Combined Compile-time and Runtime-driven, Pro-active Data Movement in Software DSM Systems

Seung-Jai Min and Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University

Abstract

Scientific applications contain program sections that exhibit repetitive data accesses. This paper proposes combined compile-time/runtime data reference analysis techniques that exploit repetitive data access behavior in both regular and irregular program sections. We present a new compiler algorithm to detect such repetitive data references and an API to an underlying software distributed shared memory (software DSM) system to orchestrate the learning and pro-active reuse of communication patterns. We evaluate the combined compile-time/runtime system on a selection of OpenMP applications, exhibiting both regular and irregular data reference patterns, resulting in average performance improvement of 28.1% on 8 processors.

1 Introduction

Software Distributed-Shared-Memory (software DSM) Systems have been shown to perform well on a limited class of applications [3] [12]. The importance and interest of broadening this class is high, as software DSM techniques can turn a low-cost distributed memory system into a shared-memory platform, providing potentially high programmer productivity. The motivation behind the presented research is to study and realize this potential. The state-of-the-art programming method for distributed memory platforms is through message passing, typically using MPI. In creating MPI applications, programmers spend a considerable amount of time carefully and often tediously performing the following tasks. First, they partition the program into parts that can be executed in parallel and that need to exchange information infrequently. Second, they insert and orchestrate send, receive, and other message passing functions so as to package and communicate that information with as little runtime overhead as possible. In shared-memory (or shared-address-space) parallel programming models, the first task remains. However, the second task is performed by the underlying system. On software DSMs this system typically includes a page-based coherence mechanism, which intercepts accesses to data not present on the current processor and requests the data from its current owner. This mechanism incurs runtime overhead, which can be large in applications that communicate frequently with small amounts of data. In most cases, the overheads are larger than in the corresponding, hand-tuned MPI program versions, although it has been shown that the opposite *can* be the case in irregular applications, where hand tuning is difficult [12] [15]. The overhead incurred by software DSM systems appears primarily as memory access latency.

Researchers have proposed numerous optimization techniques to reduce remote memory access latency on software DSM. Many of these optimization techniques aim to perform pro-active data movement by analyzing data access patterns – either at compile-time or at runtime – so as to determine the shared data references that will cause remote memory accesses and thus expensive communication. In compile-time methods, a compiler performs reference analysis on source programs and generates information in the form of directives or prefetch instructions that invoke pro-active data movement at runtime [3]. The challenges for compile-time data reference analysis are the lack of runtime information (such as the program input data) or complex access patterns (such as non-affine expressions). By contrast, runtime-only methods predict remote memory accesses to prefetch data [2] based on recent memory access behavior. These methods learn communication pattern in *all* program sections and thus incur overheads even in those sections that a compiler could recognize as not being benefitial.

The main contribution of this paper is a *combined* compile-time/runtime approach to latency reduction in software DSM systems. Both the compiler and the runtime system share the task of data reference analysis. The compiler identifies which and when shared memory accesses will exhibit repetitive access patterns. The runtime system captures the actual data communication information such as the address and the destination of remote memory accesses and repeats them where appropriate. In order to pass the gathered compile-time knowledge to the runtime system, the compiler instruments the program, telling the runtime system when to learn which communication patterns

and when and how to apply this knowledge to pro-actively move shared data between processors. The idea of combined compile-time/runtime solutions has been expressed by others [14, 11, 9]; however, our paper is the first to present a compiler algorithm and a corresponding application program interface (API), allowing the compiler and runtime system to work in concert. Our compiler algorithm focuses on the detection of repetitive data access patterns, which are then analyzed by the runtime system. This approach contrasts with methods where the compiler identifies potential remote accesses for the runtime system to learn from. By identifying repetitive patterns at compile time, our algorithm is able to avoid runtime overheads in remote accesses that do not exhibit regular patterns. Our approach also contrasts with related work that analyzes communication patterns at compile time [5, 8, 6]. Once our compiler identifies repetitive patterns, the communication of the involved program sections is analyzed at runtime, avoiding conservative compile-time decisions.

The specific contributions of the paper are

- a new compiler algorithm that detects repetitive communication patterns,
- a combined compile-time/runtime method and implementation that uses the compiler algorithm to activate learning phases and pro-active data movements in a software DSM system through an appropriate API,
- The evaluation of the combined compile-time/runtime system on a selection of OpenMP benchmarks, exhibiting both regular and irregular communication patterns, resulting in average performance improvement of 28.1% on 8 processors.

The remainder of this paper is organized as follows. Section 2 presents the compiler algorithm and the compiler instrumentation. Section 3 describes runtime mechanisms, which we implemented as extensions of the TreadMarks [1] software DSM system. Section 4 presents results. Section 5 contrasts our approach with related work, followed by conclusions in Section 6.

2 Compiler Analysis and Transformation

The compiler plays a key role in our software DSM optimization techniques. It identifies program sections that exhibit repetitive communication patterns and controls the behavior of the runtime system, which learns and then proactively repeats these patterns.

2.1 Analysis of Repetitive Data References

Figure 1 describes the algorithm that identifies repetitive reference behavior. Data references must meet two criteria to be classified as repetitive accesses in a given loop L.

```
1.  BEGIN
2.      FOR each loop, L
3.          Initialize NonRepVar_L set to { }; Get SharedVar_L set
4.          FOR each basic block, BB_L in the control-flow graph of L
5.              Calculate Path Condition Expression (PCE_BB) for BB_L
6.              PCE_is_invariant = TRUE
7.              FOR each variable v in PCE_BB
8.                  IF( NOT Is_Invariant( v, L ) )
9.                      PCE_is_invariant = FALSE
10.             IF ( PCE_is_invariant )
11.                 FOR each shared variable x_BB, accessed in BB_L
12.                     IF ( x_BB is an array variable )
13.                         IF( NOT Is_Array_Invariant(x_BB, L) )
14.                             Add x_BB to NonRepVar_L
15.                     ELSE IF( x_BB is NOT a scalar variable )
16.                         Add x_BB to NonRepVar_L
17.             ELSE
18.                 IF ( the only loop-variant variable in PCE is the loop index of L)
19.                     Annotate PCE to all the shared variables accessed in BB_L
20.                     GOTO Line 11
21.                 ELSE
22.                     FOR each shared variable x_BB, accessed in BB_L
23.                         Add x_BB to NonRepVar_L
24.         RepVar_L = SharedVar_L - NonRepVar_L
25. END
26. Boolean Is_Array_Invariant ( array X, loop L )
27.     FOR each variable v in the index expression of array X
28.         IF( v is a scalar variable )
29.             IF ( NOT Is_Sequence_Invariant ( v, L ) ) return FALSE
30.         ELSE IF( v is an array variable )
31.             IF( NOT Is_Array_Invariant (v, L ) ) return FALSE
32.         ELSE
33.             return FALSE
34.     return TRUE
35. END
```

Figure 1. Repetitive Data Reference Analysis [a simplified version for extended abstract]

(1) The reference must be made in a basic block that executes repeatedly in a sequence of iterations of L and (2) the involved variable must either be scalar or an array whose subscript expressions are *sequence invariant* in L. To check the first criterion, the algorithm determines the *path conditions* [4] for the basic block and tests for loop invariance in L of these conditions. The gist of checking the second condition is in testing sequence invariance of all array subscripts. Sequence invariance of an expression means that the expression assumes the same sequence of values in each iteration of L. (Invariance is a simple case of sequence invariance – the sequence consists of one value.) The output of the compiler algorithm is the set of shared variables that incur repetitive data references across the iterations of L.

The algorithm proceeds as follows. First, it constructs an inter-procedural control-flow graph of the program and identifies loop structures. It also gathers all shared variables, *SharedVar*. Next, for each loop L it identifies the set *NonRepVar*, which are those variables that do *not* incur repetitive references. To this end, the algorithm determines the path condition expression (*PCE*) for each basic block *BB*. The path condition of a basic block *BB* describes the condition under which the control flow reaches *BB*. Essentially, all the branch statements of a program that reach *BB* have to be

examined to compute the path condition for *BB*. For example, in an `if`-statement with condition c, $c = true$ becomes part of the *PCE* of all basic blocks in the `then` block and $c = false$ becomes part of the *PCE* of all basic blocks in the `else` block. Path condition analysis uses methods introduced by others (e.g., [4]). Loop invariance of each variable in *PCE* is checked using the function *Is_Invariant(variable, loop)*, which builds on classical compiler methods.

Shared variables in basic blocks with loop-invariant *PCE* are handled as follows. Scalars and arrays with sequence-invariant subscripts incur regular accesses. All other variables (such as pointers and structs) are included in *NonRepVar*. The sequence invariance of a variable *v* is checked using the function *Is_Sequence_Invariant(v, L)*, which traces the use-def chains of *v* to test the invariance of the sequence of values that are assigned to *v* in L.

All shared variables in *BB*s with loop-variant path condition expressions are included in *NonRepVar* with the exception of loop-index-dependent *PCE*s, where the only loop-variant variable is L's index variable. For such *PCE*s the compiler is able to determine the iterations of L that execute the involved *BB*s. We will explain this case in more detail in Section 2.3. Finally *NonRepVar* is subtracted from *SharedVar*, resulting in the desired output of the algorithm.

Not shown in Figure 1 is a corner case that becomes relevant if we take parallel execution of loop L into consideration. If there is a critical section*C* in *L*, the order of processors entering that critical section varies from iteration to iteration, causing irregular communication behavior on otherwise regular accesses. Our algorithm also moves all shared variables accessed in critical sections to the *NonRepVar* set.

2.2 Compiler/Runtime System Interface

The compiler instruments the code to communicate its findings to the runtime system. It does this through an API containing two functions, which tell the runtime system when/where to learn communication patterns and when/where to repeat them, respectively. The overall model of program execution is important at this point. We assume that the program contains an outer, serial loop (e.g., a time-stepping loop), within which there are a sequence of parallel loops, each terminated by a barrier synchronization. This execution model is consistent with that of many OpenMP programs. We refer to this outer loop as the target loop. The compiler identifies the target loop as the outer-most loop containing parallel sections and a non-empty *RepVar* set. Next, it partitions the target loop body into *intervals* – a program section delimited by barrier synchronizations. The API includes the following functions, which the compiler inserts at the beginning of each interval.

– Get_CommSched(*CommSched-id*, *StaticVarList*)

– Run_CommSched(*CommSched-id*)

In Get_CommSched API, communication schedule identifier

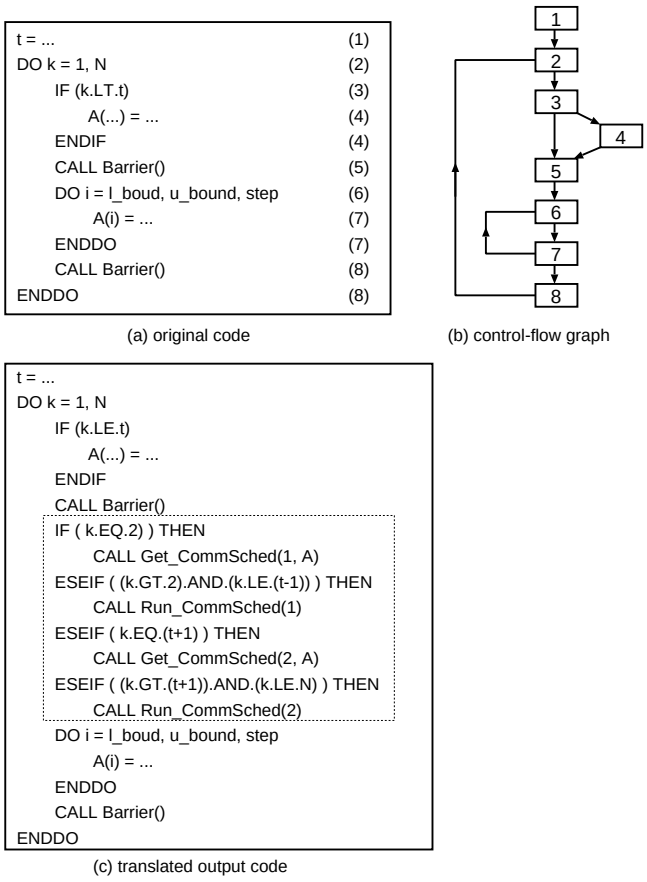

Figure 2. Compiler translation example

(*CommSched-id*) and *StaticVarList* are passed to the runtime system as parameters. *StaticVarList* is the list of shared variables with repetitive reference patterns, which will benefit from pro-active data movement at the beginning of the interval. *StaticVarList* is obtained by intersecting the set of shared variables accessed within the interval and *RepVar*, the set of shared variables with repetitive reference patterns described in Section 2. When Get_CommSched is invoked, the runtime system learns the communication pattern of the variables listed in *StaticVarList* and creates a communication schedule, which is the set of data (pages in case of page-based software DSM) that experienced remote memory access misses. On a call to Run_CommSched, the runtime system finds the communication schedule using *CommSched-id* and pro-actively moves the data according to that schedule.

2.3 Example

Figure 2 explains our algorithm by an example. Figure 2 (a) shows the OpenMP source code, annotated by basic block numbers. The corresponding control-flow graph is given in Figure 2 (b). The back edges in the control-flow graph corresponds to the two loops. We refer to the outer loop as *L1* and the inner loop as *L2*. By the algorithm in Figure 1, the path condition expression (*PCE*) for the ba-

sic block BB_4 is ($k < t$). *PCE* of BB_4 is a loop-index-dependent *PCE* because the only non-invariant variable in *PCE* is the index variable k of $L1$. From this information, the compiler recognizes that the control-flow will continuously follow the path from BB_3 to BB_4 to BB_5 when *L1* iterates from 1 to *t-1*. In BB_6, i (the index variable of an array A) is *sequence invariant* to $L1$ because the sequence of values that i will have in $L1$ do not change across the iterations of $L1$. Then, the compiler determines that a shared array *A* in loop $L1$ will exhibit two regular communication patterns during the ($1 <= k < t$) period and the ($t <= k <= N$) period, respectively.

Figure 2 (c) is the instrumented code after the translation where the statements inside the dotted box are inserted by the compiler to pass the compile-time knowledge to the runtime system. The phases of learning and applying the communication schedule are delayed by one iteration in the instrumented code, because the first iteration of each regular period tends to incur cold misses.

3 The Runtime System

3.1 The Base Runtime Shared Memory System

We use the TreadMarks [1] software DSM as the baseline software shared memory runtime system in our experiments. TreadMarks provides explicitly parallel programming primitives similar to those used in hardware shared memory machines. The system supports a lazy invalidate [10] version of a release consistent (RC) memory model. The virtual memory hardware is used to detect accesses to shared memory and the consistency unit is a virtual memory page.

3.2 The Augmented Runtime System

[*This section to be expanded in the final paper.*] We have modified the TreadMarks version 1.0.3.3 to support the pro-active data movement with message aggregation. The augmented runtime system captures the communication pattern and creates the communication schedule during the interval where *Get_CommSched* is called. On a call to *Run_CommSched*, the augmented runtime system applies the communication schedule by pro-actively moving data. When there are multiple messages to the same processor, those messages are aggregated into a single message to reduce the number of messages communicated. Also, the shared data that are pro-actively moved will not incur remote memory misses during the execution, which results in the reduction of DSM coherence overheads.

4 Results

We evaluated the combined compile-time/runtime system on a selection of SPEC OMP and NAS OpenMP benchmarks, exhibiting both regular and irregular communication patterns. Our commodity cluster consists of Pentium-II/Linux nodes, connected via standard 100Mbps Ethernet networks. We used five Fortran programs: WUPWISE, SWIM, and APPLU from the SPEC OMP benchmarks and CG from the NAS OpenMP benchmarks and SpMul. The OpenMP applications are translated into TreadMarks programs using the PCOMP compiler [13]. Among these programs, WUPWISE, SWIM, and APPLU are regular applications and CG and SpMul are mixed regular/irregular applications. We performed repetitive data reference analysis according to the proposed compiler algorithm and instrumented the programs with the described API functions for pro-active data movement. Overall, our applications show regular communication patterns in most of their execution, even in irregular program sections. For example, CG and SpMul have indirect array accesses. In both applications, the indirection arrays are defined outside each target loops and the compiler analysis is able to determine that the involved array accesses exhibit regular communication patterns. However, when our compiler analysis detects non-repetitive data references, such as array accesses with loop-variant subscript arrays or shared data accesses within critical sections in the applications, it identifies such accesses to have irregular communication patterns and does not apply pro-active data movement. Owing to the precise compiler analysis, we can selectively apply pro-active data movement to only shared variables that show regular communication patterns.

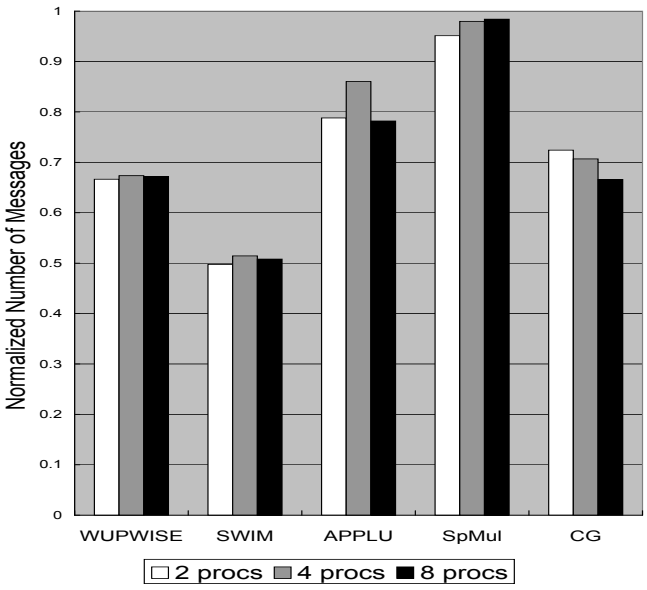

Figure 3. Number of messages of our compile-time/runtime system normalized to that of the baseline TreadMarks

We measured the number of messages generated during the execution of programs in the proposed compile-

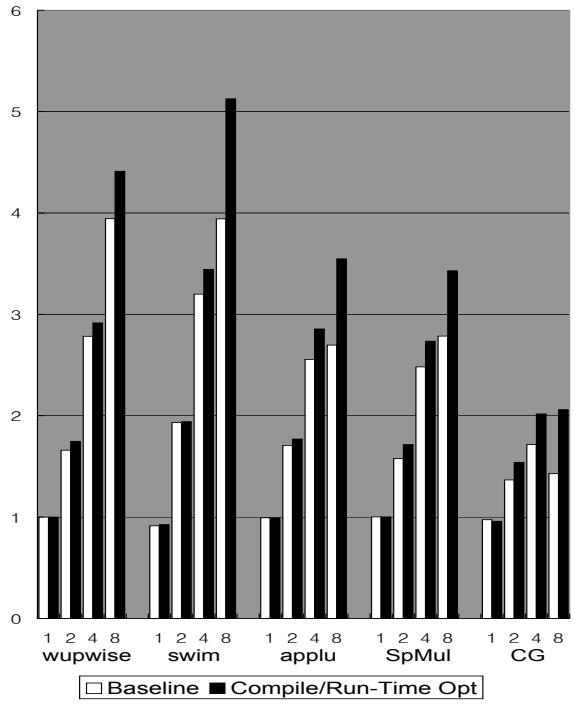

Figure 4. Speedup for TreadMarks and Compile-time/Run-time optimized version of TreadMarks

time/runtime system. Figure 3 illustrates the number of messages of the proposed system normalized to that of the baseline TreadMarks. As shown in the graph, our approach reduces the number of messages in the software DSM by up to 49% and on average by 27% on 8 processors. The reduction of the number of messages is achieved by the aggregation of communication messages. Figure 4 presents the performance of the baseline TreadMarks and that of the proposed compile-time/runtime system compared to the sequential execution times. Programs are executed on 1, 2, 4, and 8 processors. On 8 processors, the proposed technique achieves 28.1% performance improvement over the baseline TreadMarks system. The performance enhancement mainly comes from the reduction of the number of messages and the reduction of the page fault overhead. Our compiler analysis makes it possible to obtain these reductions by applying pro-active data movement to the right data at the right time.

5 Related Work

Based on how communication analysis is achieved, we categorize previous works into compile-time only, runtime only and compile-time/runtime integrated techniques.

Dwarkadas et al. [3] describe an integrated compile-time/runtime software DSM. Their work is a compile-time approach because the communication analysis is performed by the complier. Their compiler computes data access patterns for the individual processors and the runtime system uses this information to aggregate communication. Their communication analysis uses regular section descriptors to represent the regular array access as linear expressions of the upper and lower bounds along each dimension, and includes stride information. Therefore, the access patterns that can be analyzed are limited to linear expressions of array indices. Our compiler analysis does not have this restriction, as long as the expression shows repetitive data references.

Bianchini et al. [2] propose and simulated the Adaptive++ technique, a runtime only data prefetching strategy for software DSM. Their technique improves the performance of regular parallel applications by using the past history of memory access faults to adapt between repeated-phase and repeated-stride prefetching modes. Adaptive++ does not issue prefetches during periods when the application is not exhibiting one of these two types of behavior and is thus behaving irregularly. This method [2] learns communication patterns in *all* program sections throughout the program execution. Thus, this approach incurs avoidable overhead when there are communication pattern changes during the execution and even when the application has completely repetitive patterns because the runtime system has to continuously monitor and predict the communication patterns.

The third approach is a combined compile-time/runtime analysis of remote data communication. Viswanathan and Larus [14] showed how a parallel-language compiler helps a predictive cache coherence protocol to implement shared memory communication efficiently for applications with unpredictable but repetitive communication patterns. Their compiler uses data-flow analysis to detect remote shared memory accesses in parallel functions. When their compiler identifies that there will be communication in a parallel phase, it places directives to invoke a pre-send phase of the predictive protocol on all processors. Keleher et al. [11] [9] described a modified home-based protocol to selectively employ a hybrid invalidate/update coherence protocol. Their scheme uses update an protocol that pre-sends data by flushing updates at barriers when data are consistently communicated between the same set of processors and applies an invalidate protocol for the remaining data. Their compiler locates data that will likely be communicated in a stable pattern, then inserts calls to DSM routines to apply the flush operation. Since their compiler annotation does not guarantee that the communication pattern is static, there will be unnecessary data movement among processors when there are shared data with dynamic access pattern or a communication pattern changes during execution. Both Viswanathan and Larus [14] and Keleher et al. [11] [9] presented compiler analyses to detect remote references. However, their compilers do not identify repetitive patterns and thus cause runtime overheads in *all* program sections. Also, as compile-time analysis is intrinsically conservative, they do not capture remote accesses precisely. This contrasts with our precise runtime learning of such accesses, enabling further overhead re-

ductions.

Lu et al. [12] present a compiler and software DSM support for irregular applications. Their compiler algorithm identifies indirection array(s) and then the runtime system uses this information to determine the set of shared pages that each processor has accessed. These pages are aggregated into a single message, and prefetched prior to the access fault. Their runtime system traverses the indirection array to compute the list of pages to prefetch, which is a similar operation to the inspection phase of the inspector-executor models. Han and Tseng [7] described their work on irregular applications. In their approach, their compiler inserts inspectors and inspectors are executed only when there is a change to the communication pattern in order to amortize the overhead of inspectors. To detect such a change, their compiler examines global arrays accessed in the parallel loops and array subscripts containing indirection arrays are found and the indirection arrays are marked. Their compiler then examines the program to determine whether indirection arrays are modified within the time-step loop. Their compiler analysis achieves similar objective to ours in identifying static communication patterns for arrays accessed through indirection arrays.

In contrast to these two approaches, which target irregular applications, our method deals with both regular and irregular accesses in a unified framework.

6 Conclusions

We have presented a combined compile-time/runtime approach for accelerating the execution of applications with repetitive communication patterns. We have described an algorithm that detects shared data that incur repetitive accesses in both regular and irregular program sections. Our compiler algorithm is essential to accurately and selectively apply pro-active data movement to remote memory accesses showing static communication patterns. We evaluated the proposed compile-time/runtime system using OpenMP applications, consisting of both regular and irregular applications. We achieved performance improvements as significant as 44% and on average 28.1% on 8 processors.

Our work helps enable shared-memory programming methods on distributed-memory platforms. The ultimate goal is to combine high programmer productivity with cost-effective hardware solutions. While our programs do not yet perform as well as highly hand-tuned MPI programs, the presented work has brought us a step close to this goal.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R.Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] R. Bianchini, R. Pinto, and C. L. Amorim. Data prefetching for software dsms. In *the 12th international conference on Supercomputing*, pages 385–392, 1998.

[3] S. Dwarkadas, A. L. Cox, and W. 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*, pages 186–197, 1996.

[4] T. Fahringer and B. Scholz. Symbolic evaluation for parallelizing compilers. In *International Conference on Supercomputing*, pages 261–268, 1997.

[5] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication on multicomputers. In *1992 ACM International Conference on Supercomputing*, pages 357–367, Washington, D.C., 1992.

[6] M. Gupta, S. Midkiff, E. Schoenberg, B. Seshadri, D. Shields, K. Wang, M. Ching, and T. Ngo. An HPF compiler for the IBM SP-2. In *Proc. Supercomputing 95*. ACM Press, New York., San Diego, California, 1995.

[7] H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[8] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for fortran d on MIMD distributed-memory machines. In *Supercomputing*, pages 86–100, 1991.

[9] P. Keleher. Update protocols and iterative scientific applications. In *Proc. of the first Merged Symp. IPPS/SPDP (IPDPS'98)*, 1998.

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

[11] P. Keleher and C.-W. Tseng. Enhancing software DSMs for compiler-parallelized applications. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, 1997.

[12] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 48–56, 1997.

[13] S. J. Min, S. W. Kim, M. Voss, S. I. Lee, and R. Eigenmann. Portable compilers for openmp. In *International Workshop on OpenMP Applications and Tools (WOMPAT'01)*, pages 11–19, July 2001.

[14] G. Viswanathan and J. R. Larus. Compiler-directed shared-memory communication for iterative parallel applications. In *Supercomputing*, Nov. 1996.

[15] J. Zhu, J. Hoeflinger, and D. Padua. A synthesis of memory mechanisms for distributed architectures. In *Proceedings of the 15th international conference on Supercomputing*, pages 13–22. ACM Press, 2001.