh, and J. L. Hennessy, *The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors*, Technical report, Stanford University (December 1993).
39. D. Perkovic and P. Keleher, Responsiveness without Interrupts, *Proc. of the 1999 Intl. Conf. on Supercomputing (ICS-99)*, pp. 101–108 (June 1999).
40. I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, Fine-Grain Access Control for Distributed Shared Memory, *Proceedings of Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, pp. 297–306 (October 1996).
41. D. J. Scales and M. S. Lam, *An Efficient Memory Layer for Distributed Memory Machines*, CSL-TR-94-627, Computer Systems Laboratory, Stanford University (1994).
42. S. Ahuja, N. Carriero, and D. Gelernter, Linda and Friends, *Computer*, **19**(8):26–34 (August 1986).
43. D. Yeung, J. Kubiatowitcz, and A. Agarwal, MGS: A Multigrain Shared Memory System, *Proceedings of the Twenty-Third International Symposium on Computer Architecture*, pp. 44–55 (May 1996).
44. J. Protić, M. Tomašević, and V. Milutinović, Distributed Shared Memory: Concepts and Systems, *IEEE Parallel and Distributed Technology*, 63–79 (1996).

45. S. Dwarkdas, A. L. Cox, and W. Zwaenepoel, An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System, *Proc. of Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 186–197, San Jose, CA (October 1996).
46. T. Mowry and A. Gupta, Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors, *Journal of Parallel and Distributed Computing* **12**:87–106 (1991).