# Optimizing Communication in HPF Programs on Fine-Grain Distributed Shared Memory

Satish Chandra and James R. Larus

Computer Sciences Department
University of Wisconsin—Madison
1210 West Dayton Street
Madison, WI 53705 USA
{chandra, larus}@cs.wisc.edu

**Abstract.** *Unlike compiler-generated message-passing code, the coherence mechanisms in shared-memory systems work equally well for regular and irregular programs. In many programs, however, compile-time information about data accesses would permit data to be transferred more efficiently—if the underlying shared-memory system offered suitable primitives. This paper demonstrates that cooperation between a compiler and a memory coherence protocol can improve the performance of High Performance Fortran (HPF) programs running on a fine-grain distributed shared memory system up to a factor of 2, while retaining the versatility and portability of shared memory. As a consequence, shared memory's performance becomes competitive with message passing for regular applications, while not affecting (or in some cases, even improving) its large advantage for irregular codes. This paper describes the design of our implementation and reports 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 for programs with false sharing or poor locality of reference. Another well-known source of overhead is the fixed cache-coherence protocols used in shared-memory multiprocessors. For example, transferring a single piece of data from a writer to a reader processor may require four or more messages (see Figure 1) [10]. A recent study by Torrie et al. [32] showed that memory system overhead could account for more than 30% of the execution time of a suite of compiler-parallelized programs. The underlying problem is that conventional multiprocessors provide only a single, fixed coherence protocol to communicate values among processors. Unfortunately, no protocol performs well in all circumstances. In these systems, software cannot avoid coherence overhead *per se*, although many latency reducing and tolerating techniques have been proposed [11,22]. Moreover, on cost-effective systems that implement shared memory on a cluster of workstations [1,31], higher communication latencies make coherence overhead even more taxing.

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, 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.

To appear at the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Las Vegas, June 18-21, 1997.

Shared-memory systems are beginning to provide alternative ways to speed data transfers, which range from new memory operations to the option of bypassing the coherence protocol. All-hardware systems provide simpler operations, but a multiprocessor can provide memory system operations—such as poststore [29], co-operative prefetch [14], self-invalidate [14], or store-and-forward [18]—that a programmer or compiler can use to improve performance. Systems that implement coherence in software—such as Typhoon [26], Flash [19], Shasta [30], and most page-based systems [1,3]—can go further and offer message-passing-like communication primitives or customizable protocols. Even some commercial shared-memory multiprocessors, such as STiNG [21], offer some support for data transfer mechanisms beyond a fixed coherence protocol.

This paper describes a compiler-directed approach to exploiting improved communication mechanisms. Our work focuses on HPF programs running on a fine-grain distributed shared memory (DSM), though the approach is more generally applicable. In particular, we show how static program analyses previously developed to compile for message-passing systems [28,33] can also identify opportunities for efficient value transfer in shared-memory systems. Our HPF compiler uses these analyses to identify cache blocks for which efficient communication is both beneficial and safe. It then inserts run-time calls that explicitly manage

communication for these blocks. In the parts of program in which the necessary preconditions for data accesses cannot be found at compile-time, our system allows the default protocol to manage communication. A key contribution of this work is the development of a contract between the compiler and coherence protocol, so they can co-operate to reduce data-transfer costs for data structures and program phases where static analysis permits. Our techniques are appropriate for fine-grain shared-memory systems, as they bypass the default coherence mechanisms on small amounts of data. Page-based systems require a somewhat different approach, though the compiler analysis is similar.

The approach that we advocate, shared memory with coherence optimizations, offers the important benefit of greatly expanding the domain of HPF programs that can be written and compiled effectively. This expansion is made possible by combining the efficiency of message passing with the flexibility of shared memory. In our system, the compiler communicates cache blocks through explicit message passing, which bypasses shared-memory communciation overheads. This results in near-message-passing performance for regular programs.

Although compiler-generated message passing works well for *regular* programs[1], it fails for programs containing *irregular* references. Irregular references inside a parallel loop[2] force a compiler to either generate scalar messages inside the loop, or pay for extraneous communication—potentially broadcasting the entire data set. Both these options lead to poor performance [28,7]. In special cases, a compiler can use the *inspector-executor* technique [17], which ultimately implements a shared memory view of selected arrays [23]. But, a system-provided shared-memory layer frees a compiler from the straitjacket of precise static analysis and permits irregular programs to run efficiently.

To demonstrate our approach, we modified a commercial HPF compiler—the Portland Group's *pghpf*—to generate simple shared memory code [4] and to perform the communication analysis necessary to insert run-time calls to our extended coherence protocol. Our target is the Blizzard system [15], which implements distributed shared memory at the granularity of cache blocks (e.g. 32-128 bytes) and enables application code to provide its own coherence protocol. We performed our experiments on a Blizzard system running on an 16-node cluster of SparcStation20 workstations connected by a high speed network (Myricom Myrinet). Our results show that optimizations reduced the overall execution times by up to 45% on a suite of 8 regular and irregular HPF applications. On regular programs, most optimized execution times are competitive with *pghpf*'s default message-passing performance. Moreover, unlike *pghpf*, we also achieved good performance on the irregular programs in our application suite.

1. Programs in which all array subscripts are affine functions of surrounding loop indices and precise communication analysis is possible. Complicated control-flow inside loops can sometimes obstruct analysis on seemingly regular references.

2. A loop that has no loop-carried dependence and hence all its iterations could run in parallel. This could either be inferred by the compiler, or a programmer could declare a loop to be parallel by using an `INDEPENDENT` directive.

Although remote memory references in Blizzard running on the cluster of workstations are costly in absolute terms (~100 µs), they do not differ from modern hardware shared memory systems nearly as much when measured in instructions. Emerging parallel systems generally provide fine-grain shared memory. This work demonstrates that these systems should also provide coherence protocol bypasses that a compiler can exploit to improve the performance of shared-memory applications.

The paper is organized as follows. Section 2 discusses related work. Section 3 provides background materials on coherence protocols and discusses the opportunities for coherence optimizations in fine-grain distributed shared memory. Section 4 describes 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.

## 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 compilers exploit well-known techniques for static array analysis to make data transfer cheaper by reducing coherence overhead and performing sender-initiated transfers. However, Dwarkadas's techniques are targeted at page-based DSMs (such as TreadMarks[1]). In TreadMarks, detecting and preparing for writes (twinning) are expensive operations. Not surprisingly, the most profitable optimization is to prevent write-faults from occurring. Dwarkadas found that sender-initiated transfers yield minor benefits. By contrast, in fine-grain shared memory systems, such as ours, the cost of gaining write-ownership is far lower, so we optimize for the delays in true sharing.

Furthermore, Dwarkadas's compiler analysis requires and uses only localized access information between program barriers. This choice is appropriate to compile a phase of an explicitly parallel program. However, since we focus on compiler-parallelized programs, further optimizations are possible. In compiler-parallelized programs, work distribution is determined at compile-time and typically follows the owner-computes rule [28] (although our scheme can handle other computation distributions as well). This compile-time analysis enables us to track the movement of cache blocks across parallel loops, and allows us to consider for optimization only the blocks containing array elements that are involved in producer-consumer relationships. Without such global information, the compiler has to apply coherence

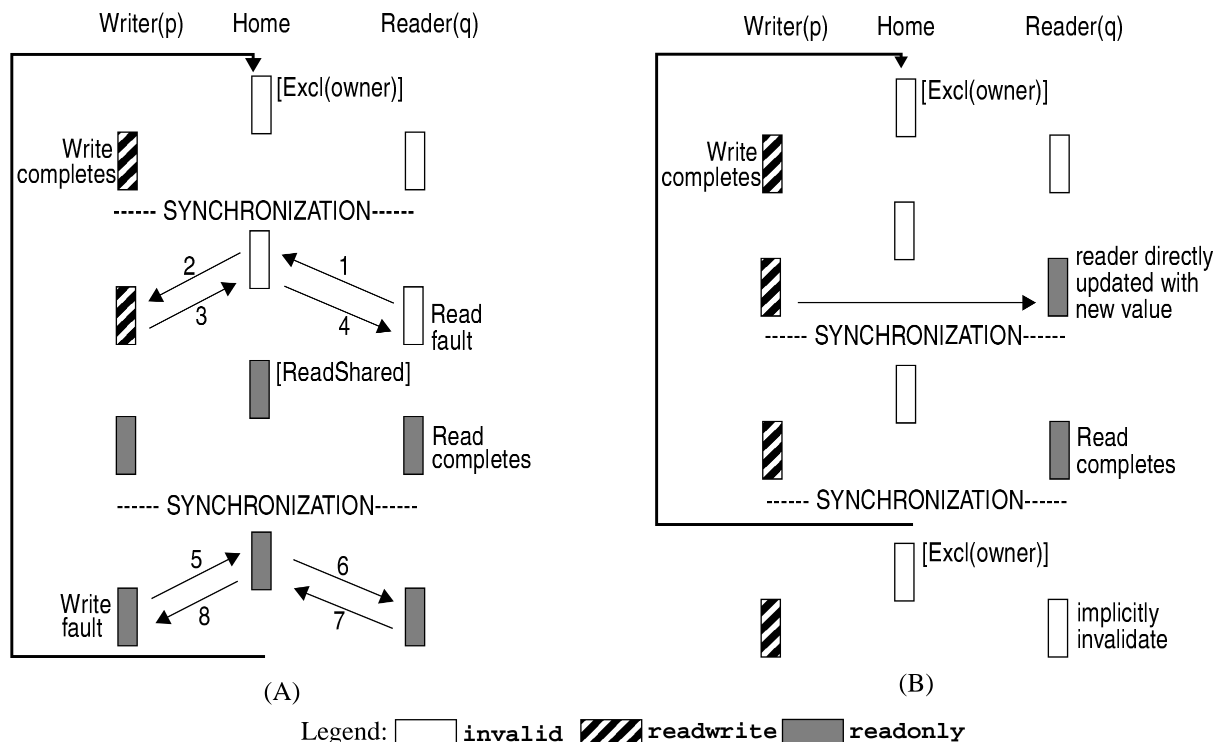



**Figure 1:** (a) Default coherence scheme. Notice the number of messages required to transfer one block. The messages are as follows: 1. read-request 2. put-data-request 3. put-data-response 4. read-response 5. write-request 6. invalidation 7. acknowledgment, and 8. write-grant. (b) Direct update message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

optimizations to *all* blocks accessed in a parallel loop. However, in Dwarkadas's work, the runtime system keeps track of modified pages and avoids the cost of validating such pages. Second, we exploit the availability of fine-grain access permissions to reduce the cost of coherence-manipulating calls to a protocol (Section 4.3). Although these calls may not incur a large overhead on a page-based system, fine-grain systems have many more transfer units (cache blocks).

Keleher and Tseng [16] reduce miss time on their page-based DSM by flushing modified pages to prospective readers, as opposed to having readers fetch them on a reference (e.g., poststore). They also use compiler analysis to find a set of pages that are communicated in a stable pattern, although the actual detection of producers and consumers is left to a run-time system (they observe that this could be done by a compiler). In contrast with Dwarkadas's and our work, their approach does not relax the system's coherence to permit a compiler to control all accesses to shared data. When applicable, we use precise compiler analysis to identity the blocks that need to be send from a writer to readers and then bypass coherence protocol when possible.

Another, proposed group of machines *require* software involvement to maintain coherence [24]. Software cache coherence raises difficult problems in identifying 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 [8,6]. However, these schemes are intrinsically conservative and must work for all data accesses—rather than the selected ones on which we focus—so they often suffer from excessive invalidations and re-fetches. In our system, a default protocol automatically fetches the latest value at a read: we only seek to make this transfer more efficient.

Larus et al. [20] used compiler-controlled incoherence to efficiently implement a data-parallel language. Their fine-grain, copy-on-write protocol for specially marked blocks reduce the 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 & Optimization

A typical coherence protocol supports two functions. First, it satisfies a load operation by shipping the current value of the requested location 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. Details, such as how soon a reader can expect to see a newly written value, vary according to the consistency model underlying the protocol. An important limitation of coherence protocols is that

respond to memory references, without global knowledge of a program's memory accesses.

As an example, consider the common cases of 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 a block's size is equal to the size of the value being transmitted. Furthermore, assume an invalidation-based protocol (general update-based protocols have analogous problems, but details are omitted for brevity). Figure 1(a) shows a typical sequence of coherence actions for this transfer. At all times, the directory (a data structure maintained at the *home* node) must correctly capture the state of the block, because any other processor is free to join the fray by accessing the location. Figure 1(a) also illustrates our default protocol[1].

In many programs, *p* and *q* repeatedly perform this communication, say in a time-driven loop. Assume that the memory location in question is neither read, nor written by any other processor. In this situation, *p* could directly send a new value to *q*, provided a send primitive is available. As long as *p* and *q* directly communicate and no other processor accesses the location, the system's directory need not track the current state or contents of the block (see Figure 1(b)). To end this phase, processors *p* and *q* make their local state consistent with the directory information. The preceding discussion is somewhat over-simplified for explanatory purposes. Section 4.2 describes the contract between our compiler and protocol in detail.

Tempest is an interface provided by a shared memory system that allows coherence protocols 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 **`invalid`**, **`readonly`**, or **`readwrite`** protection on individual blocks. An access to **`invalid`** block, or a write access to a **`readonly`** block invokes a user-specified fault handler. (3) Fine-granularity low-latency messages. Ordinary protocols that implement transparent shared memory use all three mechanisms to provide the desired consistency model. Our compiler goes further and directly invokes fine-grain access control and messaging primitives, to bypass the default coherence protocol in certain cases.

In principle, we should be able to completely bypass the default coherence protocol when perfect information about readers and writers is available for an application. However, some practical problems must be addressed. First, real DSM systems maintain coherence on blocks larger than a single word. A particular block can hold several array values, even those that straddle dimensions. For example, *a*(513,1) and *a*(1,2) could reside in the same block for a 513x513 array. In this case, it is not always possible to draw conclusions about the usage of all elements in a cache block. For example, the compiler may believe that it can orchestrate communication for *a*(513,1). However, it may not be able to ask the run-time to manipulate access permissions to the block that contains *a*(513,1), because the compiler may not have any guarantees for accesses to *a*(1,2). This problem does not manifest itself for compilers targeting message-passing machines. They synthesize a global space from private memories, so there is no notion of two array locations residing in the same shared-memory block. Second, the compiler and the coherence protocol must share a simple representation of blocks that are under compiler control. A explicit listing of blocks can introduce impractically high run-time overhead. A summary, unfortunately, introduces imprecision. Subsequent sections discuss the design choices that we made to address these issues.

1. This protocol tries to hide some write latency with a release-consistent memory model. Weaker memory models help pipeline coherence messages, but do not necessarily reduce their number. Compiler based techniques such as ours directly reduce the number of coherence messages.

# 4 Compiler-Orchestrated Incoherence

A 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 run more efficiently—this forms the core of our technique. Third, it optimizes the placement of these calls.

## 4.1 Access Information

The compiler must determine the sections of arrays that are read and written in each parallel loop, so it can find the communication involved in executing the loop. Furthermore, this computation must take into account the distribution of the arrays (as specified by 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 the owner. We currently make a simplifying assumption for data distributions: only the last dimension of a global array is distributed (either blockwise or cyclically) on a linear arrangement of processors. Multi-dimensional distributions, e.g. `(BLOCK, BLOCK)`, can sometimes reduce the overall volume of communication. Our scheme could still optimize communication along the last dimension—one approach to optimizing the remaining communication would be to copy the boundary data into contiguous buffers and then optimizing the buffer transfer.

The computation distribution is usually owner-computes, but this is not a restriction in our scheme. A compiler can

use the programmer-supplied INDEPENDENT directive to partition 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 multi-word blocks, which in most cases occur at the boundary elements of array columns. We do not optimize for these boundary cases[1]. By contrast, the large size of the coherence unit in page-based DSMs can introduce significant communication due to false sharing, and it is important to optimize for it [9].

Our implementation uses Maryland's *Omega* library [25] to compute these sets. Although the kind of sections we optimize could be represented by traditional regular section descriptors (RSD) [13], the Omega library enabled us to avoid the significant implementation effort required to build a robust RSD package. In addition, the Omega library handled symbolic variables that appeared in our test cases, as well as kept access sets parametric with respect to processor number. To obtain a succinct representation of the blocks to take under compiler control, our optimizations only apply to 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

This section describes 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 must be careful when taking a particular block under compiler control. 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, this subsetting requires iteration over the higher dimension. The boundary cases are left to the default protocol. These boundary cases could be optimized with *advisory* primitives, such as self-invalidate and co-operative prefetch [14], which may be worthwhile when the compiler-managed data set size is small. It is also possible that there are no accesses to off-section elements residing in boundary blocks. Compile-time analysis to determine this property must make assumptions about starting addresses of arrays and the block size. We have not explored this option yet.

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.

Figure 2 shows how our compiler modifies the default protocol for a non-owner read reference. These calls are produced for each non-owner reference in a loop. Similar calls for different references are grouped together to share synchronization. The first run-time call, **shmem_limits**, establishes the restricted limits, as described above, and returns a communication descriptor (Figure 2a). We also pass the values of symbolic variables to these run-time calls, where they provide input for the analytical expressions generated by the Omega library. Table 1 summarizes the remaining calls and the rationale for the additional synchronization, which are further described below.

**Table 1:** Run-time calls and their effect

| **Run-time Call** | **Effect** |
|---|---|
| shmem_limits | Calculate the starting and ending cache block. |
| mk_writable | Owner brings all the blocks in the range in exclusive mode. |
| implicit_upgrade | Readers make the specified blocks **readwrite**, so they can store the incoming data without an access fault. |
| implicit_invalidate | Readers make the specified block **invalid**. |
| send | Owner sends the specified blocks to the named readers (in bulk) |
| ready_to_recv | Readers wait until the expected number of blocks have been received and stored. |
| flush | Writer sends the specified blocks to the owner. |

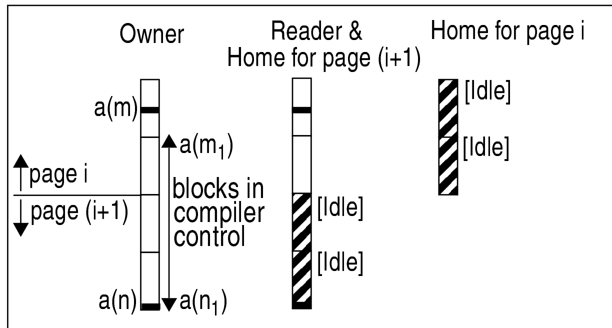
The overall goal of this optimization is to make non-owner blocks available before a parallel loop executes, so that no access faults occur during the loop. For the moment, ignore non-owner writes. We designate owners to send the relevant blocks to the readers. Senders and the receivers need to make certain preparations before this transfer can take place:

(1) Since the owner is not necessarily the home node, there is no guarantee that the owner has a copy of the block it

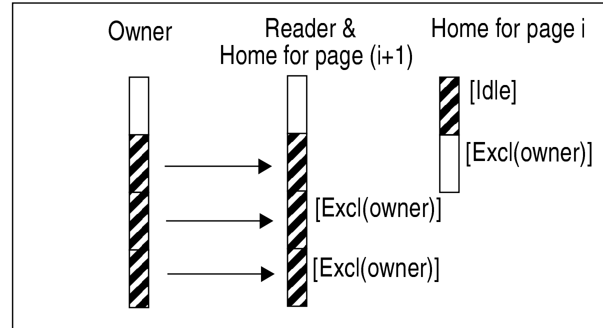
must send to potential reader(s). Therefore, all owners must first bring the relevant blocks to readable state in their caches[1]. Moreover, in anticipation that the owners will eventually write new values, we actually bring the relevant blocks into the writable state before initiating the transfer. An important side effect of this step is that the directory information for these blocks records that the owner has the current (and only) valid copy, thus relieving the actual home, if it is not the owner, of responsibility of carrying a valid copy. We will employ this observation momentarily.

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

(2) In Tempest, readers require **`readwrite`** permission to store incoming data (as for any store). In ordinary Tempest, coherence protocols, 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, the blocks being brought over

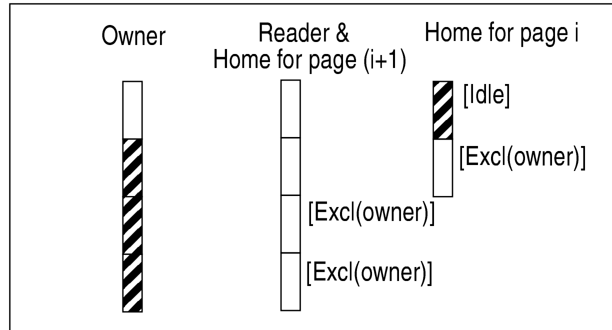



A. Initial State, and **`shmem_limits`**. Annotations next to blocks at the home nodes, e.g. [Idle], denote directory state.



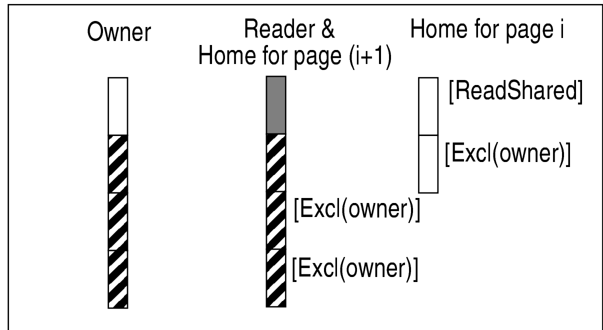

D. After **`send`** and **`ready_recv`**

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

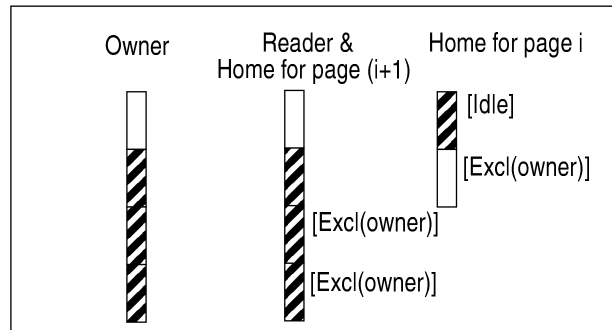



B. After **`mk_writable`**

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

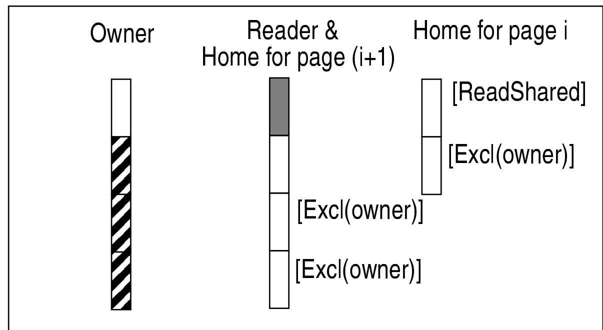



E. After the loop executes. Note that the reader has accessed locations a(m) to a(n), making the first block read-shared.




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

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




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.

are under explicit control, and the compiler can keep them in whichever state it prefers. In our system, these blocks have **readwrite** access permission, to reduce the need to change access permissions. After the parallel loop has executed, the compiler invalidates these blocks.

Steps 1 and 2 must be ordered. 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 a valid copy of a block. A barrier synchronization ensures that step 1 completes before starting step 2.

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

Another synchronization after step 2 guarantees that both the senders and receivers are ready for the transfer. The senders ship the relevant blocks to the receivers with a **send** operation on a range of addresses and a list of destination ids. The underlying protocol sends the blocks in specially tagged data messages to each recipient. Each receiver posts a **ready_to_recv** call, which holds down a counting semaphore until all blocks have arrived. As a further optimization, we group contiguous blocks and transfer them in larger payloads than a single block. This optimization provides the benefit of larger block sizes, and accounts for almost half of the net reduction in program execution time achieved using our techniques. At the conclusion of this data transfer, the non-owner data has been transferred in (at least) readable state (Figure 2D).

After the parallel loop executes, the directory for a compiler-controlled block records 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 readers also have a writable copy of those blocks. Without information about future data accesses, we invalidate readers's copies with an **implicit_invalidate** call to the protocol (Figure 2F). A final barrier ensures that the memory is again consistent with the directory.

Finally, consider 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 must **flush** its changes back to the owner and implicitly invalidate its copy, so that the owner has the only latest (writable) copy of the block.

## 4.3 Reducing Run-time Overhead

All run-time calls inserted by the compiler need not be executed every time a loop executes. For instance, if a subsequent parallel loop has the same computation distribution as an earlier loop, blocks' owners already have the blocks in writable condition (assuming no intervening read invoked default coherence actions). Therefore the **mk_writable** call can be eliminated from the second loop.

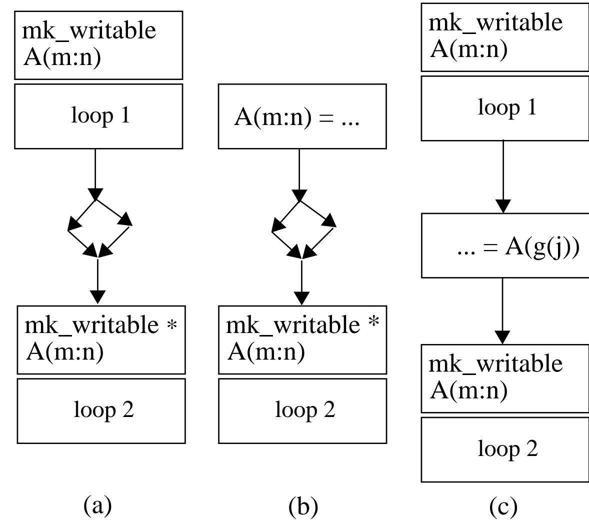



**Figure 3:** Exploiting availability of access permissions. Calls marked with an asterisk (*) are redundant. (a) and (b) show examples of *gen*erating the availability of access permissions, and (c) shows how the availability can be *kill*ed.

We cast this observation as a data-flow problem, similar to *available expression* analysis. Within a given subroutine, our data-flow universe is the set of non-owner array sections transferred under compiler control. To simplify the data-flow analysis, we treat sections as indivisible entities and do not analyze the availability of parts of sections [12]. An element in this universe is *gen*erated when a parallel loop invokes a **mk_writable** call that contains the section represented by the element. It is also *gen*erated when a parallel loop causes the owner to write all cache blocks in the section—the writes bring the corresponding cache blocks into **readwrite** state (see Figure 3). An element is *kill*ed by a "stray" non-owner read, since a read reference that is not under compiler control can engage the default protocol and destroy the condition that the cache block is in **readwrite** state at the owner. This data-flow problem is then solved in the usual iterative manner, and redundant calls to **mk_writable** are eliminated. Consequently, the barrier that enforces the completion of the redundant call before the next run-time call is also removed.

A similar scheme could also exploit the availability of already fetched remote data, and thus eliminate **send** and **ready_to_recv** pairs. Recent literature on compilers for message-passing machines reports on this optimization [12]. 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. We can also eliminate some

`implicit_invalidate` and `implicit_writable` calls that would be redundant along all possible executions.

Unfortunately, we require interprocedural analysis to draw full benefit from this framework, as most of the codes are (justifiably) written in terms of subroutines. Our current (intraprocedural) implementation removes only the redundant `mk_writable` calls, and reduces parallel execution time by only up to 3%.

# 5 Experimental Platform

Our experimental platform is a Blizzard implementation running on a 16-node cluster of SparcStation20 workstations, connection by a Myricom Myrinet high speed network. The Blizzard implementation in this study uses a custom memory-bus device to accelerate fine-grain access control [27]. Blizzard implementations that do not use this device exist, but are somewhat slower.

**Table 2:** Wisconsin COW (Cluster of Workstations).

| | |
|---|---|
| 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 |

The default coherence protocol (written completely in software as unprivileged code [5]) is an eager-invalidate, multiple-writer, release consistent protocol. It attempts to hide write latency by not waiting for a write ownership grant from a home node. At synchronization points, a node wait for all its pending transactions to complete. We augmented the protocol to support the primitives from Section 4.2.

Since our workstation nodes are dual-processor, we can either dedicate a processor to protocol processing or interleave protocol processing with computation on a single processor (ignoring the second processor altogether—as computation is always performed only on one processor). The second processor should be viewed as an accelerator for protocol processing—alternative high-end designs include a dedicated protocol processor on the memory controller [19] or the network interface [26]. We report results for both configurations to evaluate the benefit of our optimizations for two reasonable system design points. Table 2 summarizes relevant details of our system.

# 6 Results

We studied eight HPF application codes[1], listed in Table 3 along with their problem sizes and distribution of primary arrays. The first six applications are regular programs collected from various sources. The communication in *pde*, *shallow*, *grav*, *cg*, and *jacobi* consists exclusively of shifts and reductions. *Lu* is a triangular solver that broadcasts a pivotal column—a different one in each iteration—to all processors. In this application, each processor accesses almost the entire input data set. It, therefore, spends significant time mapping remote pages on its first run. To isolate this effect, we report data collected on the last 5 of 6 runs of the application (the first run took 30% more time than later runs).

**Table 3:** Application Suite

| Application | Problem Size & Distributions |
|---|---|
| pde | grid size 128, 40 iters (`RELAX` routine only)<br>`(*,*,BLOCK)` |
| shallow | 1025x513 grid, 100 iters<br>`(*,BLOCK)` |
| grav | grid size 128, 5 iters<br>`(*,*,BLOCK)` and `(*,BLOCK)` |
| lu | 1024x1024 matrix (5 runs)<br>`(*,CYCLIC)` |
| cg | 180x360 matrix, converges in 630 iters<br>`(*,BLOCK)` |
| jacobi | 2048x2048 matrix, 100 iters<br>`(*,BLOCK)` |
| lcp | 8192 variables, 0.5% sparsity<br>`(BLOCK)` |
| moldyn | 8788 molecules, 30 iters (graph built twice)<br>`(BLOCK)` |

The final two applications, written locally by the authors, are irregular programs. *Lcp* solves the linear complementarity problem on a sparse matrix. It has array subscripts that are themselves distributed array references. Communication in *lcp* arises from two sources: it accesses a distributed array of non-zero elements through an array subscript, `A(IA(I))`, leading to elemental communication; and, it performs all-to-all exchange of the solution vector, leading to broadcast communication. *Moldyn* is an *n*-body code that performs molecular dynamics calculations. Its irregularity arises from indirect references to a distributed array representing molecule forces. In addition, *moldyn* requires a generalized reduction, which cannot be directly expressed as a

1. We have put the applications used in this work at the web site http://www.cs.wisc.edu/~wwt/hpf/apps.html.

parallel operation in HPF, and therefore, is implemented externally.

The problem sizes of the applications used in our experiments ran for 50-400 seconds on a single processor node.

For each application, we present speedups on our 16-node cluster, both for single-cpu and dual-cpu configuration of the Tempest implementation. We also present the speedups obtained by PGI's message-passing compiler (ported to use Tempest messages, which were faster than IP). Although the nodes contained two processors, *pghpf* (version 2.0) did not benefit from the second processor, as it does not attempt to overlap computation with communication. All speedups are calculated relative to a uniprocessor run on a similar workstation, albeit with more (96 M) physical memory, so no applications pages. The uniprocessor codes do not incur any parallelism overhead, but they were not blocked for optimal cache performance.

Message passing worked well for all regular applications in our suite. However, pghpf (2.0) generated inner loop communication in the irregular applications that made parallel execution impractical. The situation might improve if the compiler employed the alternative of broadcasting complete data sets. However, we do not expect that this strategy will produce good speedups either (data in [7] corroborates this intuition). Both *lcp* and *moldyn* would benefit from inspector-executor technique.

Figure 4 shows the overall speedup improvements for the eight applications. Overall improvements are quite encouraging. All optimized versions, except *grav*, had efficiency of 50% or more on 16 nodes. Even in the irregular application *lcp*, our optimizations improved the phase that exhibited only regular references.

As expected, the single-cpu shared-memory versions run slower than the dual-cpu versions. We reiterate that the dual-cpu versions used the second cpu only for protocol processing purposes. Computation still ran on only one processor in all versions.

The most notable result from this experiment is that, except for *lu*, optimized shared memory performed almost as well as message passing on regular programs and provided respectable speedup on irregular programs. Surprisingly, *cg* failed to produce good speedup on message passing—we have not yet been able to isolate the cause of this anomaly.

Figure 5 shows the effectiveness of our optimizations at reducing cache misses and reducing communication time for the dual-cpu case.The single-cpu case showed qualitatively similar behavior, so we do not report those numbers here. Cache misses included read misses, write misses, and access faults caused by stores to read-only blocks. The communication time included the time during which the computation was suspended waiting for remote data or waiting at a barrier. We lumped synchronization time with miss time, because miss processing time shows up partly as increased wait at barriers, which would be misleading to include as synchronization time. Time spent in executing reductions (which are implemented using low-level messages) is also counted as communication time. The communication time in the optimized versions includes the time spent in run-

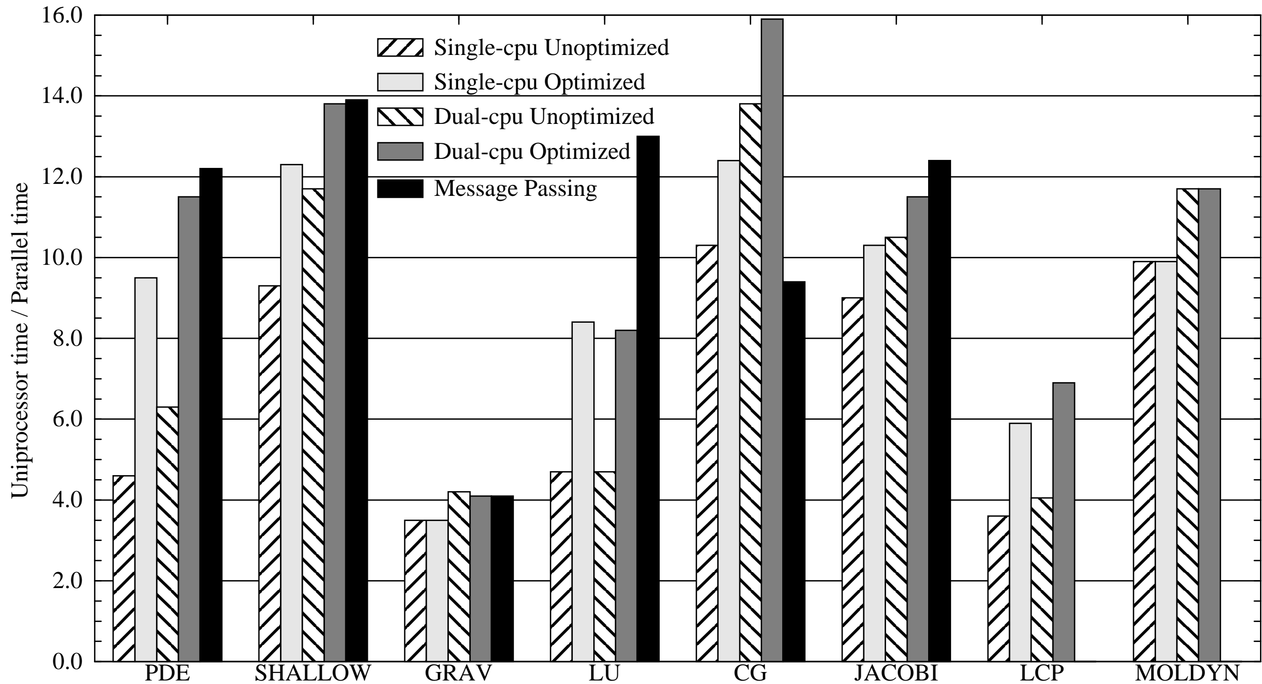



**Figure 4:** Speedups for single-cpu, dual-cpu shared memory, with and without the coherence optimizations. Speedups obtained with PGI's message-passing backend are also reported (expect *lcp* and *moldyn*, which could not be parallelized well).

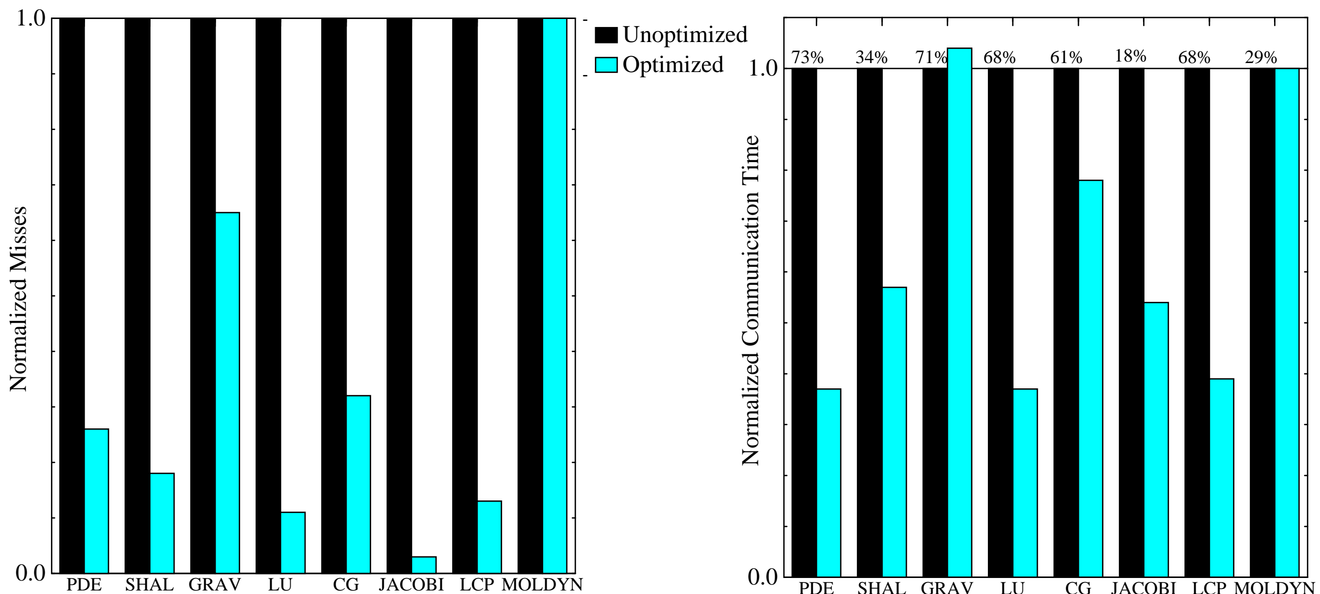



**Figure 5:** Bar charts showing reduction in misses and reduction in communication times. The normalized misses show the unoptimized miss count divided by the optimized miss count. The normalized communication times show the unoptimized communication time divided by the optimized communication time. The numbers on top of the unoptimized communication time bars show the percentage of communication time in the total program execution time in the unoptimized case.

time calls to the protocol, in addition to previously mentioned overheads.

All regular applications, except *grav*, show a significant decrease in the number of misses, which demonstrates that our approach effectively puts most of the communication under compiler control. The remaining misses are due to both first access (cold misses) and the edge cases in each array section. *Grav* shows a shortcoming of our approach. A significant fraction (62%) of its misses are not removed because the arrays in *grav* are rather small (129x129 reals and 129x129x129 reals), and thus the edge effects are pronounced at 128-bytes cache block size. The cost of additional synchronization outweighs the benefits from the coherence optimizations. Moreover, *grav* executes a large number of SUM reductions, which, although efficiently implemented, ultimately limits speedups for both shared memory and message passing.

*Lu* performs LU-decomposition, which broadcasts a pivotal column to all processors in each iteration. Since the loop is triangular, the size of this column decreases over the iterations. In the final columns, the edge effects limit the efficacy of our optimizations. Although the overall miss counts decreases by 85%, and communication costs by about 50%, shared memory is still not as fast as message-passing for this application. The message-passing *pghpf* (2.0) compiler eliminated an inner copy loop, which we could not remove from the shared memory version. The program receives the pivotal column into a private array `col`:

```
FORALL (j=i+1:N) col(j) = a(j,i)
```

This loop executes on all nodes under shared memory (although the communication is optimized). However, PGI's compiler directly deposits the incoming data into the `col` array and eliminates this loop.

*Pde*, *shallow*, *cg* and *jacobi* are regular programs, with relatively large columns shared between processors in a producer-consumer relationship. Our techniques are ideally suited for these cases, and we get good speedups. We were able to eliminate a large fraction of the misses and significantly decrease communication costs.

# 7 Conclusion

This paper describes a new method for optimizing communication in regular HPF programs running on a fine-grain distributed shared memory system. Our approach uses static program analysis in a compiler to identify cache blocks that are candidates for protocol optimizations. A contract between a compiler and the system's coherence protocol provides the means by which the compiler manages communication for these blocks. By transferring contiguous blocks in bulk and eliminating run-time coherence overhead, compiler management yields substantial performance improvements on a suite of HPF application codes. The most important consequence of this improvement is that it makes shared memory competitive with message passing on regular applications, while not affecting (or in some cases, even improving) its existing advantage in irregular codes.

Our techniques can be directly applied to other systems that implement fine-grain shared memory in software, such as Flash [19] and Shasta [30]. However, this work also shows that fully customizable coherence protocol are not neces-

sary to optimize regular communication. Our compiler only requires a limited set of primitives that bypass the general coherence protocol. We believe that most emerging commercial parallel systems will provide fine-grain shared memory and optional ways to bypass global coherence. This work motivates these bypasses by showing that a compiler can exploit them to significantly improve the performance of shared-memory applications.

In future, we plan to compare the effectiveness of semantics preserving primitives, such as prefetch and poststore, against semantics altering primitives, such as the ones used in this study. We also plan to extend our technique to handle more general data distributions.

# 8 Acknowledgments

We extend our gratitude to Krisna Kunchithapadam, Anne Rogers, and the anonymous reviewers for providing helpful comments; to Portland Group Inc. for giving us the source for their HPF compiler and answering many questions about it; to Marc Dionne and Yannis Schoinas for help with the Wisconsin COW; to Shubu Mukherjee and Guhan Viswanathan for explaining the application *moldyn* to us; to Zhichen Xu for the performance debugging tools; and to Bill Pugh's research group for building and distributing the Omega library.

# 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] 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.

[4] 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.

[5] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

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

[7] Alan L. Cox, Sandhya Dwarkadas, Honghui Lu, and Willy Zwaenepoel. Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers. In *Proceedings of the Eleventh International Parallel Processing Symposium*, April 1997.

[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] 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.

[12] 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.

[13] Paul Havlak and Ken Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[14] 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)/.

[15] 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.

[16] Pete Keleher and Chau-Wen Tseng. Enhancing Software DSM for Compiler-Parallelized Applications. In *Proceedings of the Eleventh International Parallel Processing Symposium*, April 1997.

[17] Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.