# An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System

Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel
Department of Computer Science
Rice University
e-mail: {sandhya, alc, willy}@cs.rice.edu

**Abstract**

On a distributed memory machine, hand-coded message passing leads to the most efficient execution, but it is difficult to use. Parallelizing compilers can approach the performance of hand-coded message passing by translating data-parallel programs into message passing programs, but efficient execution is limited to those programs for which precise analysis can be carried out. Shared memory is easier to program than message passing and its domain is not constrained by the limitations of parallelizing compilers, but it lags in performance. Our goal is to close that performance gap while retaining the benefits of shared memory. In other words, our goal is (1) to make shared memory as efficient as message passing, whether hand-coded or compiler-generated, (2) to retain its ease of programming, and (3) to retain the broader class of applications it supports.

To this end we have designed and implemented an integrated compile-time and run-time software DSM system. The programming model remains identical to the original pure run-time DSM system. No user intervention is required to obtain the benefits of our system. The compiler computes data access patterns for the individual processors. It then performs a source-to-source transformation, inserting in the program calls to inform the run-time system of the computed data access patterns. The run-time system uses this information to aggregate communication, to aggregate data and synchronization into a single message, to eliminate consistency overhead, and to replace global synchronization with point-to-point synchronization wherever possible.

We extended the Parascope programming environment to perform the required analysis, and we augmented the TreadMarks run-time DSM library to take advantage of the analysis. We used six Fortran programs to assess the performance benefits: Jacobi, 3D-FFT, Integer Sort, Shallow, Gauss, and Modified Gramm-Schmidt, each with two different data set sizes. The experiments were run on an 8-node IBM SP/2 using user-space communication. Compiler optimization in conjunction with the augmented run-time system achieves substantial execution time improvements in comparison to the base TreadMarks, ranging from 4% to 59% on 8 processors. Relative to message passing implementations of the same applications, the compile-time run-time system is 0-29% slower than message passing, while the base run-time system is 5-212% slower. For the five programs that XHPF could parallelize (all except IS), the execution times achieved by the compiler optimized shared memory programs are within 9% of XHPF.

## 1 Introduction

A shared memory programming model for a distributed memory machine can be implemented either solely by run-time methods (e.g., [21]) or solely by compile-time methods (e.g., [14]). Distributed shared memory (DSM) run-time libraries dynamically detect shared memory accesses, and send messages accordingly to implement consistency. Compilers use static analysis of the shared memory access patterns to generate a message passing program. Compile-time systems offer better performance for programs with regular access patterns that allow precise analysis, because they avoid much of the overhead encountered by a run-time system. The class of programs that allow this precise analysis is, however, limited. Run-time systems do not suffer from similar limitations.

In this paper, we demonstrate a combined compile-time run-time approach that

1. provides the same efficiency as a pure compile-time approach for regular programs, and
2. retains the same efficiency as pure run-time systems for programs that defy precise compiler analysis.

In this combined compile-time run-time system, the run-time library retains its original role of detecting shared memory accesses and sending messages, if necessary, to maintain consistency. The compiler, however, serves a function very different from its function in a pure compiler-based approach. In particular, when its analysis is successful, our compiler does *not* generate a message passing program. Instead, it inserts in the source program, calls to (an augmented version of) the shared memory run-time library. Roughly speaking, these calls inform the run-time library of future shared memory accesses, obviating the need for run-time detection, avoiding on-demand remote data fetches, and allowing for aggregation of several remote data fetches into a single message exchange. If the compiler analysis fails, the program is passed on without modification, and executed with the run-time library as in a pure run-time system. If the compiler is partially successful, for instance, it can analyze some phases or some data structures in a program but not others, then those phases or data structures for which analysis succeeds benefit from optimized execution. In summary, in the combined system, the run-time library remains the basic vehicle for implementing shared memory, while the compiler performs optimization rather than implementation.

Our compiler starts from explicitly parallel shared memory programs written for lazy release consistency (LRC) [17]. We use regular section analysis [13] to determine the shared data access patterns between synchronization statements. The resulting regular section descriptors (RSDs) are used to identify opportunities for communication aggregation, consistency overhead elimination, merging synchronization and data messages, and replacing global with point-to-point synchronization. The paper presents the following contributions:

1. An experimental evaluation of the benefit of compiler support to improve the performance of DSM, including a comparison of both optimized and unoptimized DSM to hand-coded and compiler-generated message passing.

2. An experimental evaluation of the contributions of the individual optimizations to the overall performance improvement.

3. A comparison of the performance of different run-time strategies for taking advantage of the data access patterns provided by the compiler, in particular, synchronous vs. asynchronous data fetching, and combining aggregation with synchronization.

We extended the Parascope parallel programming environment [18] to analyze and transform explicitly parallel programs. We also extended the interface to the TreadMarks run-time DSM system [2] to take advantage of the compiler analysis. We have measured the performance of these techniques on an 8-node IBM SP/2 for six Fortran applications: Jacobi, 3D-FFT, Integer Sort, Shallow, Gauss, and Modified Gramm-Schmidt, with two data sets for each application. This selection includes applications that can be expressed in a data parallel language and parallelized efficiently by a pure compile-time approach. This does not undo our case for compiler support for explicitly parallel DSM programs. Explicit parallelism provides a more general programming model than data parallel programs, allowing the expression of applications that would be difficult or impossible to express in a data parallel language. For this more general model to be viable, however, it must provide performance competitive with that of parallelizing compilers for data parallel programs. This is exactly one of the areas in which explicitly parallel DSM programs have been lagging [22]. Our claim is that compiler support for explicitly parallel DSM programs can close this performance gap for data parallel programs, while the underlying DSM system still retains the advantage of being able to support a wider class of applications.

Compiler optimization in conjunction with the augmented run-time system achieves substantial execution time improvements in comparison to the base run-time system, ranging from 4% to 59% on 8 processors. Relative to message passing implementations of the same applications, the base run-time system is 5-212% slower, while the compile-time run-time system is only 0-29% slower. For the five programs that XHPF could parallelize, the execution times achieved by the compiler optimized shared memory programs are within 9% of XHPF. IS was not amenable to parallelization by XHPF because of an indirect access to the main array, and represents an example where partial compiler analysis is beneficial.

On our platform and up to 8 processors, communication aggregation and consistency elimination were the most effective optimizations, in that order. Combining data with synchronization operations is useful when the data is small enough in size such that it can be piggy-backed on the synchronization, or when data can be broadcast at a barrier. With large data sizes, it is better to delay the data fetch until after the synchronization operation, because there is no longer any significant reduction in the number of messages and there is some processor overhead involved in combining data aggregation and synchronization (see Section 3.3). Asynchronous data fetching improved performance more than synchronous fetching.

The methods described in this paper generalize to software DSM systems other than TreadMarks. Every software DSM system must contend with the same issues of message cost, read latency, false sharing, and consistency maintenance. The methods for dealing with these issues may differ in other systems, and the relative values of the improvements obtained by compiler support may differ as well, but the methods remain applicable.

The outline of the rest of this paper is as follows. Section 2 motivates the approach using a comparison of the performance of shared memory and message passing. Section 3 describes the augmented run-time interface. Section 4 presents the compiler analysis used to generate calls to the augmented run-time. Section 5 describes the experimental environment in which the measurements were made. Section 6 presents the performance results. Finally, we survey related work in Section 7 and conclude in Section 8.

## 2 Motivation

DSM provides a shared memory abstraction on distributed memory machines. We focus here on explicitly parallel systems that provide a load-store interface to memory, and that do not require annotations (e.g. [6]) or access to shared memory through object methods (e.g. [5]). Various techniques have been used to optimize the performance of such DSM systems. To make this discussion specific, we describe the techniques used in TreadMarks [2].

TreadMarks uses lazy release consistency [17] to reduce communication and consistency overhead. Lazy release consistency delays consistency-related communication until the time of an *acquire* synchronization operation [10]. In TreadMarks, which uses locks and barriers for synchronization, an *acquire* corresponds to a lock acquisition or to a departure from a barrier. At that time, the acquiring processor is informed by *write notices* of modifications to shared pages. TreadMarks uses an invalidate protocol: the write notices cause the corresponding pages to be invalidated, resulting in a page fault at the time of access. The write notices inform the faulting processor whom it needs to communicate with to get the necessary modifications to the page. TreadMarks uses a multiple-writer protocol, retrieving *diffs* [8] at the time of an access miss rather than whole pages. Diffs are produced by the TreadMarks write detection mechanism. Initially, a page is write-protected. When a processor first writes to the page, it incurs a protection violation and TreadMarks makes a *twin*, a copy of the unmodified page. When the modifications to a page are requested by a remote processor, the twin is compared to the modified copy to create a diff containing the changes. This diff is transmitted to the faulting processor and merged into its copy of the page. In addition to reducing communication, multiple writer protocols have the benefit of reducing false sharing overheads by allowing multiple concurrent writers [8].

Recent studies (e.g., [22]) have shown that, for relatively coarse-grained applications, software DSM provides good performance, although there still remains a sizable gap between the performance of DSM and message passing for some applications. In particular, in a comparison of PVM and TreadMarks on a network of workstations [22], a number of issues were identified as contributing to the performance gap between TreadMarks and PVM: absence of bulk data transfer, separation of synchronization and data movement, consistency overhead, and false sharing. The thesis of this work is that for many applications these shortcomings can be overcome by adding compiler analysis. We focus here on the first three performance issues; false sharing is not directly addressed in this paper. Compiler transformations to reduce false sharing in shared memory programs are discussed elsewhere [12, 16].

We illustrate the performance differences between message passing and DSM with a simple example, the Jacobi program. Jacobi is an iterative method for solving partial differential equations, with nearest-neighbor averaging as the main computation. A shared memory version of a parallel Jacobi appears in Figure 1. To simplify the discussion, we assume that there is no false sharing, i.e., boundary columns start on page boundaries and their length is a multiple of the page size (Our methods work in the presence of false sharing. This simplification is for explanatory purposes only). Processes arrive at `Barrier(2)` at the end of each iteration, resulting in $2(n-1)$ messages with $n$ processors. At the departure from the barrier, pages containing elements of the boundary columns are invalidated on the neighboring processors. When a processor accesses a page in one of its neighbor's boundary columns in the first half of the next iteration, it takes a page fault, which causes TreadMarks to fetch a diff from its neighbor. With $m$ pages in a boundary column, the result is $4m(n-1)$ messages. In addition, there are another $2(n-1)$ messages at `Barrier(1)` that ends the first half of the iteration. Finally, there is consistency overhead for write detection during the second half of the iteration, including memory protection operations, memory protection violations, twinning and diffing. In a message passing version of Jacobi, at the end of an iteration, each processor sends two messages: one to each of its neighbors containing the boundary column to be used by that neighbor in the next iteration. It waits to receive the boundary columns from its neighbors, and proceeds with the next iteration. The result is only $2(n-1)$ messages per iteration for the message passing program. A parallelizing compiler can achieve the same performance for Jacobi.

Compiler analysis and transformation can substantially reduce the number of messages and the consistency overhead for the Jacobi DSM program. First, by intersecting the sections of data written by individual processors before `Barrier(2)` and read afterwards, the compiler recognizes that `Barrier(2)` can be replaced by a `Push` point-to-point message exchange between neighboring processors. The `Push` eliminates barrier overhead and pushes the data rather than pulling it. Second, the compiler can determine that during the second half of the iteration a processor writes *all* elements of the pages in its assigned section of the array. It inserts a `Validate` for that section with a `WRITE_ALL` argument, which causes the run-time *not* to make twins and diffs

```
do k = 1,100
    do j = begin,end
        do i= 2,M-1
            a(i,j) = 0.25 *
            (b(i-1,j)+b(i+1,j)+b(i,j-1)+b(i,j+1))
        enddo
    enddo
    call Barrier(1)
    do j = begin,end
        do i= 1,M
            b(i,j) = a(i,j)
        enddo
    enddo
    call Barrier(2)
enddo
```

Figure 1: Pseudo-code for the TreadMarks Jacobi program: The variables *begin* and *end* are used to partition the work among the processors, with each processor working on a different partition of the shared array *b*.

```
do k = 1,100
    do j = begin,end
        do i= 2,M-1
            a(i,j) = 0.25 *
            (b(i-1,j)+b(i+1,j)+b(i,j-1)+b(i,j+1))
        enddo
    enddo
    call Barrier(1)
    call Validate(b[1,M:begin,end], WRITE_ALL);
    do j = begin,end
        do i= 1,M
            b(i,j) = a(i,j)
        enddo
    enddo
    call Push(b[1,M:begin(p)-1,end(p)+1],
              b[1,M:begin(p),end(p)])
enddo
```

Figure 2: Pseudo-code for the transformed Jacobi program: A `Validate` has been inserted, and `Barrier(2)` has been replaced by `Push`. In the arguments to `Push`, the dependence of `begin` and `end` on the processor number `p` has been made explicit.

for these pages, eliminating consistency overhead. Figure 2 shows the transformed program. While not all overhead is eliminated, the reduction is nonetheless substantial.

The next sections generalize the ideas outlined in this example. We describe in detail the augmented run-time interface, the compiler analysis, and the resulting source-to-source transformations.

## 3 Augmented Run-Time System

The run-time system was augmented in order to take advantage of program access pattern information provided by the compiler. Section 3.1 and Figure 3 describe the compiler interface. Section 3.2 and Figure 4 describe the underlying communication and consistency primitives.

### 3.1 Interface

There are two primary interfaces between the compiler and the run-time system: `Validate` and `Push`. The compiler passes describes the data accessed to the run-time system in

```
/* N is the number of processors *
/* P is the processor id */

Validate( section, access_type )
{
    case ( access_type ) of

    /* preserves consistency */

    READ:           Fetch_diffs(section); Apply_diffs(section);                         Write_protect(section)
    WRITE:          Fetch_diffs(section); Apply_diffs(section); Create_twins(section); Write_enable(section)
    READ&WRITE:     Fetch_diffs(section); Apply_diffs(section); Create_twins(section); Write_enable(section)

    /* does not preserve consistency - compiler analysis must be exact */

    WRITE_ALL:                                                                          Write_enable(section)
    READ&WRITE_ALL: Fetch_diffs(section); Apply_diffs(section);                         Write_enable(section)
}

Validate_w_sync( section, access_type )
{
    /* Uses Fetch_diffs_w_sync instead of Fetch_diffs.  Otherwise, is identical to Validate. */
}

/* does not preserve consistency - compiler analysis must be exact */

Push( r_section[0..N-1], w_section[0..N-1] )
{
    for all processors i != P
        if( r_section[i] intersect w_section[P] != empty )
            send( r_section[i] intersect w_section[P] ) to i

    for all processors i != P
        if( w_section[i] intersect r_section[P] != empty )
            receive from i
}
```

Figure 3: Augmented run-time interface

```
/* Point-to-point communication primitives */

Fetch_diffs( Section )
{
    for all pages in Section
        determine the set of write notices without diffs

    send request for diffs by write notice
}

Fetch_diffs_w_sync( Section )
{
    for all pages in Section
        determine the current timestamp for that page

    at synchronization time
        send request for diffs by timestamp
            on synchronization request
}

Apply_diffs( Section )
{
    receive diffs

    apply diffs to pages in Section
}
```

```
/* Consistency primitives */

Create_twins( Section )
{
    for all pages in Section
        create a twin
}

Write_enable( Section )
{
    for all pages in Section
        insert page into dirty list
        enable write access for the page
}

Write_protect( Section )
{
    for all pages in Section
        disable write access for the page
}
```

Figure 4: Run-time communication and consistency primitives

the form of sections, or so called *regular* sections [13] (for a precise definition, see Section 4)

### 3.1.1 Validate

Validate and its variant, Validate_w_sync, take two parameters: a section and the access pattern, access_type, to that section. access_type is one of READ, WRITE, READ&WRITE, WRITE_ALL, or READ&WRITE_ALL. In essence, the first three access types enable the compiler to reduce execution overhead by bypassing, but *not* disabling, the page-fault based consistency mechanisms. For all three access patterns, the run-time system fetches the diffs to update the pages and applies them. For READ, it write-protects the page, whereas, for WRITE and READ&WRITE, it makes a twin and enables write access to the page.

In contrast, the last two access types *disable* the consistency mechanisms. WRITE_ALL indicates that the entire section is written before it is read. Consequently, the run-time system need not make the pages within the section consistent. In other words, it can avoid fetching the diffs to update the pages. Furthermore, since the entire contents of every page will be overwritten, it need not twin or diff any of the pages. Finally, READ&WRITE_ALL indicates that the entire section is written, but at least part of the section is read before it is written. Therefore, while the run-time system must fetch the data to update the pages, it need not twin or diff the pages.

The only distinguishing feature of the Validate_w_sync variant is that it piggy-backs the request for diffs on the next synchronization operation.

### 3.1.2 Push

Push is used to replace a barrier and to send data to a processor in advance of when it is needed. The arguments to Push are the sections of data that are written by individual processors before the barrier and read after the barrier. Push on processor $P$ computes the intersection of the sections written by $P$ with those that will be read by the other processors and sends the data in the intersection to the corresponding processor. $P$ then computes the intersection of the sections written by other processors with the sections that will be read by $P$, and posts a receive for that data. Push guarantees consistency only for the sections of data received through the Push. The rest of the shared address space may be inconsistent until the next barrier. Push can be used only if the compiler has determined with certainty that the program does not read the regions of shared data left inconsistent. This directive provides the capabilities of a message passing interface within a shared memory environment. The run-time system ensures that if a global synchronization separates the current phase from the rest of the program, all data is made consistent on all processors after that global synchronization. Unlike Validate, Push receives data in place; it does not first receive the incoming message in diff space and then apply the diff.

## 3.2 Run-Time Primitives

### 3.2.1 Communication Primitives

Three primitives are used by the higher-level compiler interface routines to implement communication. The first two, Fetch_diffs and Fetch_diffs_w_sync, each take a section as their argument, and direct the run-time to fetch the modified data in that section. Several sections can be fetched at the same time. In a Fetch_diffs_w_sync the fetch request is piggy-backed on the next synchronization request. In the case of a lock acquire, the requested data is piggy-backed on the response. The third primitive, Apply_diffs, enables the processor to wait for the completion of a Fetch_diffs or a Fetch_diffs_w_sync, and applies the diffs.

Fetch_diffs first converts the section arguments to a list of pages. For all of those pages, Fetch_diffs then finds the set of write notices whose timestamps dominate the timestamp of the local copy of the page. Next, it determines for which write notices in that set it does not have the corresponding diffs, and requests those diffs from the appropriate processors. Each of those processors returns all the requested diffs in a single response message.

Since a Fetch_diffs_w_sync is sent before synchronization completes, the processor is not aware of all modifications to shared data, because it has not yet received write notices for the latest intervals. Hence, the sections given as arguments to the Fetch_diffs_w_sync call are sent along with the synchronization request. This message is sent to the last releaser in the case of a lock or to all other processors in the case of a barrier (by piggy-backing the information on the barrier arrival message to the master, and then forwarding it on the barrier departure messages from the master). The processor also includes the current vector timestamps [17] for the pages in the sections requested to allow other processors to determine what diffs it has and has not seen. These other processors then determine what diffs to communicate to the acquirer. Diffs for the pages in the section with write notices that dominate the page's timestamp are aggregated into a single message and sent to the acquirer. Only the diffs present locally are sent. Other diffs cause an access miss on the acquirer and are faulted in.

Fetch_diffs_w_sync results in additional overhead compared to Fetch_diffs, especially at a barrier, since each processor must examine potentially large address ranges that it did not necessarily modify. Whether to perform data aggregation with or after synchronization is therefore dependent not only on the ability to analyze the code and move the fetch call up to the synchronization point, but also on whether the savings in messages compensate for the additional run-time overhead.

When used with a barrier, Fetch_diffs_w_sync uses broadcast if the processor can determine that it sends the same data to all other processors.

### 3.2.2 Consistency Primitives

Three primitives are used by the higher-level compiler interface routines to implement consistency. All of the primitives take a section as their sole argument. The first, Create_twins, makes a twin of every page within the section. The last two primitives, Write_enable and Write_protect, enable or disable write access to every page within the section. In addition, Write_enable places all of the pages within the section on the processor's dirty page list.

### 3.2.3 Synchronous Vs. Asynchronous Communication

Validate, Validate_w_sync, and Push may be performed either synchronously or asynchronously. Figure 3 only shows the synchronous interface. The asynchronous version of Validate only calls Fetch_diffs. The processor continues executing until the next page fault. At that time, the remainder of the synchronous Validate is executed in the page fault handler. The asynchronous versions of Validate_w_sync and Push work similarly. Asynchronous communication is likely to outperform synchronous communication if there is some computation between the fetch and the first access to the shared data.

### 3.3 Implementation and Limitations

The implementation of the interface was done in conjunction with TreadMarks [2]. Although in the above we specified the parameters to the augmented run-time system calls as sections, this is done for ease of explanation only. To reduce run-time overhead, in the actual implementation these section parameters are translated by the compiler into a set of contiguous address ranges. Furthermore, our implementation currently supports only the synchronous version of `Push`.

## 4 Compiler Analysis

In our analysis we deal with explicitly parallel programs written for an LRC memory model. This observation considerably simplifies the analysis. With *lazy* release consistency, consistency is enforced only at an acquire. For instance, for an invalidate protocol as used in TreadMarks, all invalidations happen at the time of an acquire. As a result, any data item that is accessed after an acquire $a_1$ but before the next acquire $a_2$ can be fetched immediately after $a_1$. Such a fetch always returns the correct value, and never gets invalidated before it is accessed. Also, since we are fetching into memory (and not into cache), there is no issue of replacement of fetched data, as with non-binding prefetching. Our compiler analysis therefore focuses on regions of code between two consecutive synchronization statements. It determines the set of accesses made in such a region, and inserts a `Validate` for the corresponding data immediately after the first acquire. In practice, limitations of the analysis may restrict the extent to which we can implement this general principle. For instance, the presence of conditional statements or — in the absence of interprocedural analysis — procedure calls may limit the region of code for which we can analyze the shared memory access patterns. The corresponding `Validate` is then inserted at the beginning of this region. The algorithms used by the compiler are detailed below.

### 4.1 Access Analysis

In the following, let $V$ be the set of shared variables, let $S$ be the set of all synchronization operations in the program, and let $F$ be the set of "possible fetch points", the locations in the program where a `Validate` or a `Push` may be inserted. $F$ includes the set $S$, but in addition includes conditional statements, and, in the absence of interprocedural analysis, procedure calls.

Access analysis generates a summary of shared data accesses associated with each element of $F$, and the type of such accesses. Our main tool is regular section analysis [13]. Regular section descriptors (RSDs) are used as the representation to concisely provide information about array accesses in a loop nest. The RSDs represent the accessed data as linear expressions of the upper and lower loop bounds along each dimension, and include stride information.

For each statement $p$ in the program,

1. Determine the set $F_{prec}(p)$ of all possible fetch points in the program that directly precede the statement. This is done by traversing the control flow graph to determine all the possible control flow directions that contain a fetch point that could precede the statement. Determine the set $S_{succ}(p)$ of all synchronization points in the program that directly succeed the statement.
2. For each statement $f$ in the set $F_{prec}(p)$,
   (a) Determine the location of the outermost loop that encloses $p$ but not $f$ or any member of the set $S_{succ}(p)$.
   (b) Construct a section for each definition or reference in $p$ to a variable in $V$. Associate a {`read`} or {`write`} tag with the section depending on the access type.
   (c) Perform a union of the resulting section, including the tag, with the other sections that have already been generated for $f$. A union of the tags {`read`} and {`write`} is {`read, write`}.
   (d) Determine the reaching definitions for each reference to a variable in $V$. If this definition occurs after the fetch point but before the use, add the attribute `write-first` to the tag. A section that is written but never read will always acquire the tag {`write, write-first`}.

### 4.2 Transformations

For each element $f$ of $F$

1. If $f$ is a barrier, determine the set $F_{prec}(f)$ of elements of $F$ that immediately precede $f$, and the set $F_{succ}(f)$ of elements of $F$ that immediately succeed $f$.
2. If $F_{prec}(f)$ contains one and only one barrier, $F_{succ}(f)$ is non-empty and contains only barriers, the sections associated with $F_{prec}(f)$ and $f$ are exact representations of the data accessed, and the sections associated with $F_{prec}(f)$ contain write accesses, then

   - replace $f$ with a `Push`, passing as arguments, the read sections of $f$, and the write sections of $F_{prec}(f)$ in terms of processor identifiers.

   else

   - If the section is exact, tagged as {`read, write`} but not {`read, write, write-first`}, and refers to a contiguous range of addresses, then insert a `Validate` at $f$ with the section and access type `READ_WRITE_ALL`. If the section is exact, the tag contains the attribute `write-first`, and refers to a contiguous range of addresses, insert a `Validate` at $f$ with the section and access type `WRITE_ALL`.

   else

   - If $f$ is a synchronization statement, then insert a `Validate_w_sync`, specifying the sections and the access type, just before $f$. Although it is always correct to insert a `Validate_w_sync` under these conditions, we will see in Section 6 that it is sometimes better to insert a `Validate` after $f$.

   else

   - If the section is not unknown (compiler could not analyze the access pattern), insert a `Validate` at $f$, specifying the section and the access type.

### 4.3 Examples

We illustrate the analysis with the Jacobi example in Figure 1. In this example, $V$ is the array b, and both $F$ and $S$ contain the barriers 1 and 2, which we will denote b1 and b2.

For statement $p_1$, the assignment to a(i,j), $F_{prec}(p_1)$ contains b2, and $S_{succ}(p_1)$ contains b1. For each reference to the array b in the righthand side, a section with a {read} tag is constructed, of the form $[1, M-2 : begin, end]$, $[3, M : begin, end]$, $[2, M-1 : begin-1, end-1]$, and $[2, M-1 : begin+1, end+1]$ respectively. Each of these sections is added to the union of the sections for b2, resulting in a final section with a {read} tag and of the form $[1, M : begin-1, end+1]$. For $p_2$, the assignment to b(i,j), $F_{prec}(p_2)$ contains b1, and $F_{succ}(p_2)$ contains b2. The assignment to the array causes a section with a {write} tag of the form $[1, M : begin, end]$ to be constructed.

Going next to the transformation phase, for b2, the conditions for a Push are satisfied, and b2 is replaced by a Push. For b1, we have a {write, write-first} section spanning a contiguous range of addresses, so we can insert a Validate after b1, with an access type of WRITE_ALL.

In the Jacobi example, analysis is precise: the compiler can determine exactly what data is read or written by what processor. In such a case it is also possible for the compiler to directly generate a message passing program. As will be seen in Section 6 the performance of this strategy and ours are very similar. Our methods can, however, also be applied to applications for which analysis cannot be made precise. The IS program from the NAS benchmarks [4], discussed in more detail in Section 6, provides a good example. Here a large sub-array is passed between processors under the control of a lock. Our analysis creates a section for the sub-array and issues a Validate when the lock is acquired, resulting in significant performance improvement. In order for the compiler to generate a message passing program, it would in addition need to determine which processor last held the lock and wrote the data. This information is not available at compile-time. The sequential program also has an indirect access to the main array. Hence, it is difficult to express this program in a data parallel style.

### 4.4 Implementation and Limitations

We implemented the analysis using the Parascope parallelizing environment [18]. We modified the Parascope analysis to work on explicitly parallel programs written for the release consistency model. We added passes to recognize synchronization calls, and to generate data access summaries at each of these calls. Our current framework does not perform inter-procedural analysis. All shared variables must be allocated in a single common block named shared_common. Our regular section analysis handles only indices that are dependent on zero or one induction variable. The loop bounds can themselves be linear functions of variables.

## 5 Experimental Environment and Applications

Our experimental environment is an 8-processor IBM SP/2 running AIX version 3.2.5. Each processor is a thin node with 64 KBytes of data cache and 128 Mbytes of main memory. Interprocessor communication is accomplished over the IBM SP/2 high-performance two-level cross-bar switch. Unless indicated otherwise, all results are for 8-processor runs.

We used six Fortran programs: IS and 3D-FFT from the NAS benchmark suite [4], the Shallow benchmark, and Jacobi, Gauss, and Modified Gramm-Schmidt (MGS), three locally developed benchmarks. For each application, we use two data set sizes to bring out any effects from changing the computation to communication ratio. Table 1 describes the data set sizes and the corresponding uniprocessor execution times. Uniprocessor execution times were obtained by removing all synchronization from the TreadMarks programs; these times were used as the basis for speedup figures reported later in the paper.

| Application | Data set size | Time (secs) |
|---|---|---|
| Jacobi - 4Kx4K | 4096x4096 | 288.3 |
| Jacobi - 1Kx1K | 1024x1024 | 17.7 |
| 3D-FFT - 6x6x6 | $2^6 \times 2^6 \times 2^6$ | 9.5 |
| 3D-FFT - 5x6x5 | $2^5 \times 2^6 \times 2^5$ | 2.3 |
| Shallow - 1Kx1K | 1024x1024 | 74.8 |
| Shallow - 1Kx.5K | 1024x512 | 36.9 |
| IS - 23-19 | $N = 2^{23}, B_{max} = 2^{19}$ | 91.2 |
| IS - 20-15 | $N = 2^{20}, B_{max} = 2^{15}$ | 3.9 |
| Gauss - 2Kx2K | 2048x2048 | 3344.8 |
| Gauss - 1Kx1K | 1024x1024 | 271.5 |
| MGS - 2Kx2K | 2048x2048 | 449.3 |
| MGS - 1Kx1K | 1024x1024 | 56.4 |

Table 1: Applications, data set sizes, and uniprocessor execution times

We present the performance of these applications in four different versions:

1. The base TreadMarks program executing with the base TreadMarks run-time system.
2. The compiler-optimized TreadMarks program executing with the augmented TreadMarks run-time system.
3. A message passing version automatically generated by the Forge XHPF compiler [3] from Applied Parallel Research, Inc. (APR). The results for the XHPF compiler are provided in order to compare performance against a commercial parallelizing compiler for data-parallel programs.
4. A hand-coded PVMe message passing [9] version. The results for PVMe are included to estimate the best possible performance that can be achieved on this platform.

All four systems underneath use IBM's user-level Message Passing Library (MPL). The minimum roundtrip time using send and receive for the smallest possible message is 365 $\mu$seconds, including an interrupt.[1] In TreadMarks, the minimum time to acquire a free lock is 427 $\mu$seconds. The minimum time to perform an 8-processor barrier is 893 $\mu$seconds. Under AIX 3.2.5, the time for both page faults and memory protection operations is a linear function of the page number and the number of pages in use. For instance, the memory protection operation time can vary between 18 and 800 $\mu$seconds with 2000 pages in use.

[1] Although substantially faster round-trip times are possible if interrupts are disabled, interrupts are required to implement lock and page requests in TreadMarks. For XHPF and PVMe interrupts were disabled.

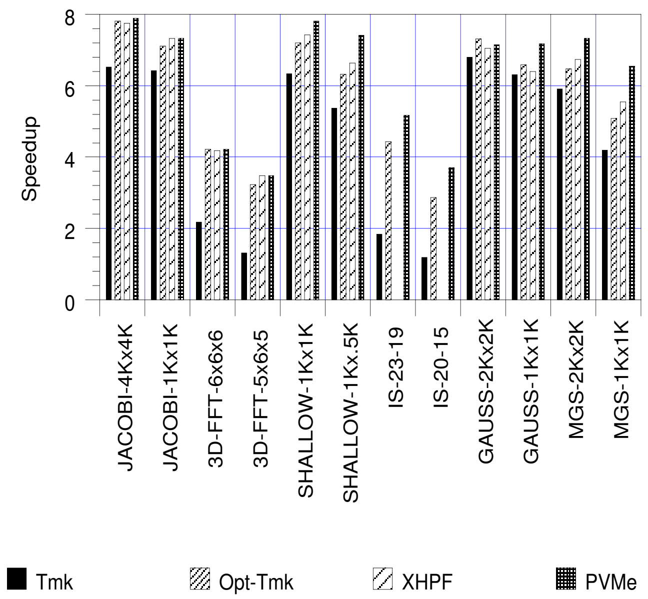


Figure 5: Comparison of TreadMarks, best compiler optimized version of TreadMarks, XHPF, and PVMe. The IS bar is missing for XHPF because it cannot parallelize IS.

| Application | % segv | % msg | % data |
|---|---|---|---|
| Jacobi-4Kx4K | 100.0 | 79.9 | -2312 |
| Jacobi-1Kx1K | 100.0 | 49.7 | -614 |
| 3D-FFT - 6x6x6 | 100.0 | 70.6 | 0.8 |
| 3D-FFT - 5x6x5 | 99.2 | 44.0 | 46.3 |
| Shallow - 1Kx1K | 86.9 | 56.4 | 3.5 |
| Shallow - 1Kx.5K | 85.0 | 47.6 | 3.2 |
| IS - 23-19 | 99.5 | 96.5 | 58.9 |
| IS - 20-15 | 90.1 | 60.7 | 66.3 |
| Gauss - 2Kx2K | 100.0 | 40.0 | 0.1 |
| Gauss - 1Kx1K | 100.0 | 25.0 | 0.4 |
| MGS - 2Kx2K | 100.0 | 53.5 | 0.2 |
| MGS - 1Kx1K | 100.0 | 29.0 | 40.5 |

Table 2: Percentage reduction in page faults ("segv"), messages ("msg"), and data ("data") for the compiler-optimized version of TreadMarks in comparison to the base version of TreadMarks

## 6 Results

### 6.1 Overall Results

Figure 5 shows the speedups achieved for all applications in their four different scenarios: base TreadMarks, compiler-optimized TreadMarks, APR's XHPF, and PVMe. The numbers for the compiler-optimized TreadMarks version reflect the gains achieved by the most sophisticated level of analysis possible for each application and by the best choice of run-time support. There are no entries for IS using XHPF in the figure. XHPF cannot parallelize IS because of an indirect access to the main array in the computation.

Table 2 shows the percentage reduction in the number of page faults, the number of messages, and the amount of data in the compiler-optimized version of TreadMarks compared to the base version.

Compiler optimization achieves substantial execution time improvements in comparison to the base TreadMarks, ranging from 4% to 59%.[2] For programs for which base TreadMarks achieves relatively good speedups (Jacobi, Shallow, Gauss, and MGS), the execution time improvements are moderate, 4% to 16%. For the two programs (IS and 3D-FFT) for which base TreadMarks performs poorly compared to message passing, execution time improvements are quite large, ranging from 48% to 59%. Table 2 shows that the optimized programs have almost all their page faults eliminated for our test programs. The number of messages is reduced from 25-96%. The amount of data transferred differs only in the case where TreadMarks sends multiple diffs per page (MGS and IS) or where false sharing is eliminated by using `Push` to replace barriers (3D-FFT).

For four out of the six programs, the execution times for the compiler optimized shared memory programs are within 0-17% of the PVMe message passing programs. For IS, the difference is larger, 17% or 29%, depending on the data set. This result is a substantial improvement over the base TreadMarks shared memory programs, which lag behind the PVMe execution time by 5-14% in the best case (Gauss) and by 181-212% in the worst case (IS).

For the five programs that XHPF could parallelize, the execution times achieved by the compiler optimized shared memory programs are within 0-9% of XHPF.

### 6.2 Detailed Discussion of Applications and Optimizations

Figure 6 presents a detailed breakdown of the performance of each application under different levels of optimization. XHPF and PVMe results are also presented for comparison. For each of the applications we show speedups under the following scenarios: 1) TreadMarks, 2) communication aggregation, 3) communication aggregation and consistency overhead elimination, 4) if applicable, communication aggregation, consistency overhead elimination, and merging data with synchronization, and 5) if applicable, the `Push` optimization. Communication aggregation reduces the number of messages by combining multiple page transfers into a single message. Consistency overhead elimination reduces the write detection overhead: twinning, diffing, and page protection. The `Push` optimization eliminates synchronization overhead in addition to reducing the number of messages and eliminating consistency overhead.

The **Jacobi** program was described in Section 2. All levels of optimization can be applied to Jacobi. The compiler optimized program shows a 10-16% improvement in execution time over the base TreadMarks and is within 8% of the execution times of the XHPF and PVMe versions.

For the 4096x4096 data set, Jacobi derives most of its improvement from communication aggregation, because of a significant reduction in the number of messages (from 13.8k to 2.7k). There is only a small added benefit from consistency elimination, because the reduced number of memory protection operations, twins and diffs is offset by the increase in the amount of data transmitted (see Table 2). With diffing, only the data whose values change actually get transmitted. In this case, since the internal elements of the matrix are initially zero, a large part of each page

[2] Percentage improvements are calculated by the formula $(base - opt) \div base$.

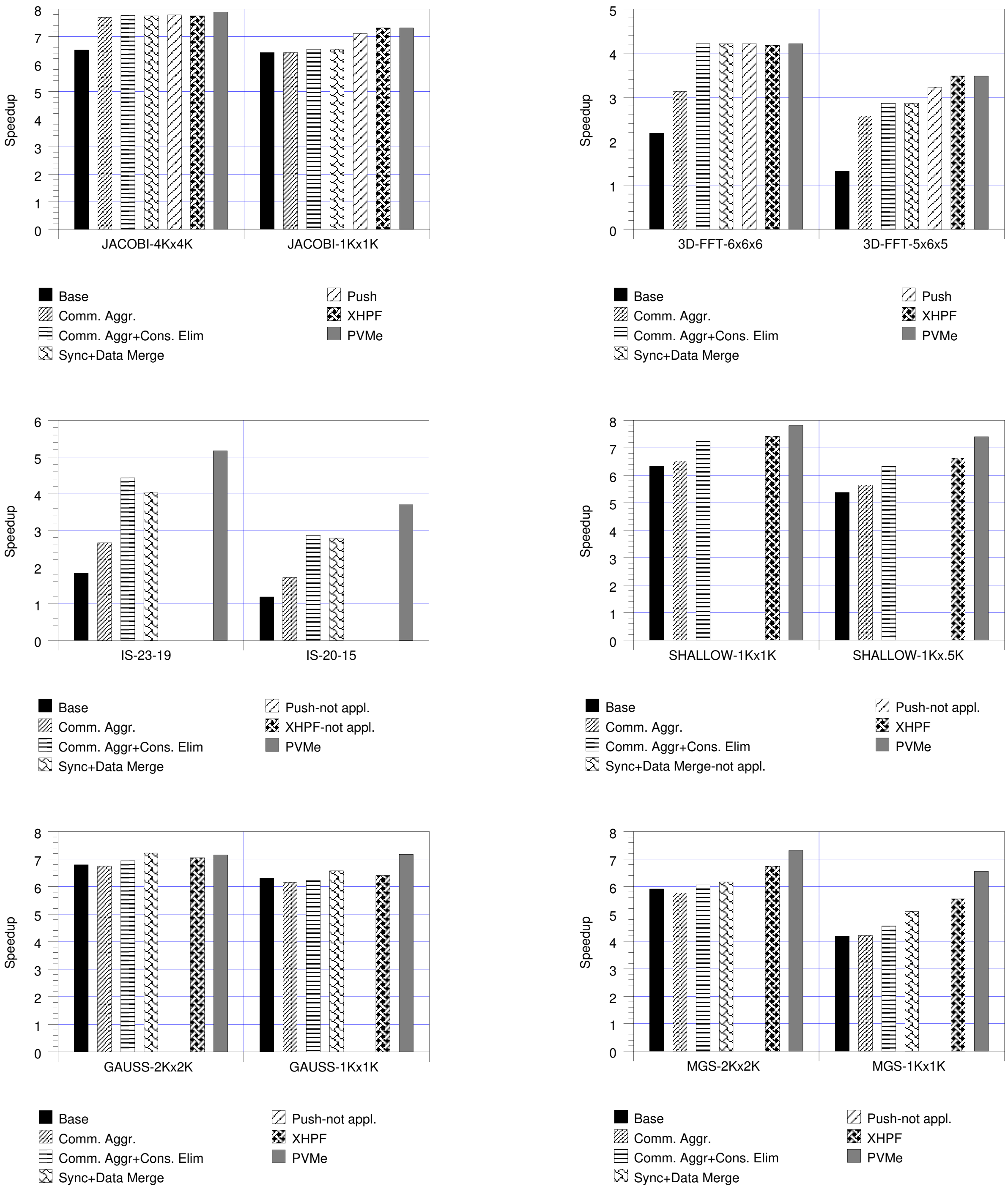


Figure 6: Speedups at 8 processors under varying levels of optimization. The IS speedup is missing for XHPF because it cannot parallelize IS. The other missing speedups are because our compiler is unable 1) to merge synchronization and data movement for Shallow or 2) to replace barriers with a `Push` for Gauss, MGS, IS, and Shallow.

remains unmodified, and hence the diffs are small relative to the page size. There is no gain from merging data with synchronization. The reduction in the number of messages is small, and offset by the extra overhead of each processor determining whether it has modified any of the requested pages. Similarly, there is little gain from replacing the barrier with a `Push`, since the barrier synchronization is only a small part of the total communication overhead.

The results for the 1024x1024 data set differ in two ways. First, the communication aggregation does not improve execution time, because the boundary rows are exactly one page. Second, there is a gain from using `Push`. With a smaller data set, the cost of the barrier becomes proportionally higher, and hence its elimination results in some improvement in running time (10%).

**3D-FFT** is the three-dimensional Fast Fourier Transform program from the NAS suite [4]. 3D-FFT numerically solves a certain partial differential equation using three dimensional forward and inverse FFTs. The phases of the program are separated by barriers, with a transpose between some of the phases to reduce the array traversal cost. Each processor is assigned a contiguous section of the array. The transpose thus causes the producer-consumer communication at the barrier.

All five levels of analysis were applicable and performed. The compiler optimized shared memory program shows a 48-59% improvement in execution time over the base TreadMarks and is within 0-8% of the PVMe and XHPF execution times.

For the large $2^6 \times 2^6 \times 2^6$ data set, large and similar sized gains result from communication aggregation and consistency overhead elimination. Communication aggregation reduces the number of messages from 13.3k to 3.9k. Consistency overhead elimination eliminates all page faults, all protection operations, and all twinning and diffing (see Table 2). Combining synchronization and data transfer and replacing the barrier by a `Push` do not result in additional gains, because the main bottleneck for 3D-FFT is the large amount of data transferred.

For the small $2^5 \times 2^6 \times 2^5$ data size, the results differ in two ways. First, the gains from communication aggregation are significantly larger than those stemming from consistency overhead elimination. For this data set, each contiguous piece of data spans less than a single page. Hence, there is little reduction in the number of memory protection operations due to consistency overhead elimination. There are still gains because of page fault, twin, and diff overhead reduction. Second, the `Push` eliminates some false sharing (reducing the data transfer from 12 to 6 MBytes), resulting in a 11% gain compared to the execution time of the version with merging synchronization and data but with the barrier instead of a `Push`.

**Integer Sort (IS)** is another program from the NAS benchmark suite [4]. IS ranks an unsorted sequence of $N$ keys. The *rank* of a key is the index value $i$ that the key would have if the sequence of keys were sorted. All the keys are integers in the range $[0, B_{max}]$ and the method used is bucket sort. IS divides the keys evenly among the processors. At first, processors count the number of occurrences of each key in their private buckets. In the next phase, the values in the private buckets are summed up using locks to acquire exclusive access to the shared buckets. The shared buckets are divided into as many sections as there are processors. Access to each section is controlled by a lock. The processors access the sections in a staggered manner. The data is migratory in this phase. In the final phase, separated by a barrier to ensure that all the processors are done with updating the shared buckets, the processors read the sums in the shared buckets and rank the keys in their own memory.

Communication aggregation, consistency elimination, and synchronization and data merging were applied to IS. The barrier could not be replaced by a `Push` because the compiler cannot statically determine from where the data is going to come (in other words, which processor held the lock last). The compiler optimized TreadMarks program shows a 55-57% improvement in execution time over the base TreadMarks. Execution time is still 17-29% worse than that of PVMe because the PVMe version pipelines the data transfer to the next processor. XHPF could not parallelize this program because of an indirect access to the main array.

There are no qualitative differences in the results for the two data sets. Both communication aggregation and consistency overhead elimination result in substantial improvements. The gains from consistency overhead elimination are more significant than communication aggregation since consistency elimination also results in reduced data transfer (745 Mbytes to 299 Mbytes). The compiler generates a `Validate` with type `READ&WRITE_ALL`. As a result, no twins or diffs are made. In contrast, TreadMarks suffers from a *diff accumulation* phenomenon [22]. In TreadMarks, if a processor incurs an access miss on a page, it is sent all the diffs created by processors who have modified the data. In IS, the shared array is modified by all processors, causing many diffs to be sent to the faulting processor, even though all of the diffs overlap completely. Merging synchronization with data leads to a worsening of the performance. The increased run-time overhead resulting from going through a large page list outweighs the benefits of a reduction in number of messages (see Section 3.3).

**Shallow** is the shallow water benchmark from the National Center for Atmospheric Research. This code solves difference equations on a two dimensional grid for the purpose of weather prediction. Parallelization is done in bands, with sharing only across the edges. Barriers are used to synchronize the processors between phases.

Only communication aggregation and consistency elimination are performed for this program. Combining synchronization and data transfer or replacing the barriers with `Push` calls would require interprocedural analysis, which our implementation does not support. The compiler optimized program shows a 12-15% improvement in execution time over the base TreadMarks and is within 3-5% and 8-17% of the execution times of the XHPF and PVMe versions, respectively.

There are no qualitative differences in the results for the two data sets. Shallow is in many ways similar to Jacobi, with a small improvement resulting from communication aggregation. The improvement from consistency overhead elimination, is, however, larger for Shallow than for Jacobi. The reason is that the larger number of pages used by Shallow makes page faults and memory protection operations more expensive.

**Gauss** implements Gaussian elimination with partial pivoting to solve a set of linear equations. Parallelization is done in a cyclic manner in order to improve load balance. At every iteration, one processor determines the pivot row, and assigns its row number to a shared variable. The other processors read this shared variable as well as the column containing the pivot element. Logically, both the row number and the column are broadcast. Barriers are used to synchronize the processors between iterations.

Communication aggregation, consistency elimination, and synchronization and data merging were applied. The compiler optimized program shows a 4-6% improvement in execution time over the base TreadMarks, a 2-3% improvement over XHPF execution times, and is within -1-9% of the execution times of the PVMe version.

There are no qualitative differences in the results for the two data sets. In contrast to the other programs, synchronization and data merging was the most effective because it allows the data to be broadcast. The data can be broadcast to all other processors at the time of the barrier, since all processors require exactly the same data, and are aware of who the last producer was.

**Modified Gramm-Schmidt (MGS)** computes an orthonormal basis for a set of N-dimensional vectors. At each iteration $i$, the algorithm normalizes the $i$th vector, and makes all vectors $j > i$ orthogonal to vector $i$. Parallelization is done in a cyclic manner in order to reduce communication while balancing the load in each iteration. The communication and synchronization pattern is similar to that in Gauss.

Once again, communication aggregation, consistency elimination, and synchronization and data merging were applied. The compiler optimized program shows a 4-18% improvement in execution time over the base TreadMarks and is within 9% and 19-29% of the execution times of the XHPF and PVMe versions, respectively. Both XHPF and the optimized TreadMarks suffer some loss in performance relative to PVMe because of the strided access pattern, which results in extra overhead at run-time.

There are no qualitative differences in the results for the two data sets. The use of a broadcast in merging data communication with the barrier was the most effective optimization, resulting in a 2-10% improvement over the base TreadMarks.

### 6.3 Synchronous vs. Asynchronous Data Fetching

Figure 7 shows the speedups achieved by synchronous vs. asynchronous data fetching for the large data sets. Asynchronous data fetching dominates synchronous data fetching in almost all cases. Virtually all applications benefit from the overlap of communication and computation, more so than they are hurt by the additional memory protection operations. Given that the memory management operations on the IBM SP/2 are relatively slow, we expect this result to hold a fortiori on most other architectures.

### 6.4 Summary

We conclude that for the programs and environment used in this study,

1. Communication aggregation and consistency elimination always improve performance, in some cases by a substantial amount.

2. Merging data and synchronization leads to a significant improvement, only if the data is small and can be sent in the same message as the synchronization, or if the data can be broadcast.

3. Gains resulting from the use of a `Push` to replace a barrier are minor. Replacing barriers with a `Push`, in addition, suffers from the problem that a barrier is needed later to restore release consistency. Most of the gain of the `Push` is due to avoiding false sharing. The use of a `Push` may, however, be more beneficial at larger numbers of processors, since the overhead of global synchronization and consistency increases.

4. Asynchronous data fetching gets better performance than synchronous data fetching.

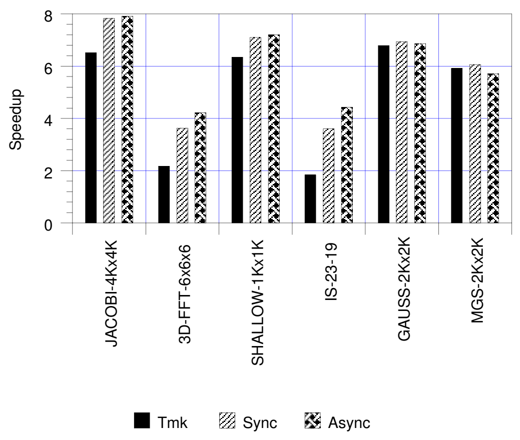


Figure 7: Synchronous vs. asynchronous data fetching

We therefore conclude that a suitable general purpose strategy is to do communication aggregation and consistency elimination. Merging data with synchronization and replacing barriers with a `Push` are only useful when the data to be communicated is small relative to the overhead of the barrier messages or when there is false sharing.

## 7 Related Work

Research and commercial compilers for distributed memory machines have to date targeted the underlying message passing layer directly [3, 14]. Careful optimization to minimize data movement by improving locality is performed, and data and work is distributed according to the owner computes rule. At the other end, compilers such as SUIF [1] parallelize directly to shared memory and do not take advantage of bulk transfer capabilities. This work attempts to bridge the gap by providing the flexibility of shared memory while taking advantage of bulk transfer.

Several recent proposals for hardware shared memory machines include a message passing subsystem designed in part to allow applications to take advantage of bulk data transfer [19, 20]. Woo et al. [24] evaluate one such design in the context of the Flash system. There are many differences between their work and ours. The Flash bulk data transfer consists of multiple *cache lines* as opposed to multiple *pages* in our work, and the latencies used in the Flash simulation are much smaller than in our implementation. Finally, while Woo et al. focus on establishing the magnitude of the performance benefits of bulk data transfer, we have explored in addition ways for the compiler to automate the use of the bulk data transfer facility.

Mowry et al. [23] discuss the design and evaluation of a compiler algorithm for prefetching. Their algorithm concentrates on improving the performance of cache-based systems and issues prefetch requests for data that are likely to incur a cache miss. Porterfield et al. [7] present an algorithm for inserting prefetches one loop iteration ahead. Gornish, Granston, and Veidenbaum [11] present an algorithm for determining the earliest time when it is safe to prefetch shared data. Our work differs in the granularity of information required, and takes advantage of the software-based consistency maintenance. Our optimizations perform aggregation with a view to exploiting the explicit synchronization in relaxed consistency models.

Jeremiassen et al. [15] present a static algorithm for computing per-process memory references to shared data in coarse-grained parallel programs. This information is then used to determine cross-process memory references in order to direct the type of coherence protocol to use in a bus-based architecture. We use a similar analysis in terms of processor identifiers in order to replace a barriers with a `Push`.

## 8 Conclusion

We have experimentally demonstrated that the addition of compiler-driven communication aggregation and consistency overhead elimination improves the performance of software DSM on distributed memory multiprocessors. Our compiler computes data access summaries using regular section analysis and feeds that information to the release-consistent TreadMarks DSM system. Improvements in execution time range from 4 to 59% on an 8-processor IBM SP/2 in comparison to the base run-time system for the applications analyzed. Among the various run-time options, asynchronous communication aggregation coupled with consistency overhead elimination works best. Combining synchronization and data transfer, and replacing a barrier by a `Push`, result in gains if the data transfer overhead is small in comparison to the synchronization overhead, and if it reduces false sharing or allows a broadcast.

A combined compile-time run-time system of this nature retains the ease of programming of shared memory. No additional user input is required. It also supports the same wide class of programs as shared memory. The combination of static prediction of shared memory accesses by the compiler with dynamic detection of accesses by the run-time allows the combined system to approach the performance of hand-coded or compiler-generated message passing. It does so without incurring the programming difficulties of message passing or the limitations on automatic parallelization of data-parallel programs for message passing targets.

### Acknowledgements

We would like to thank Gene Wagenbreth of Applied Parallel Research, Inc. and Honghui Lu of Rice University for developing the XHPF versions of the benchmarks.

This work is supported in part by the National Science Foundation under Grants CCR-9410457, BIR-9408503, CCR-9457770, CCR-9502500, CCR-9521735, CDA-9502791, and MIP-9521386, by the Texas TATP program under Grant 003604-017, and by grants from IBM Corporation and from Tech-Sym, Inc.

### References

[1] S. P. Amarasinghe et al. The SUIF compiler for scalable parallel machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[2] C. Amza et al. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, February 1996.

[3] Applied Parallel Research. *FORGE High Performance Fortran User's Guide*, version 2.0.

[4] D. Bailey et al. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.

[5] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE-TSE*, June 1992.

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

[7] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of ASPLOS-4*, April 1991.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM TOCS*, August 1995.

[9] G.A. Geist and V.S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, June 1992.

[10] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of ISCA-17*, May 1990.

[11] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of ICS-90*, 1990.

[12] E. Granston and H. Wijshoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of ICS-93*, July 1993.

[13] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE-TPDS*, July 1991.

[14] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *CACM*, August 1992.

[15] T.E. Jeremiassen and S. Eggers. Computing per-process summary side-effect information. In U. Banerjee et al., editors, *Fifth Workshop on Languages and Compilers for Parallelism*, August 1992.

[16] T.E. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of PPoPP-95*, July 1995.

[17] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of ISCA-19*, May 1992.

[18] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice and Experience*, October 1993.

[19] D. Kranz et al. Integrating message-passing and shared-memory: Early experience. In *Proceedings of PPoPP-93*, May 1993.

[20] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Proceedings of ISCA-21*, April 1994.

[21] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, November 1989.

[22] H. Lu et al. Message passing versus distributed shared memory on networks of workstations. In *Proceedings SuperComputing '95*, December 1995.

[23] T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of ASPLOS-5*, October 1992.

[24] S.C. Woo, J.P. Singh, and J.L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of ASPLOS-6*, October 1994.