Optimizing Communication in HPF Programs for Fine-Grain Distributed Shared Memory

Satish Chandra
James R. Larus

Technical Report #1332

January 1997

Optimizing Communication in HPF programs for Fine-Grain Distributed Shared Memory

Satish Chandra and James R. Larus

Computer Sciences Department
University of Wisconsin—Madison
1210 W. Dayton Street
Madison, WI 53706 USA
{chandra, larus}@cs.wisc.edu

December 1996

Abstract. The coherence mechanisms underlying most shared-memory systems work equally well—or poorly—for regular and irregular programs. In both cases, data accesses go through a general-purpose protocol, which provides the only communication primitive available to software. However, in parts of many codes, compile-time information about data accesses could be used to transfer data more efficiently than a standard coherence protocol, if the underlying system provided suitable primitives. This paper demonstrates that for HPF programs running on fine-grain distributed shared memory [32], cooperation between the compiler and the coherence protocol can produce significant performance gains (upto 26%), while retaining the versatility and portability of coherent shared memory. This paper also describes the design choices in our implementation and reports detailed experimental results.

1 Introduction

Parallel programs running on shared-memory multiprocessors often spend considerable time waiting for the underlying memory system. This overhead is particularly acute when a program exhibits false sharing or poor locality of reference. Another significant source of overhead is the fixed cache-coherence protocol used in a multiprocessor—for example, a single piece of data may require four or more messages to move from the writer to a reader processor [10]. A recent study by Torrie et al. [33] showed that such memory system overhead could comprise more than 35% of the execution time on a suite of compiler-parallelized programs. Moreover, conventional multiprocessors offer a fixed coherence protocol as the only general-purpose method of communicating values between processors. In such systems, software cannot avoid the coherence overhead *per se*, although several latency reducing and tolerating techniques have been proposed [13,24]. On systems that implement shared memory over a cluster of workstations [1,32], the higher communication latencies make coherence overhead even more taxing.

Because of these problems, parallel shared-memory systems are starting to offer ways of optimizing data transfer that range from new memory operations to the options of bypassing a coherence protocol. All-hardware systems are limited to simpler operations, so a multiprocessor can provide memory system operations such as poststore [30], co-operative prefetch [15], self-invalidate [15] and store-and-forward [19] that can be used by a programmer or a compiler to optimize performance. Systems that implement coherence in software, such as Typhoon [27], Flash [20] and Shasta [31], and most page-based systems [1,5], can go further and offer message-passing like communication primitives or alternate protocols to an application. Emerging commercial shared-memory multiprocessors, such as STiNG [22], now offer at least some support for data transfer mechanisms beyond a fixed coherence protocol.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, an NSF NYI Award CCR-9357779, NSF Grant MIP-9625558, DOE Grant DE-FG02-93ER25176, and donations from Sun Microsystems, and The Portland Group. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U S. Government.

This paper describes a compiler-directed approach to exploiting such communication mechanisms in the context of HPF programs running on fine-grain distributed shared memory (DSM). The analyses developed for compiling for message-passing machines [4,29,34] can also be used to identify opportunities for employing more efficient value transfer mechanisms in shared-memory systems. Our compiler uses these analyses to identify cache blocks for which short-cuts to the default coherence actions can be safely made. It then inserts run-time calls that explicitly manage communication on these blocks. In program phases in which the necessary preconditions on data accesses cannot be shown at compile-time, our system allows the default protocol take over by first bringing all the blocks in a globally consistent state. A key contribution of this work is the development of a contract between the compiler and the coherence protocol, so they can co-operate and reduce data-transfer costs where static analysis permits it. Our techniques are suited for fine-grain shared-memory systems because they bypass the default coherence mechanisms on fairly small chunks of data. Page-based systems need a somewhat different approach (though the compiler analysis involved is quite similar). We review two compiler-directed approaches for page-based systems [9,17] in related work (Section 2).

We modified a commercial HPF compiler—the Portland Group's *pghpf*—to generate simple shared memory code [6], and furthermore, to perform the communication analysis necessary to insert run-time calls to the coherence protocol. Our target is the Tempest system [16], which implements distributed shared memory at the granularity of cache blocks (e.g. 32-128 bytes) and allows user-code to provide its own coherence protocol or use an existing one. We performed our experiments on a Tempest implementation running on an 8-node cluster of SparcStation20 workstations connected by Myricom's Myrinet. Our results show that optimizations reduce the overall execution times by 3%-26% on a suite of 6 applications; most of the optimized execution times are competitive with *pghpf*'s message-passing backend. As previously demonstrated [6], the simpler shared-memory approach lets a wider class of HPF programs run far more efficiently than message passing—those benefits remain undiminished in the current approach.

The rest of the paper is organized as follows. Section 2 discusses related work in the area of modifying the coherence behavior by compiler techniques. Section 3 provides the relevant background material on coherence protocols and discusses the opportunities for coherence optimizations in the context of fine-grain distributed shared memory. Section 4 describe our compilation model and the interface to the coherence protocol. Section 5 presents our experimental setup. Section 6 presents detailed experimental evaluation of our technique. Section 7 concludes the paper.

2 Related Work

Our techniques closely resemble those used by Dwarkadas et al. [9] to optimize coherence overhead on a page-based DSM system. Both works exploit the well-developed techniques for static analysis on arrays to make data transfer cheaper by reducing coherence overhead and performing sender-initiated transfers. However, Dwarkadas's techniques are targeted at page based DSMs (TreadMarks[1]). In TreadMarks, detecting and preparing for writes (twinning) are high overhead operations. Not surprisingly, their most profitable optimization is to prevent write-faults from occurring at run-time. They found that sender-initiated transfer in their system accrues very minor benefit. In fine-grain DSM systems, such as ours, the cost of gaining write-ownership is far smaller, so we optimize for the delays incurred in true sharing, i.e. a cross processor dependence that exercises the coherence protocol.

Furthermore, Dwarkadas's compiler analysis requires and uses only localized access information between barriers. This is appropriate when dealing with a phase in an explicitly parallel program, but it does not exploit global access information that might be available. By contrast, this work is applicable to compiler-parallelized codes, where the loop distribution is usually fixed for a program—work distribution is determined at compile-time, typically following the owner-computes rule [29] (although our scheme can handle other computation distributions as well). This leads to two important observations: first, only blocks that contain array elements involved in producer-consumer relationships[1] pay coherence overheads—the other "inner" cache

[1] A memory location is written by one processor (producer) and subsequently read by another processor (consumer).

blocks are brought once and for ever in to the local memory and pay no further overhead (assuming no cache-replacement, as is the case in our system). Dwarkadas's scheme must validate all the pages accessed by a processor before each parallel loop, even if the "remote" data accessed in the loop is small. Second, we exploit the availability of access permissions to reduce the overhead of coherence-manipulating calls to a protocol. While these calls may not comprise a large overhead on a page-based system, fine-grain systems have many more transfer units (cache blocks), so reducing run-time calls can improve performance.

Keleher and Tseng [17] reduce miss time on their page-based DSM by flushing modified pages to prospective readers, as opposed to having readers fetch them on a miss. They also use compiler analysis to mark the set of pages that are communicated in a stable pattern, although the actual detection of the producers and consumers is left to the run-time system (they observe this could be done by the compiler). In contrast with Dwarkadas's and our work, their work does not relax the system's coherence and permit the compiler to take control of all accesses to shared data. We use precise compiler analysis (where applicable) to identity the set of blocks that need to be propagated from a writer to the readers and bypass coherence where appropriate.

Another class of machines *require* software involvement to maintain coherence [25]. In software cache coherence, the difficult problems are to identify exactly how long to keep a value in the programmable cache and when to fetch a new value. Several researchers have studied compiler techniques for this problem [8,7]. However, these schemes must be conservative and work for all data accesses—rather than the selected ones we focus on—so they often suffer from excessive invalidations and re-fetch. In our case, the default protocol would automatically fetch the newest value at a read: we only seek to make this transfer more efficient.

Larus et al. [21] have used compiler-controlled incoherence for efficiently implementing the semantics of a data-parallel language. They implement fine-grain copy-on-write on especially marked blocks in the coherence protocol instead of paying high copying overhead necessitated by conservative static analysis. By contrast, we use compiler-controlled incoherence to make statically identifiable communication more efficient.

3 Coherence Overhead and Optimization

A typical coherence protocol provides two functions. First, it satisfies a load by shipping the current value of the requested address to the faulting node, in a manner transparent to the program. Second, it maintains currency of values by either invalidating, or updating existing cached copies when new values are written. Various details, such as how soon a reader can expect to see a newly written value, vary according to the consistency model implemented by a coherence protocol. However, coherence protocols generally cannot make any further assumptions on a program's memory accesses.

Consider a producer-consumer relation in which a block of data is written by processor p and subsequently read by processor q. For the moment, assume that the block's size is one word, equal to the size of the value being transmitted. Furthermore, assume an invalidation-based protocol (general update-based protocols have analogous problems, but the details are omitted for brevity). Figure 1(a) shows the coherence actions that this transfer can expect to undergo. Note that at all times, the directory (a data structure at the *home* node) must be aware of the state of the block, because any other processor r is free to join the fray and access the shared memory location. Figure 1(a) also serves to illustrate the working of our default protocol[1] at a high-level; the figure caption gives additional detail on messages.

Typically, in many programs, p and q perform this communication repeatedly, say in a time-step loop. Assume that the memory location in question is neither read, nor written by any other processor. In this situation, p can directly send its value to q, provided such a primitive is available. While p and q engage in direct communication, and no other processor accesses the same location, the directory need not track the current state of the block, or the up-to-date contents of the block. See Figure 1(b). At the end of this phase, processors p and q must make their state consistent with the directory information.

[1] We try to hide some of the write latency by implementing a release-consistent memory model [11].

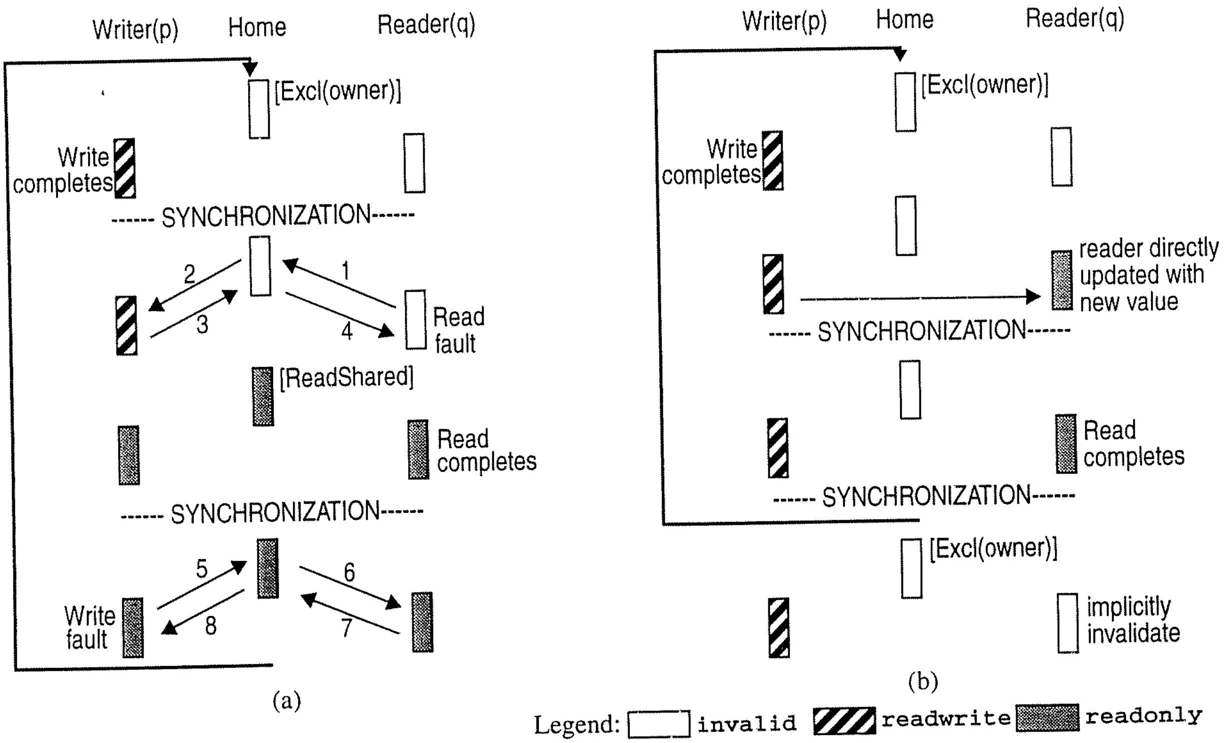


Figure 1: (a) Default coherence scheme. Notice the number of messages required to transfer one block. The messages are as follows: 1. read-request 2. put-data-request 3. put-data-response 4. read-response 5. write-request 6. invalidation 7. acknowledgement, and 8. write-grant. (b) Direct update message to the reader. Note the reduction in the messages. A final step is required to ensure coherence.

The preceding discussion is somewhat over-simplified for explanatory purposes; we describe in detail the contract between our compiler and the protocol in Section 4.2. For the present, we briefly mention the Tempest features that allow us to implement explicit coherence control. Tempest allows a coherence protocol to be written as user-level code, by exposing the following primitives. (1) Locally mapping remote pages in the shared segment, so the program can use global virtual addresses. (2) Fine-grain access control, which allows user code to put `invalid`, `readonly` or `readwrite` protection on individual blocks. An access to `invalid` block, or write access to a `readonly` block invokes a user-specified fault handler. (3) Messaging at fine-granularity, i.e., sending active messages with optionally a block worth of data in them. Ordinary protocols that implement transparent shared memory, use all three mechanisms to implement the desired consistency model. Our compiler inserts calls that directly invoke the fine-grain access control and messaging primitives, in order to bypass the default coherence protocol in certain cases.

In principle, we should be able to completely bypass the default coherence when perfect information on the readers and writers of memory locations is available in a particular application. However, some practical problems must be addressed. First, any DSM system maintains coherence on finite-sized blocks that are typically larger than one word. Hence, a particular block can hold a range of array indices, even straddling dimensions, e.g. $a(513,1)$ and $a(1,2)$ could be in the same block for a 513x513 array. In such a case, it is not always possible to draw conclusions about the usage of all elements that compose a cache block. For example, the compiler may assert that it can orchestrate all communication for $a(513,1)$. However, it cannot ask the runtime to manipulate access permissions to the block on which $a(513,1)$ resides, because it may not have any guarantees about the accesses to $a(1,2)$. Note that this problem does not manifest itself for compilers targeting message-passing machines, as they synthesize a global space from private memories; there is no notion of two array locations lying on the same shared-memory block. Second, the compiler and the coherence protocol must share a simple representation that summarizes blocks under compiler control; explicit listing of all such blocks can introduce high runtime overhead. This summary can also introduce imprecision. In the subsequent sections, we will discuss the design choices we made to address these issues.

4 Compiler-Orchestrated Incoherence

The compiler has three tasks in our approach. First, it performs analysis to calculate the read and write sets for arrays accessed in parallel loops. Second, it generates calls to the coherence protocol, so certain data transfers can take place more efficiently—this forms the core of our technique. Third, it can optimize the placement of these calls.

4.1 Access Information

We need to determine the sections of arrays that are read and written in each parallel loop, so we can find the communication involved in executing the loop. This computation takes into account the distribution of the arrays (as specified in the user directives) and the computation distribution of the parallel loop. The data distribution determines the *owner* relation: an array element $a(i,j)$ is owned by a processor p, if it *logically* resides on the processor p. It is important to bear in mind that $a(i,j)$ may have its home on any processor in the system, since the home is not necessarily the same as owner. We currently make a simplifying assumption on data distributions: only the last dimension of a global array is distributed (either blockwise or cyclically) on a linear arrangement of processors. The computation distribution is usually owner-computes, but this is not a restriction in our scheme. The compiler can use the programmer-supplied `INDEPENDENT` directive to divide a loop in any fashion, e.g. blockwise by loop-index, or according to an `ON HOME` directive.

Based on the data and computation distributions, the compiler computes access sets. For each distributed array accessed in a parallel loop, it computes the *non-owner-read* and *non-owner-write* sets by taking the set difference of the array sections that a processor reads or writes and the array sections it owns. If these sets are null, no values need be transmitted. In a fine-grain DSM, the only communication that would then take place is due to false sharing caused by multiword blocks—in most cases, this occurs at the boundary elements of array columns[1]. We do not optimize for these boundary cases. By contrast, in page-based DSMs, due to large size of the coherence unit, significant communication can be expected due to false sharing, and it is important to optimize for it [9].

In our implementation, we use Maryland's *Omega* library [26] to compute these sets. Although the kind of sections we optimize could be represented by traditional regular section descriptors (RSD) [3], the Omega library enabled us to avoid the significant implementation effort required to build a robust RSD package. In addition, the Omega library handles symbolic variables that appear in our test cases, as well as keep access sets parametric with respect to processor number, without extra effort. To obtain succinct representations of the blocks that we may wish to take under compiler control, we address our optimizations only to those array sections that can be shown, at compile-time, to form contiguous virtual addresses. We also allow two-dimensional sections, represented as contiguous ranges separated by a fixed stride. Omega library can be directed to generate C code as a static representation describing such sets: at run-time we invoke these code-fragments with the values of symbolic variables to obtain the bounds of the corresponding access sets.

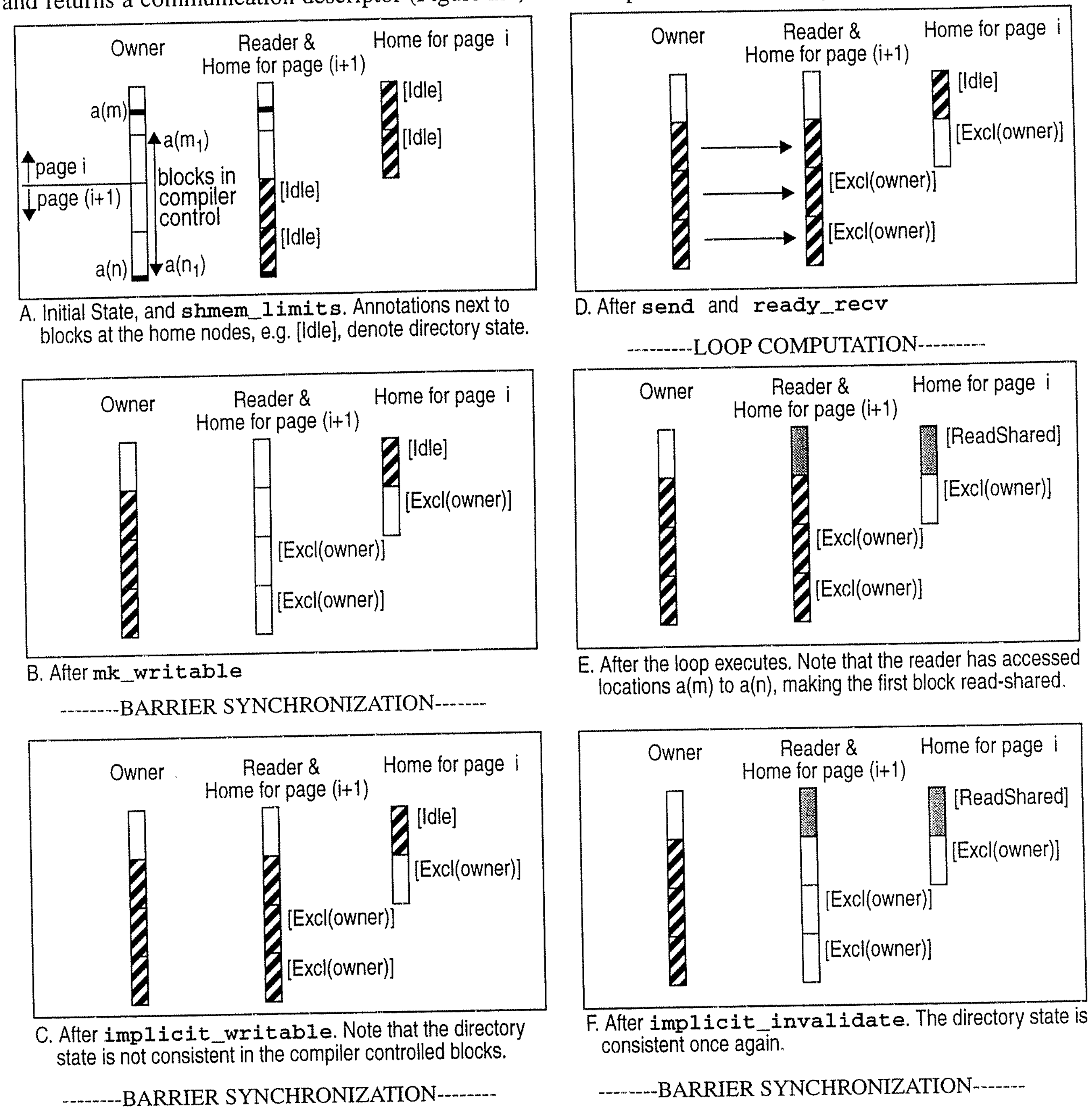
4.2 Overriding the Default Protocol

In this section, we describe the run-time calls generated by our compiler, based on the information collected in the first phase. We first post-process this information to determine contiguous ranges of cache blocks that can be taken under compiler control. Recall from Section 3, that due to multi-word block size, we have to be careful when taking a particular block under compiler control. Therefore if the array section $a(m{:}n)$ is a candidate for optimizations, we select the subset $a(m_1{:}n_1)$, such that $m_1 \geq m$ and $n_1 \leq n$, and $a(m_1)$ and $a(n_1)$ fall within closest fitting block boundaries. For two-dimensional transfers, we have to do this subsetting by iterating over the higher dimension. We take the blocks thus determined under explicit compiler control, leaving the boundary cases to the default protocol—our default protocol is a standard invalidation-based protocol as

[1] It is possible to specify data distributions in which under owner-computes rule there will be significant false-sharing, e.g. (CYCLIC,*) with column major addressing. We do not address that problem here. Anderson et al.[2] present one approach to mitigate that effect.

described in Section 3. These boundary cases could also be optimized by *advisory* primitives, such as self-invalidate and co-operative prefetch [15], and may be a worthwhile optimization where the data set size is small. It is also possible that there are no accesses to the off-section elements residing at the boundary blocks. Compile-time analysis to determine this must make assumptions about starting addresses of arrays and the block size. We have not explored this option yet.

Figure 2 shows the run-time calls that our compiler generates to modify the default protocol behavior for a non-owner read reference. These calls are generated for each non-owner reference in the loop. Like calls for all references are grouped together, so they can share the synchronization requisites between these calls. The first run-time call, **`shmem_limits`**, establishes the restricted limits as described in the previous paragraph, and returns a communication descriptor (Figure 2A). We also pass the values of symbolic variables in these



A. Initial State, and **`shmem_limits`**. Annotations next to blocks at the home nodes, e.g. [Idle], denote directory state.

B. After **`mk_writable`**

--------BARRIER SYNCHRONIZATION-------

C. After **`implicit_writable`**. Note that the directory state is not consistent in the compiler controlled blocks.

--------BARRIER SYNCHRONIZATION--------

D. After **`send`** and **`ready_recv`**

---------LOOP COMPUTATION---------

E. After the loop executes. Note that the reader has accessed locations a(m) to a(n), making the first block read-shared.

F. After **`implicit_invalidate`**. The directory state is consistent once again.

--------BARRIER SYNCHRONIZATION-------

Legend: □ `invalid` ▨ `readwrite` ▒ `readonly`

Figure 2: The run-time calls and their effect on block states.

run-time calls, so they can be used in the analytical expressions generated by the Omega library. The remaining calls and the rationale for the additional synchronization are described in the following paragraphs.

Our goal is to make the non-owner blocks available before a parallel loop, so no access fault occurs while executing the loop. For the moment, ignore non-owner writes. We designate owners to send the relevant blocks to the readers. The senders and the receivers need to make certain preparations before this transfer can take place:

(1) Since the owner is not necessarily the home node, there is no guarantee that at this point in the execution, the owner has a copy of the block it has to send to the potential reader(s). Therefore, all senders must first bring the relevant blocks to readable state in their caches.[1] In anticipation that eventually the senders will write new values, we bring all the relevant blocks in writable state before initiating the transfer. An important side effect of this step is that the directory information for these blocks reflects that the owner has the current (and only) valid copy, thus relieving the actual home, if it is not owner, of carrying a valid copy. We will employ this observation momentarily.

(2) In Tempest, readers require `readwrite` permission to store the incoming data (as is the requirement for any store). In ordinary coherence protocols on Tempest, a read-miss handler makes a block `readwrite` for the purpose of receiving the incoming data, then switches the access permission back to `readonly` after the data has been stored. In compiler-controlled coherence, however, we have established that the blocks being brought over are under explicit control, and the compiler can keep them in whichever state it likes. Therefore, the we direct all the blocks to be received to have `readwrite` access, even though no data resides in them until written in. After the parallel loop has executed, the compiler promises to invalidate such blocks.

There is an ordering requirement between steps 1 and 2. Since a reader may be the home node of a block, step 2 may destroy the only copy in the system. However, step 1 guarantees that the owner processor has the only valid copy of the block: we only need to enforce that step 1 has completed before starting step 2. A barrier synchronization enforces this ordering.

Step 1 is achieved by a `mk_writable` call on an specified range of blocks (see Figure 2B). The protocol interprets this call as if a write fault is incurred for all the blocks in the specified range, except in a pipelined fashion. The second step, after the synchronization, is achieved by an `implicit_writable` call to the protocol (Figure 2C). This call sets the access permissions of all blocks in the specified range to `readwrite`.

Another synchronization after step 2 guarantees that the senders and receivers are both ready for the transfer. The senders ship the relevant blocks over to the receivers by calling a `send` operation on a range of addresses, and destination ids, which the underlying protocol complies by sending the blocks in a specially tagged data message to each recipient. Each receiver posts a `ready_to_recv` call, which holds down a counting semaphore until all the blocks have arrived. As a further optimization, we group contiguous blocks and transfer them in larger payloads than a single block size. This optimization gives us the benefit of using larger block sizes, and is evaluated in Section 6. At the conclusion of this data transfer, our goal of providing non-owner read data in (at least) readable state before the loop is achieved (Figure 2D).

After the parallel loop executes, the directory for a compiler-controlled block believes that the block is in exclusive state at the processor designated as sender (Figure 2E). This information does not reflect the correct state of affairs, because the readers have a writable copy of those blocks as well. With no further information about future data accesses, we need to invalidate the readers. This is achieved via the `implicit_invalidate` call to the protocol (Figure 2F). A final barrier assures that things are consistent again with the information at the directory.

Let us now address the issue of non-owner writes. In this case, the owner has to send the block to the writer, just as in the non-owner read case. The only difference is that at the end of the loop, the writer has to `flush` its changes back to the owner, and implicitly invalidate, so the scenario at the end is that the owner has the only latest (writable) copy of the block, and directory correctly reflects this information.

[1] There is no replacement from this cache—this is software managed remote data in main memory.

4.3 Reducing the Run-time Overhead

There are two types of overheads introduced by the basic scheme described in Section 4.2. First, all run-time calls need not be repeatedly executed. If a subsequent parallel loop has the same computation distribution, the owners already have the blocks in writable condition (assuming no intervening read exercised default coherence actions). Therefore the **`mk_writable`** call can be eliminated for the second loop. Similar arguments hold for the **`implicit_invalidate`** call. Second, this scheme does not take into account the *availability* [14,12] of data. If there is no intervening write to the same non-owner read data between two loops, it need not be re-communicated at the second loop. In fact, the default coherence protocol will not communicate a block twice, unless it is invalidated by a write.

Both these problems naturally fall in the framework of partial redundancy elimination(PRE) [23]. Researchers have already established how to avoid redundant communication (second problem mentioned in the previous paragraph) using variants of PRE [12,14,18]. The first problem, *viz.* the placement of calls, can also be cast as a PRE problem by formulating the availability of access permissions as a flow problem. Unfortunately, we require interprocedural analysis to draw full benefit from this framework, as most of the codes are (justifiably) written in terms of subroutines.

A simple run-time scheme partially addresses the call overhead problem under the following "whole program" assumptions: (1) the computation distribution is strictly owner-computes, (2) the program has no non-owner reads to un-initialized memory locations, and (3) all non-owner reads are under explicit compiler control. Fortunately, all these conditions hold in our current benchmark suite. Indeed, to our knowledge, all compilers targeting message-passing machines work under assumptions 1 and 3. If the preceding assumptions are true, we can eliminate the **`mk_writable`** calls, as the owner nodes are guaranteed to have the blocks in **`writable`** state by the effect of the default protocol. Consequently the barrier after the **`mk_writable`** call can also be eliminated. The **`implicit_invalidate`** after the loop can also be eliminated (as there are no non-owner writes, and all non-owner reads are realized by the compiler). The matching **`implicit_writable`** call can be made faster: we only need to change access permissions if it is the first time around that a range of blocks is asked to do **`implicit_writable`**. At subsequent times the call need only do the test and nothing more, as long as it is the same range of blocks (there is some extra work required for dealing with overlapping ranges; we omit the details).

We are implementing an intra-procedural version of PRE to systematically reduce both types of overhead. We intend to compare the PRE-based approach with the run-time scheme.

5 Experimental Platform

As mentioned earlier, our experimental platform is a Tempest implementation on a 8-node cluster of SparcStation20 workstations, connection by the Myricom Myrinet. The default coherence protocol (written completely in software as unprivileged code) is an eager-invalidate multiple-writer release consistency protocol—it attempts to hide write latency by not waiting for the write ownership grant from the home node. At synchronization points, a node waits for all pending transactions to complete. We have augmented the protocol to support the primitives mention in Section 4.2; the primitives are called only after waiting for all outstanding transactions. The Tempest implementation used in this study accelerates access control functions by using a custom memory-bus device [28]. Tempest implementations that do not use this device exist, but are somewhat slower. Since our workstation nodes are dual-processor, there is an option of either dedicating a processor for protocol processing, or interleaving protocol processing with computation on the same processor (ignoring the second cpu altogether—computation is always done only on one cpu). We report results on both configurations to evaluate the overhead reductions of our scheme on two reasonable system design points. Some details on various components of the system are summarized in Table 1.

Table 1: Some details of the cluster configuration used.

Processor	66 MHz HyperSPARC (2)
Network Interface	Myricom's Myrinet
Minimum roundtrip latency for short (4 bytes) message	40 μs
Network bandwidth	20 MB/s
Read miss processing time for 128 byte block (2 cpu)	93 μs

6 Results

We present results on six HPF applications listed in Table 2 with their problem sizes and memory usage.

Table 2: Application Suite

Application	Source of HPF version	Problem Size	Memory(Mb)
pde	Genesis. HPF by PGI	grid size 128, 40 iters (RELAX routine only)	56
shallow	NCAR. HPF by PGI	1025x513 grid, 100 iters	28
grav	HPF by Syracuse	grid size 128, 5 iters	17
lu	Stanford. HPF by authors	1024x1024 matrix (5 runs)	4
cg	HPF by MIT	180x360 matrix, converges in 630 iters	4.6
jacobi	HPF by authors	2048x2048 matrix, 100 iters	32

For each application, we present speedups on our 8-node cluster, both for single-cpu and dual-cpu configuration of the Tempest implementation. We also present speedups obtained by directly porting the PGI's message-passing run-time to use Tempest messages. Note that all the applications in this suite are parallelized well by the message-passing compiler—however the messaging layer does not perform well on some applications. All speedups are calculated relative to a uniprocessor run on a similar workstation albeit with more (96 M) physical memory, so none of the applications page. While the uniprocessor codes do not have any runtime parallelism overhead, they are not blocked for cache performance [35], which explains the superlinear speedups that we obtain on some programs. We report timings for 5 iterations of application *lu,* because for the problem size used, it spends significant fraction of its time in mapping remote pages during the first iteration.

Figure 3 shows the overall speedup improvements in the six applications. Overall improvements in the speedups are quite encouraging. All our optimized versions, except *grav*, show good speedups on 8 nodes. As expected, the single-cpu shared-memory versions run somewhat slower than the dual-cpu versions, and therefore show proportionately higher improvements with the compiler optimizations. We reiterate that the dual-cpu versions use the second cpu only for protocol processing purposes: the computation is still done only on one cpu in all versions—thus there are overall 8 computation threads in all versions. The message-passing version has a performance advantage over all shared-memory versions only in *lu*. Surprisingly, in all other examples, message-passing perform somewhat slower than dual-cpu shared-memory versions, particularly so in *cg*. We believe this is due to some (as yet unidentified) performance bottlenecks in PGI's messaging run-

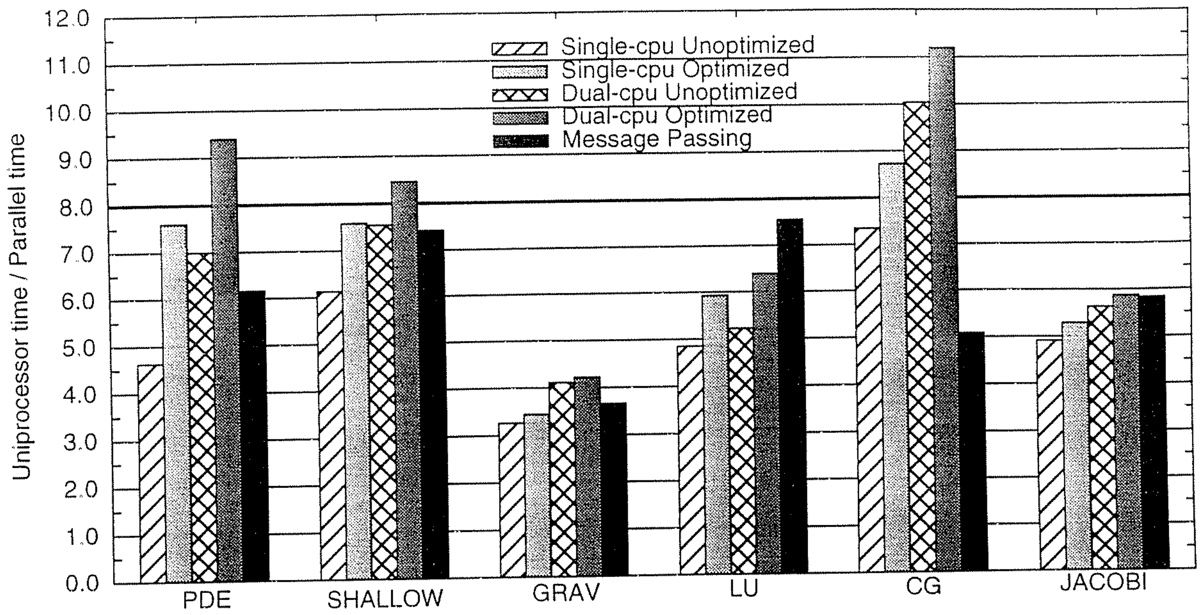


Figure 3: Speedups with various configurations. Compiler-directed protocol optimizations improve shared memory speedups in all cases.

time, or in our adaptation of PGI's primitives to use the Tempest dual-cpu communication facilities well—this is under further investigation.

Table 3 shows detailed timing breakups and event counts that allow us to analyze the performance results.

Table 3: Reduction in miss count and communication time.

Application	Compute time (seconds)	Comm time dual-cpu (seconds)	% reduction with opt	Comm time single-cpu (seconds)	% reduction with opt	Miss Count (K)	% reduction with opt
pde	33.6	26.1	58.6%	56.5	61.9%	293.8	74.6%
shallow	35.2	10.9	45.9%	21.5	50.2%	55.8	85.7%
grav	12.0	11.6	5.5%	17.8	9%	42.5	38.2%
lu	51.1	27	53%	32.9	47.4%	85.8	85%
cg	13.6	9.8	24.4%	18.4	27.7%	57.9	68.7%
jacobi	31	4.3	33%	9.5	30.5%	22.5	96.7%

Communication times report the spent in the unoptimized versions waiting for servicing misses and for synchronization. Communication time in the optimized case (not explicitly shown) includes the time spent in various protocol calls. Miss counts are the average number of misses per-node encountered in the unoptimized programs (same for single and dual-cpu). The percentage reduction columns give the reduction in the corresponding quantity with the optimizations enabled. Note that the reduction in miss count does not proportionately reduce the communication time, because data transfer and synchronization costs may differ in each case.

In all applications except *grav*, we see a significant decrease in the number of misses, which shows our approach is quite effective in capturing most of the communication under compiler control—the remaining misses arise due to first access (cold miss) and due to the edge cases in each array section that we omit by our shmem_limits call. *Grav* shows a shortcoming of our approach, in that a significant fraction of misses are not removed (only 38% are removed). This is because the array extents in *grav* are rather small (129x129 reals and 129x129x129 reals), and thus the edge effects are pronounced at 128-bytes blocksize. *Grav* executes a large number of SUM reductions, which, while efficiently implemented using low-level messages, ultimately limit speedups in both shared memory and message passing.

Lu performs LU-decomposition, in which during each iteration a pivotal column is broadcast to all processors. Since it is a triangular loop, the size of this column decreases with successive iterations, and in the later columns the edge effects limit the efficacy of our optimizations. Although the overall miss counts decrease by 85%, and communication costs by about 50%, we are still not as fast as message-passing on this application.

The rest of the applications are regular stencil based computations, with relatively large columns shared between processors in a producer-consumer relationship. Our techniques are ideally suited for such cases, and we get good speedups. We were able to eliminate a large fraction of the misses and significantly decrease communication costs. *Shallow*, *pde*, and *cg* show opportunities for redundant communication elimination, which should increase performance even further.

Finally, Figure 4 shows the contribution of performing larger-payload transfer of contiguous blocks (bulk transfer) and run-time overhead elimination (Section 4.3) in reducing total execution time for dual-cpu case (the single-cpu case is qualitatively similar). The base optimizations bar shows the reduction in execution time relative to unoptimized version, when only sender-initiated transfers (Section 4.2) are performed. The next two bars shows further improvements when bulk transfers and run-time overhead elimination are also performed. As evident from this data, both these optimizations are important to gather the most benefit from our scheme (however bulk transfer is the more important optimization).

7 Conclusion

We described a method for optimizing communication costs for regular HPF programs on a fine-grain distributed shared memory system. Our method uses analyses similar to those used in compilers targeting message-passing machines, to identify cache blocks that are candidates for protocol optimizations. We described how to handle the problem of multi-word cache blocks, and also presented a contract between a compiler and a coherence protocol that makes use of incoherence for performance advantages. With bulk-transfer of contigu-

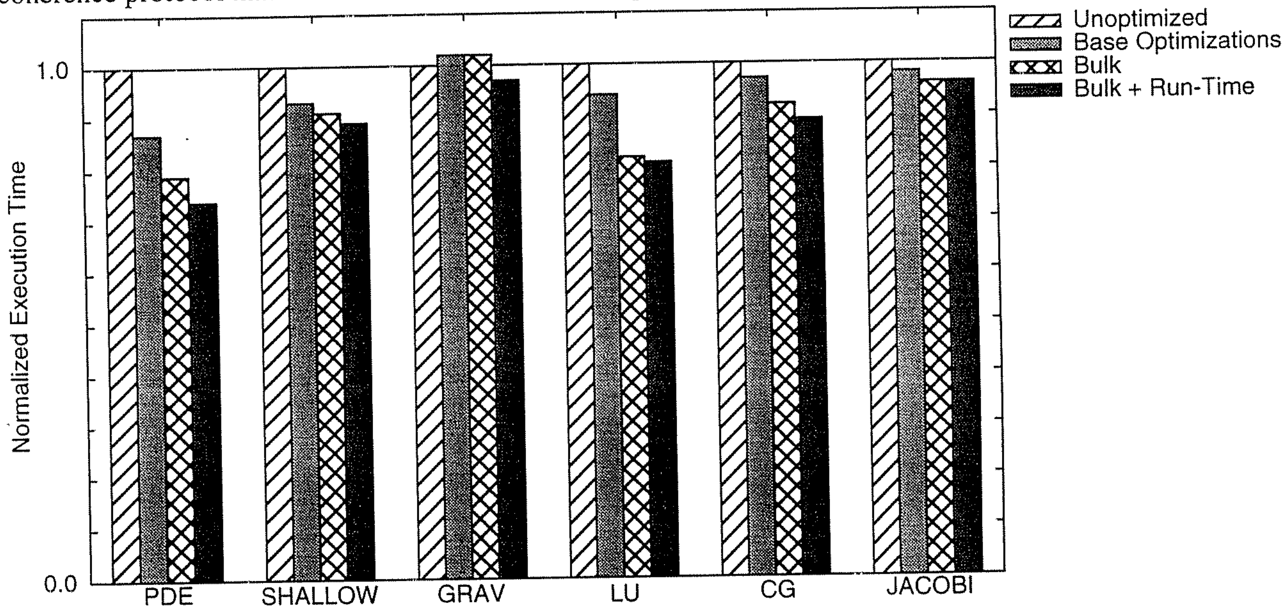


Figure 4: Benefits of bulk-transfer and run-time overhead elimination.

ous blocks, and elimination of run-time overhead, we get substantial performance improvements on this suite of applications.

Our techniques should directly apply to other systems that implement fine-grain shared memory in software, such as Flash[20] and Shasta[31]. In fact, from the compiler's perspective, the full generality of customizing coherence protocols is not required for optimizing regular communication: following the discussion in Section 4.2, we only need certain primitives to be able to bypass the general coherence protocol. We believe that most emerging commercial parallel systems will provide fine-grain shared memory (there is no going back from the SMP programming model), and optionally provide escapes to bypass global coherence. This work has shown a compiler-directed approach to exploit such escapes.

In the future, we intend to incorporate PRE based analysis to systematically reduce overheads as discussed in Section 4.3. We also plan to include more benchmarks in our workload, particularly those that show a mix of simple affine array subscript and indirect array subscripts, and are not amenable to purely message-passing approaches.

References

[1] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstation s. *IEEE Computer*, pages 18–28, February 1996.

[2] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and Computation Transformations for Multiprocessors. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, July 1995.

[3] Vasanth Balasundaram. A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. *Journal of Parallel and Distributed Computing*, 9(?):154–170, November 1990.

[4] David Callahan and Ken Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[6] Satish Chandra and James R. Larus. HPF on Fine-Grain Distributed Shared Memory: Early Experience. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[7] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. *IEEE Computer*, 23(6):39–48, June 1990.

[8] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic Management of Programmable Caches. In *Proceedings of the 1988 International Conference on Parallel Processing (Vol. II Software)*, pages 229–238, Aug 188.

[9] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 186–197, October 1996.

[10] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[11] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[12] E. Granston and A. Veidenbaum. Detecting Redundant Accesses to Array Data. In *Proceedings of Supercomputing '91*, pages 854–965, 1991.

[13] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, June 1991.

[14] Manish Gupta, Edith Schonberg, and Harinia Srivivasan. A Unified Framework for Optimizing Communication in Data-Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):689–704, July 1996.

[15] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in it Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)/.

[16] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[17] Pete Keleher and Chau-Wen Tseng. Enhancing Software DSM for Compiler-Parallelized Applications. Technical Report CS-TR-3698, Computer Science Department, University of Maryland, College Park, September 1996.

[18] K. Kennedy and N. Nedeljkovic. Combining Dependence and Data-Flow Analyses to Optimize Communication. In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[19] D.A. Koufaty, X. Chen, D.K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multprocessors. In *Proceedings of the 1995 International Conference on Supercomputing*, page ?, 1995.

[20] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[21] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–218, October 1994.

[22] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[23] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[24] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth Proceedings of Symposium on Architectural Support for Programming Languages and Operations Systems*, pages 62–73, October 1992.

[25] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

[26] William Pugh and David Wonnacott. Eliminating False Data Dependencies using the Omega Test. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, pages 140–151, June 1992.

[27] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[28] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[29] Anne Marie Rogers. Compiling for Locality of Reference. Technical Report TR 91-1195, Department of Computer Science, Cornell University, March 1991. PhD thesis.

[30] E. Rosti, E. Smirni, T.D. Wagner, A.W. Apon, and L.W. Dowdy. The KSR1: Experimentation and Modeling of Poststore. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1993.

[31] Dan Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Appraoch for Suppoting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 174–185, October 1996.

[32] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lucas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, and David A. Wood. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[33] E. Torrie, C-W. Tseng, M. Martonosi, and M. W. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, June 1995.

[34] Chau-Wen Tseng. *An Optimizing FORTRAN D Compiler for Distributed Memory MIMD Machines*. PhD thesis, Rice University, January 1993. Also available as Rice CRPC-TR93291-S.

[35] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 30–44, June 1991.