International Journal of Parallel Programming, Vol. 31, No. 3, June 2003 (© 2003)

Optimizing OpenMP Programs on Software Distributed Shared Memory Systems

Seung-Jai Min,[1,2] Ayon Basumallik,[1] and Rudolf Eigenmann[1]

This paper describes compiler techniques that can translate standard OpenMP applications into code for distributed computer systems. OpenMP has emerged as an important model and language extension for shared-memory parallel programming. However, despite OpenMP's success on these platforms, it is not currently being used on distributed system. The long-term goal of our project is to quantify the degree to which such a use is possible and develop supporting compiler techniques. Our present compiler techniques translate OpenMP programs into a form suitable for execution on a Software DSM system. We have implemented a compiler that performs this basic translation, and we have studied a number of hand optimizations that improve the baseline performance. Our approach complements related efforts that have proposed language extensions for efficient execution of OpenMP programs on distributed systems. Our results show that, while kernel benchmarks can show high efficiency of OpenMP programs on distributed systems, full applications need careful consideration of shared data access patterns. A naive translation (similar to OpenMP compilers for SMPs) leads to acceptable performance in very few applications only. However, additional optimizations, including access privatization, selective touch, and dynamic scheduling, resulting in 31% average improvement on our benchmarks.

KEY WORDS: OpenMP applications; software distributed shared memory; benchmarks; performance characteristics; optimizations.

[1] School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana 47907-1285.

[2] To whom correspondence should be addressed. E-mail: smin@ecn.purdue.edu

1. INTRODUCTION

OpenMP[1] has established itself as an important method and language extension for programming shared-memory parallel computers. On these platforms, OpenMP offers an easier programming model than the currently widely-used message passing paradigm. Programs written in OpenMP can usually be parallelized stepwise, starting from a sequential program. OpenMP programs often resemble their original, sequential versions, the main difference being the inserted OpenMP directives. This approach contrasts with programs written in MPI,[2] for example, which generally need to be translated into parallel form as a whole, and which can look drastically different from their sequential versions.

While OpenMP has clear advantages on shared-memory platforms, message passing is today still the most widely-used programming paradigm for distributed-memory computers, such as clusters and highly-parallel systems. In this paper, we begin to explore the suitability of OpenMP for distributed systems as well. Our basic approach is to use a Software DSM (Distributed Shared Memory) system, which provides the view of a shared address space on top of a distributed-memory architecture. The primary contribution of this paper is to describe compiler techniques that can translate realistic OpenMP applications into this model, and to measure the resulting performance.

To this end, we have implemented a compiler that translates OpenMP programs into the Treadmarks Software DSM programs.[3] To achieve good performance, such a translation must do more than the usual transformations applied by OpenMP compilers for shared-memory platforms. The present paper describes this translation. Furthermore, we have studied a number of OpenMP programs by hand and identified optimization techniques that can improve performance significantly.

For our measurements, we used a commodity cluster architecture consisting of 16 PentiumII/Linux processors connected via standard Ethernet networks. We expect the scaling behavior of this architecture to be representative of that of common cluster systems with modest network connectivity. We have measured basic OpenMP low-level performance attributes via the kernel benchmarks introduced in Ref. 4 as well as the application-level performance behavior of several SPEC OMP benchmarks.[5]

Our work complements related efforts to implement OpenMP programs on distributed systems. Both language extensions and architecture support have been considered. Several recent papers have proposed language extensions. For example, in Refs. 6–8, the authors describe data distribution directives similar to the ones designed for High-Performance Fortran (HPF).[9] Other researchers have proposed page placement techniques to map data to the most suitable processing nodes.[6] In Ref. 10, remote procedure calls are used to employ distributed processing nodes. Another related approach is to use explicit message passing for communication across distributed systems and OpenMP within shared-memory multiprocessor nodes.[11] Providing architectural support for OpenMP essentially means building shared-memory multiprocessors (SMPs). While this is not new, an important observation is that increasingly large-scale SMPs continue to become commercially available. For example, recent reports of SPEC OMP benchmarks include systems up to 128 processors. (www.spec.org/hpg/omp2001/results/).

While most of these related efforts have considered language extensions for OpenMP on distributed systems, our goal is to quantify the efficiency at which standard OpenMP programs can be implemented using advanced compiler techniques. We have found that, while results from small test programs have shown promising performance, little information on the behavior of realistic OpenMP applications on Software DSM systems is available. This paper is meant to fill this void. It is organized as follows. Section 2 will present basic compiler techniques for translating OpenMP into Software DSM programs. Section 3 will discuss the performance behavior of such programs. Section 4 will present advanced optimizations, followed by conclusions in Section 5.

2. TRANSLATING OPENMP APPLICATIONS INTO SOFTWARE DSM PROGRAMS

2.1. Brief Overview of OpenMP Constructs and the Microtasking Model

At the core of OpenMP are *parallel regions* and *work sharing constructs*, which enable the program to express parallelism at the level of structured blocks within the program, such as loops and program sections. Work sharing constructs in OpenMP include the *OMP DO* loops in Fortran (correspondingly, in C/C++ there are *omp for* loops) and OpenMP *sections*. OpenMP directives for work-sharing constructs may include a scheduling clause, defining the way work will be mapped onto the parallel threads.

Currently, most OpenMP implementations use the Microtasking Model[12] which fits in well with the fork-join type of parallelism expressed by OpenMP. In this model, the *master processor* begins program execution as a sequential process. During initialization, the master creates *helper processes* on the participating processors. The helpers *sleep* until needed. When a parallel construct is encountered, the master wakes up the helpers and informs them about the parallel code to be executed and the environment to be setup for this execution. An important OpenMP property is that data is shared by default, including data with global scope and local data of subroutines called from within sequential regions. This can be implemented efficiently on today's *Symmetric Multi-Processors*, which support fully-shared address spaces and low communication latencies. A discussion on the overheads introduced by the OpenMP constructs may be found in Ref. 4.

2.2. Differences Between OpenMP on Shared and Distributed Memory

In the transition from shared-memory to distributed systems, the first major change is that each participating node now has its own private address space, which is not visible to other nodes. The Software DSM layer creates a shared data space, but the elements of this shared space now have to be explicitly allocated. This creates a challenge for the implementation of most OpenMP programs, where variables are shared by default. A second point of distinction is that the access times for this shared data space on distributed systems is much higher than those on an SMP. In previous work,(13) we have discussed the overheads for the different OpenMP constructs on such a network using a page based Software DSM system. The synchronization required to maintain coherence for the shared data is also more expensive on clusters as compared to SMPs. We have found that this coherency overhead often increases considerably with the size of the shared data space used. In Section 4, we will revisit these distinctions.

2.3. Automatic Translation using the Microtasking Model

As a first step, our compiler converts the OpenMP application to a microtasking form. Constructs for expressing parallelism, such as OMP PARALLEL regions and OMP PARALLEL DO loops are translated by extracting the parallel region or the loop body of the parallel loop into a separate *microtasking subroutine*. The extracted parallel region or loop body is replaced by a call to the corresponding *microtasking subroutine*.

When the program execution reaches these constructs, the master process calls a scheduling and dispatching runtime library, which in turn invokes *microtasking subroutine* on all helper processes. Besides this, the master process explicitly invokes a synchronization barrier for ensuring shared memory consistency before and after the parallel region. The OpenMP standard also has a set of four scheduling clauses which can be specified for parallel loops-static, dynamic, guided and runtime. Currently, our compiler supports only static and dynamic scheduling.

The fork-join overhead for the microtasking scheme is large in a distributed system, compared to the corresponding overhead on SMPs. The worker threads (in this case, the participating processes on the distributed nodes) spin-wait on a shared lock while they *sleep* and wait for work. In a Software DSM with a release consistency model, there needs to be a synchronization point after this lock is updated by the master process for the update to be visible to the worker processes. This synchronization adds to the fork overhead.

2.4. Shared Data Allocation and the Need for Inter-Procedural Analysis

The previous subsection described the translation of OpenMP control constructs. The next step is the conversion of OpenMP shared data into the form necessary for Software DSM systems. In case of SMPs, implementing OpenMP shared variables is straightforward, because all data is shared between threads by default. The translation only needs to decouple the variables that are explicitly declared as private. By contrast, in Software DSM systems, all variables are private by default, and shared data has to be explicitly allocated as such. The following issues arise when trying to do so by the compiler.

The first issue arises because OpenMP shared data may include subroutine-local variables, which reside on the process stack. Stacks on one process are not visible to other Software DSM processes as described in Section 2.2. The OpenMP compiler must identify these variables and change their declaration to an explicit allocation in shared space. In our Fortran implementation, the compiler places these variables into a shared common block. An implementation issue arises if the variables are dynamically sized, as the size of Fortran common blocks needs to be known at compile time. In our experiments, this problem has arisen in very few cases only, in which we have determined the maximal static variable sizes by hand. A better solution would be to allocate these variables from the shared heap.

A performance-related issue is that shared data intrinsically incur higher overhead than private data, as they are subject to the Software DSM paging and coherency mechanisms. Solutions that would conservatively allocate a large amount of data in shared space is thus not feasible. In our experiments, this issue has become more pressing as we have tried to implement realistic applications, such as the SPEC OMPM2001 benchmarks, which have large shared data set size, as opposed to small kernel programs. Therefore, we have chosen a strategy that carefully identifies the minimal set of OpenMP shared variables and allocates them in Software DSM shared space.

The algorithm for identifying and placing variables in shared space is presented in Fig. 1. All variables used within parallel regions, which are not explicitly declared by OpenMP directives to be private, must be identified as shared and allocated in the Software DSM shared space. For each such variable, the possible scenarios are:

(a) It is used within a parallel region and declared within the subroutine that contains the parallel region.

(b) It is used within a parallel region and passed as a parameter from another program unit which calls the subroutine that contains the parallel region.

(c) It is used in a subroutine that is called from within a parallel region.

The *THEN* part of the *IPA_ProgramUnit* block of our algorithm handles case (a). The *IPA_Up* block of our algorithm handles case (b). It traces upwards in the call graph to identify the first point of definition of a shared variable that is passed as a parameter to a subroutine containing parallel regions. The *IPA_Down* block handles case (c). It identifies the subroutines called from within a parallel region and recursively traverses their call chains to identify all variables used within these subroutines.

To implement blocks *IPA_Up* and *IPA_Down*, we need to perform Inter-Procedural Analysis. We assume the availability of an appropriate compiler infrastructure, including a subroutine call graph, and use/def sets for each program unit. In Fig. 1, *UseSet(PU)* and *DefSet(PU)* denote the set of variables used and defined respectively, in the program unit PU. *ParamSet(PU)* is the set of variables in the parameter list of PU and *LocalSet(PU)* is the set of local variables in PU.

3. PERFORMANCE MEASUREMENTS

In this section, we describe and discuss initial measurements carried out on the performance of OpenMP kernels, microbenchmarks and real-application benchmarks on our cluster. The performance of the kernel program provides an upper bound for the performance of OpenMP applications on our system. We have used the microbenchmarks to quantify the performance of specific OpenMP constructs. We then examine the performance of realistic OpenMP applications in two step. First, we examine the performance of a representative OpenMP application and consider the detailed behavior of different program sections. Second, we present the baseline performance of our OpenMP-to-Software DSM translation scheme across the set of SPEC OMP benchmarks.

```
// The driver routine for the IPA algorithm to recognize shared data
IPA_Driver
    G = {};    // G is the set of all shared global variables
    FOREACH ProgramUnit PU
        call IPA_ProgramUnit(PU);

// For every program unit, IPA algorithm is applied
IPA_ProgramUnit (ProgramUnit PU)
    IF ( PU is a microtasking subroutine )
    THEN    // it contains no nested OpenMP parallel constructs
        A = {};    // initialize a temporary shared variable set A
        A = UseSet(PU) ∪ DefSet(PU);
        A = A - ParamSet(PU) - LocalSet(PU);
        find the set P of private variables defined by the OpenMP directive for PU;
        A = A - P;
        G = G ∪ A;
        FOREACH subroutine PU_S called from within PU
            call IPA_Down(PU_S);
    ELSE
        // if this program unit contains an OpenMP parallel construct, then it has been replaced
        // by a call to the corresponding microtasking subroutine
        FOREACH Call Statement S to a microtasking subroutine
            FOREACH parameter p in the parameter list of S
                IF ( p belongs to the ParamSet(PU) ) THEN
                    let param_loc be the position of p in PU's parameter list;
                    call IPA_Up(PU, param_loc);

// recursively add variables of non-local scope in subroutines called from within parallel regions
// to the set G
IPA_Down(ProgramUnit PU)
    B = {};    // initialize a temporary shared variable set B;
    B = UseSet(PU) ∪ DefSet(PU);
    B = B - ParamSet(PU) - LocalSet(PU);
    G = G ∪ B;
    FOREACH subroutine PU_C called from within PU
        IPA_Down(PU_C)

// trace up a shared variable passed as parameter to the point where it is first defined and add it to G
IPA_Up(ProgramUnit PU,  int param_loc)
    FOREACH ProgramUnit Caller_PU that calls PU
        FOREACH Call Statement S in Caller_PU that calls PU
            let p be the parameter at position param_loc in the parameter list of S;
            IF ( p belongs to the ParamSet(Caller_PU) ) THEN
                set param_loc to the position of p in Caller_PU's parameter list;
                call IPA_Up(Caller_PU, param_loc);
            ELSE
                G = G ∪ {p};
```

Fig. 1. Inter-procedural analysis algorithm for recognizing shared data: In our implementation, the resulting set of shared variables, *G*, is placed in shared space by including these variables in shared common blocks.

3.1. Speedup Bounds: Performance of an OpenMP Kernel

In order to test the basic capability of scaling parallel code, we measured the performance of a simple OpenMP kernel, which is a small, highly parallel program that calculates the value of π using the integral approximation $\int_0^1 \frac{4.0}{(1.0+x^2)} dx$. The OpenMP constructs used within the program include a single OMP DO with a REDUCTION clause. The execution times and speedups obtained (shown in Fig. 2) provides an upper bound on the performance we can expect for our system. This kernel makes use of only three shared scalar variables. Thus the coherence actions of the

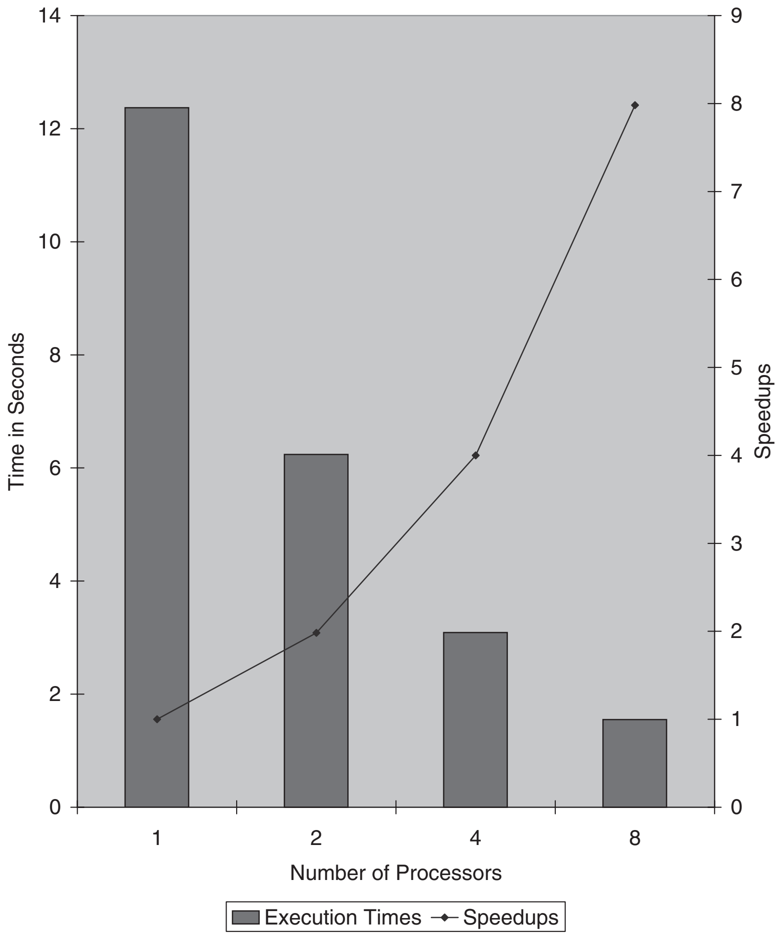

Fig. 2. Execution time and speedup of the *PI* kernel on 1, 2, 4, and 8 processors.

Software DSM system at the barrier terminating the parallel region are minor. This point will assume significance when we compare the performance of this kernel to real applications.

3.2. Performance of the OpenMP Synchronization Microbenchmark

In order to understand the overheads incurred by the different OpenMP constructs on our system, we have used the kernel benchmarks introduced in Ref. 4. The overheads of our Software DSM are exhibited clearly by the Synchronization Microbenchmark, shown in Fig. 3.

The trends displayed in Fig. 3 look similar to those for the NUMA machines (such as the SGI Origin 2000) enumerated in Ref. 4. This is

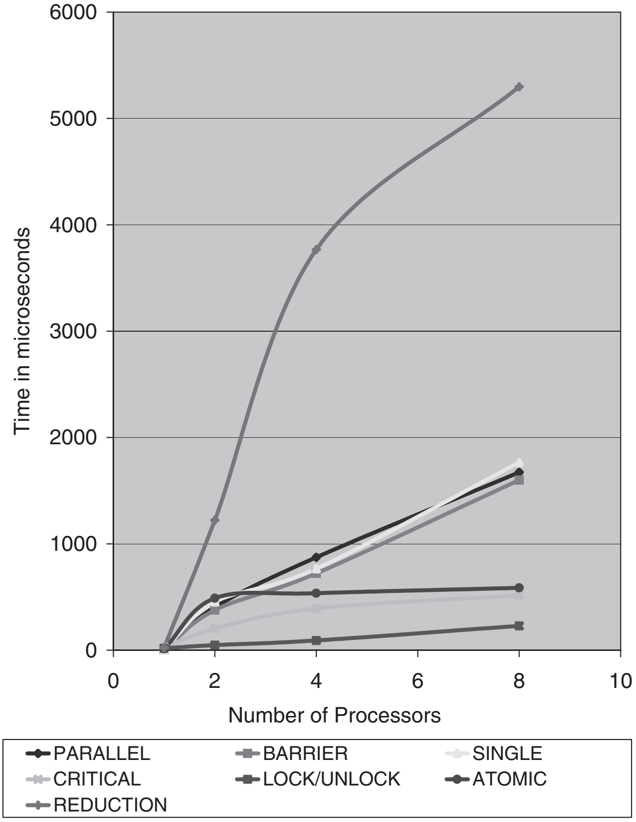

Fig. 3. Overheads of OpenMP synchronization constructs, as measured by the OpenMP synchronization microbenchmark. The overheads have been measured on a system of 1, 2, 4, and 8 processors.

consistent with the fact that a NUMA machine is conceptually similar to a Software DSM system. However, for a Software DSM system, the overheads are now in the order of milliseconds as compared to microseconds overhead in NUMA SMPs. The *barrier* overhead provides an indication of the potential loss of performance incurred by shared-memory coherence activities. Coherence overheads usually grow with the increase in shared memory activity within parallel constructs in a Software DSM. This fact is not captured here, because the Synchronization Microbenchmark, like the *PI* kernel benchmark, uses a very small amount of shared memory. To quantitatively understand the behavior of OpenMP programs that utilize shared data substantially, we next examine the performance of the SPEC OMPM2001 benchmark suite on our system.

3.3. Performance Evaluation of Real Application Benchmarks

The SPEC OMPM2001 suite of benchmarks(5) consists of realistic OpenMP C and Fortran applications. We performed a detailed instrumentation of these applications in order to measure the performance as well as the overheads incurred in the different program sections. The trends in our performance measurements were consistent for these applications. For brevity, we will limit the discussion of detailed measurements to the *EQUAKE* benchmark, which is a C application from the SPEC OMPM 2001 suite.

3.3.1. Overall Execution Time

Figure 4 shows the execution times for the *EQUAKE* benchmark run, using the *train* data set. For each processor, the total elapsed time has been expressed as a sum of the user and the system times.

The figure shows a speedup of *EQUAKE* in terms of user times from one to eight processors. However, considering total elapsed time, the overall speedup is much less. For eight processors, the performance degrades so that no speedup is achieved. This fact is consistent with the system time component, which grows from the serial to the eight-processor execution. We verified that this system time component is not caused by system scheduling or paging activity. Instead, we attribute the system time to shared-memory access and coherence activities.

3.3.2. Detailed Measurements of Program Sections

We instrumented each program section to separately measure the time spent in useful computation (*Loop Time*) and the time spent in waiting at the barrier (*Barrier Time*). The barrier time provides information about two effects: (a) The load imbalance caused by read accesses to shared data

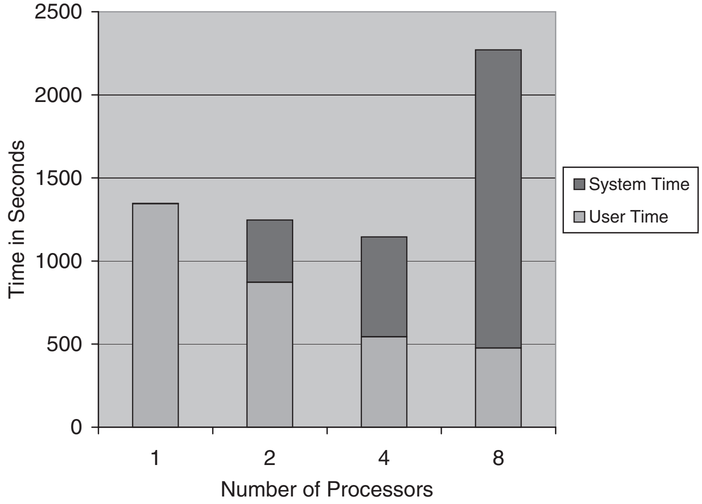

Fig. 4. Baseline execution times for the *equake* SPEC OpenMP benchmark running on 1, 2, 4, and 8 processors, shown in terms of the measured user times and system times.

that the Software DSM must retrieve from a remote node, and (b) The overhead caused by the shared-memory coherence actions that must be performed at the barrier.

The *EQUAKE* code has two small serial sections and several parallel regions, which cover more than 99% of the serial execution time. We first discuss two parallel loops, *main_2* and *main_3*, which show the desirable speedup behavior. The loop time and the barrier time are shown separately. Loop *main_3* is one of the important parallel loops, consuming around 30% of the program time. Figure 5 shows the behavior of these loops.

Loop *main_2* shows a speedup of about two and loop *main_3* shows almost linear speedup on eight processors in terms of elapsed time. In both cases, the loop time itself speeds up almost linearly, but the barrier time increases considerably. In fact, *main_2* has a considerable increase in barrier overheads from the serial to the eight processor execution. In *main_3*, the barrier overhead is within a more acceptable range. The reason is that the loop body of *main_2* is very small. By comparison, *main_3* has a larger loop body, which amortizes the cost of the barrier. The comparatively small barrier time in *main_3* mainly reflects the slight load imbalance caused by data being accessed from remote nodes.

We next look at the performance of the serial sections (*serial_1* and *serial_2*) and the most time-consuming parallel region (*smvp* $_0$) within the program. Figure 6 shows the behavior of these regions. In our system, the

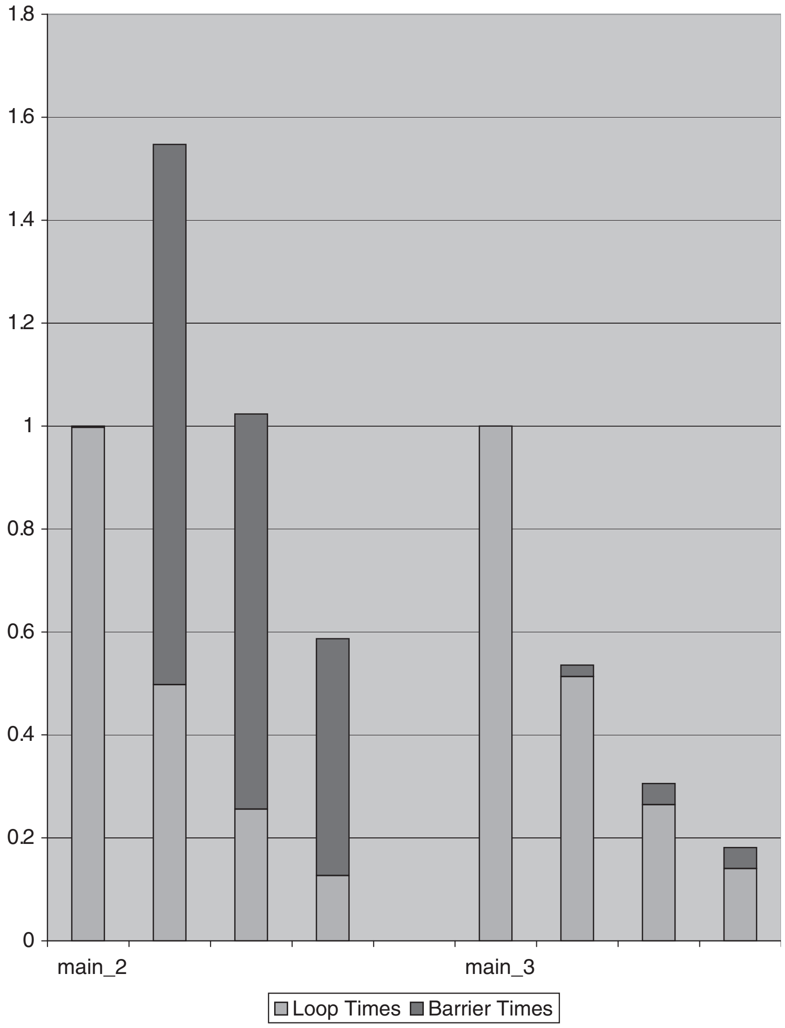

Fig. 5. Normalized execution times for the parallel loops, *main_2* and *main_3*, on 1, 2, 4, and 8 processors. The total execution time for each is expressed as a sum of the time spent within the loop and the time spent on the barrier at its end. Timings for each loop are normalized with respect to the serial version.

performance of the serial section may be affected as well, when executing in parallel. This is because a serial section may contain read references for which the data was last written by a remote node, or may contain several write references to shared memory, which will increase the coherence overhead at the barrier. As previously discussed in Section 2.3, barriers

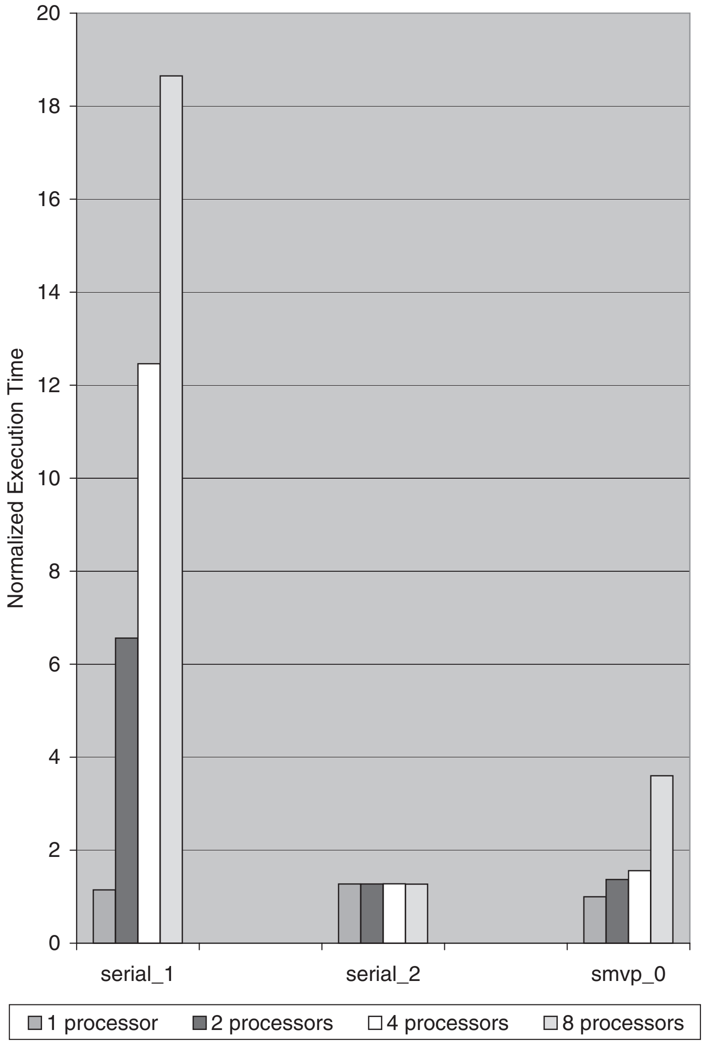

Fig. 6. Normalized execution times for the serial sections *serial_1*, *serial_2* and the parallel region *smvp_0*. For each region the times are normalized with respect to the serial version.

have to be placed at the end of a serial region, so that shared memory write operations by the master processor become visible to the other processors. Figure 6 shows that these factors result in serial section *serial_1* experiencing a slowdown, with growing number of processors. Section *serial_2* is relatively unaffected. The parallel region *smvp_0* also suffers from the same effect—it contains several parallel loops and after each of these, a barrier has to be placed. The parallel region *svmp_0* takes up more than 60% of the total execution time, hence its performance has a major impact on the overall application.

To summarize our measurements, we note that a naive transformation of realistic OpenMP applications for Software DSM execution does not give us the desired performance. A large part of this performance degradation is owing to the fact that shared-memory activity and synchronization is more expensive in a Software DSM scenario and this cost is several orders of magnitude higher than in SMP systems. The underlying Software DSM system that we have used implements *Lazy Release Consistency*(14) at the page level. At every synchronization point, the participating processes exchange information about which node has written to which pages in the shared memory since the last synchronization. If one node has written to a page in shared memory, the page is marked invalid on all other nodes. When a node tries to read from a page that has been marked invalid, it requests updates from all the nodes that have written to the page before the last synchronization point. Thus, barrier costs now increase with the number of shared memory writes in parallel regions. Shared memory reads may become very expensive depending upon which node last wrote the required data item. Realistic applications use a large shared-memory space and display rich and complex access patterns. The resulting high data coherence overheads explain the seemingly contradicting performance behavior of kernels and real applications. Having examined a representative application in detail, we next provide the results of our performance evaluation across several applications in the SPEC OMPM2001 suite.

3.4. Performance of the Baseline OpenMP Translation on Software DSM

Using our compiler, we translated four SPEC OMPM2001 Fortran programs and evaluated the performance on a commodity cluster consisting of PentiumII/Linux nodes, connected via standard Ethernet networks. Figure 7 shows the resulting speedups. These benchmark programs are known to exhibit good speedups on shared memory systems.(15) However, their performance on Software DSM systems shows different behavior. The programs WUPWISE and MGRID exhibit speedups, whereas the performance of SWIM and APPLU degrades significantly as the number of nodes increases. Evidently, porting well-performing shared-memory programs to Software DSM does not necessarily lead to uniformly good efficiency. A better understanding of the causes of performance degradations is necessary. In Section 4 we will analyze the causes for performance degradation

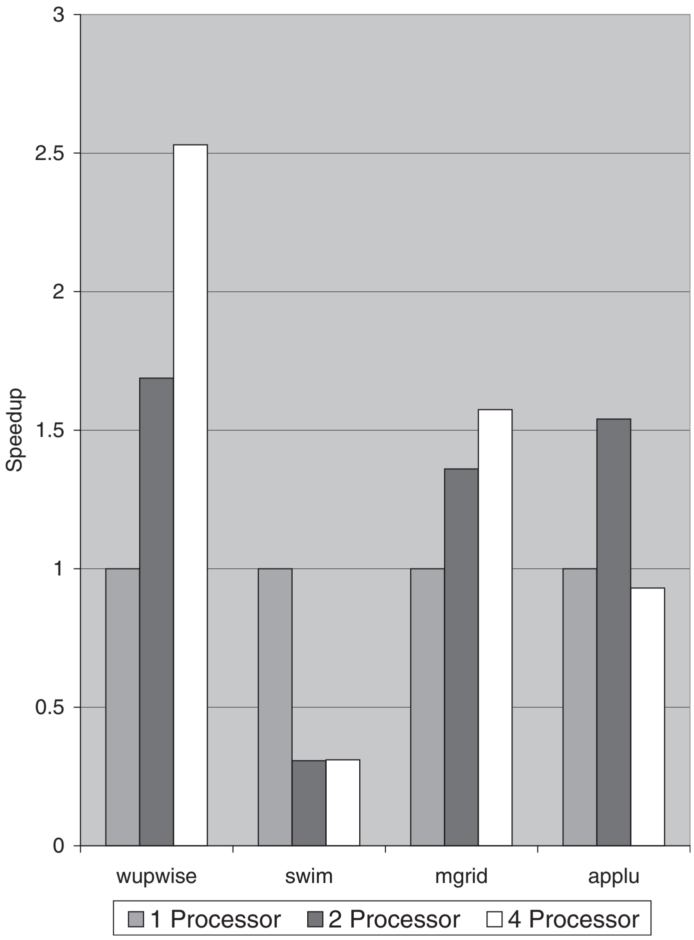

Fig. 7. Performance of the Fortran benchmarks in SPEC OMPM2001 after baseline translation.

further and propose optimization techniques for OpenMP programs on Software DSM system.

4. ADVANCED OPTIMIZATIONS

The baseline translation of SPEC OMPM2001 Fortran programs, described in Section 3.4, shows that shared-memory programs require optimization in order to get a desirable performance on Software DSM. Software DSM implementations have shown acceptable performance on kernel programs.(16) However, kernels differ from realistic shared memory applications in two essential ways: (1) in terms of the size of the shared-data space and (2) in terms of access patterns for the shared data.

As described in Section 3.3.2, as the size of shared data is increased, we observed that the coherence and update traffic increased as well. This effect is not brought out by kernel programs, which use a very small amount of shared data. Typical realistic shared memory applications, such as the SPEC OMPM2001 applications, may have data sets that are in the order of gigabytes. Large data sections are placed in the shared address space, which significantly affects this message traffic.

Secondly, a realistic application typically consists of several algorithms that access the shared data in different ways. These access pattern may result in complex message patterns in the underlying Software DSM layer that are expensive from a performance viewpoint. Kernel programs do not exhibit these communication patterns of full-sized applications and thus do not bring out these incurred costs. When a kernel program does use a large amount of shared data, it may actually exaggerate the effect of locality.

To address the above issues, we have implemented optimizations that fall into three categories.

- Reduction of shared data space through privatization.
- Improving locality through selective data touch.
- Overlapping computation and communication through dynamic scheduling and barrier elimination.

4.1. Privatization Optimizations

This optimization is aimed at reducing the size of the shared data space that must be managed by the Software DSM system. Potentially, all variables that are read and written within parallel regions are shared variables and must be explicitly declared as such. However, we have found that a significant number of such variables are read-only within the parallel regions. Furthermore, we have found that, for certain shared arrays, the read and write references access disjoint parts of the array from different nodes and a node only reads from the region of the array that it writes to. We refer to these variables as *single-owner* data. In the context of the OpenMP program, these are shared variables. However, in the context of a Software DSM implementation, instances of both can be privatized with certain precautions.

Privatization of the read-only shared variables is accomplished by the following steps

1. The variables belonging to this class are recognized as those that are read and not written within any parallel regions.
2. The statements in the serial regions that define (write or update) these variables are marked to be executed redundantly on all nodes. These updates need not be followed by any barrier synchronization since the changes are being made to private variables only. However, they must be preceded by a barrier if other shared variables need to be read for these updates.

Single-owner data is transformed to private arrays on each node. The size of each private instance is set to the size of the partition accessed by each node in the original program. This case is rather common in the OpenMP C codes, which allocate certain arrays as temporaries, or scratch spaces on each node.

The first benefit of privatization stems from the fact that access to a private variable is typically faster than access to a shared variable, even if a locally cached copy of the shared variable is available. This is because accesses to shared variables need to trigger certain coherency and book-keeping actions within the Software DSM. Privatization thus reduces the access time of many memory references compared to the original references. The overall coherency overhead is also reduced because coherency has to be now maintained for a smaller shared data size.

An additional important benefit of privatization is the effect on eliminating false sharing. Consider a program executing on two nodes A and B in a page-based Software DSM system. Now, consider a page that contains a truly shared variable X as well as a shared array Y. Assume Y is used as a temporary, thus it is single-owner data and would be privatizable by our method. Node A writes to its scratch space in Y in a parallel loop, in which both nodes also write to X. In a later parallel loop, A tries to read from its scratch space in Y, but it finds that this page has been written also by node B (since both nodes wrote to X). By the Software DSM coherence mechanism, the access to Y has to wait for an update from the other node B, though A is actually trying to read only the part of Y that it wrote to. This is a form of false sharing, which is eliminated by privatizing Y.

4.2. Selective Data Touch

Page-based Software DSM systems implement consistency by exchanging information at the page level. Between synchronization points the participating nodes exchange information about which nodes wrote into each page. A node that writes to a page in shared memory thus becomes the temporary *owner* of that particular page. The way this ownership changes during the program may significantly affect the execution time for the application. We will describe two cases where this effect significantly degrades the performance.

The first example code excerpt is from subroutine CALC2 of the SWIM program. In Fig. 8, there is an OpenMP parallel loop followed by a serial loop. The nodes mostly access pages they have themselves written in the parallel loop, and for these accesses they need not request updates from other nodes. Then the master node writes a single row of each shared array in the serial loop. If the array is allocated column-major, this could result in an invalidation of pages occupied by the array on all other nodes. When the array is accessed subsequently by the other nodes, all of them would have to request updates from the master node. This would incur substantial overhead, even though the actual array elements requested may already reside on the requesting node. The described scenario is also an example of false-sharing, due to the page-level granularity at which the Software DSM maintains coherence.

The second example code is from subroutine RHS of APPLU, shown in Fig. 9. In the first parallel loop, each node will write to the part of the shared array *rsd* partitioned according to the index *k*. However, in the second parallel loop each node will read the shared array *rsd* using index *j*. This access pattern change will incur a large number of remote node requests in the second parallel loop.

The manner in which shared data is read and written thus makes a considerable difference in the execution time. In this respect, though the access pattern for shared data may not impact SMP systems much, it may degrade performance significantly for a distributed system. To avoid inefficient access patterns, the program needs to be selective about which nodes

```
!$OMP PARALLEL DO                      !$OMP PARALLEL DO
      DO 200 J=1,N                           DO 200 J=1,N
      DO 200 I=1,M                           DO 200 I=1,M
      UNEW(I+1,J) = ...                      UNEW(I+1,J) = ...
      VNEW(I,J+1) = ...                      VNEW(I,J+1) = ...
      PNEW(I,J)   = ...                      PNEW(I,J)   = ...
  200 CONTINUE                           200 CONTINUE
!$OMP END PARALLEL DO                  !$OMP END PARALLEL DO

                                       !$OMP PARALLEL DO
      DO 210 J=1, N                          DO 210 J=1, N
      UNEW(1,J) = UNEW(M+1,J)                UNEW(1,J) = UNEW(M+1,J)
      VNEW(M+1,J+1) = VNEW(1,J+1)            VNEW(M+1,J+1) = VNEW(1,J+1)
      PNEW(M+1,J) = PNEW(1,J)                PNEW(M+1,J) = PNEW(1,J)
  210 CONTINUE                           210 CONTINUE
                                       !$OMP END PARALLEL DO


     (a)original code          (b)after selective touch optimization
```

Fig. 8. Selective data touch: Subroutine CALC2 code from SWIM.

```
!$OMP PARALLEL DO
      DO k=2, nz-1
       DO i=ist, iend
        DO j=jst, jend
         DO m=1, 5
          rsd(m,i,j,k)=...
          ...
         ENDDO
        ENDDO
       ENDDO
      ENDDO
      ...
!$OMP END PARALLEL DO

!$OMP PARALLEL DO
      DO j=jst, jend
       DO i=ist, iend
        DO k=2, nz-1
         DO m=1, 5
          rtmp(m,k)=rsd(m,i,j,k)-...
          ...
         ENDDO
        ENDDO
       ENDDO
      ENDDO
      ...
!$OMP END PARALLEL DO
```

(a) original code

```
!$OMP PARALLEL DO
      DO k=2, nz-1
       DO i=ist, iend
        DO j=jst, jend
         DO m=1, 5
          rsd(m,i,j,k)=...
          ...
         ENDDO
        ENDDO
       ENDDO
      ENDDO
      ...
!$OMP END PARALLEL DO

!$OMP PARALLEL
      DO j=jst, jend
       DO i=ist, iend
!$OMP DO
        DO k=2, nz-1
         DO m=1, 5
          rtmp(m,k)=rsd(m,i,j,k)-...
          ...
         ENDDO
        ENDDO
!$OMP END DO
       ENDDO
      ENDDO
      ...
!$OMP END PARALLEL
```

(b) after selective touch optimization

Fig. 9. Selective data touch: Subroutine RHS code from APPLU.

touch which portions of the data. For example, in the first case mentioned above, a single-row update in the serial region can be performed in parallel by the nodes "owning" the data. This consideration needs to be factored into the decision which parallel loops to execute as such. The tradeoff differs from that in SMPs, where small parallel loops may be executed serially to eliminate fork-join overheads. The second case can have a consistent access pattern across loops if the compiler partitions the inner *k*-loop instead of the outermost *j*-loop of the second loop nest.

4.3. Dynamic Scheduling of OpenMP Parallel Loops

The non-uniform access time for shared memory elements on a Software DSM system may result in a variance in the time required for otherwise balanced loop iterations.

Consequently, for several OpenMP loops, though the OpenMP application did not specify dynamic scheduling, we found it beneficial to have dynamic iteration scheduling for the Software DSM implementation.

The disadvantage of dynamic scheduling is the additional synchronization required for each iteration. However, we found that the overhead incurred is usually negligible when compared with the performance benefit obtained. Furthermore, in our experiments, this optimization was often useful in recognizing the need for the *selective touch* optimization mentioned above. A speedup through dynamic scheduling indicated that certain nodes, owing to the pattern of shared memory access in earlier parts of the program, waited considerable lengths of time to receive updates from other nodes.

4.4. Results

We applied the described optimizations by hand to several of the SPEC OMPM2001 benchmarks and achieved marked performance improvements. Figure 10 shows the final performance obtained by the baseline translation and subsequent optimization. We present the speedups for four Fortran codes (WUPWISE, SWIM, MGRID, APPLU) and two C benchmarks (ART, EQUAKE). The baseline translation for the Fortran programs was performed by our compiler and C applications were transformed by hand.

We found that for two of the Fortran codes, WUPWISE and MGRID, the baseline translation already had acceptable speedups, and so we did not apply further optimizations.

For the two other Fortran codes, our optimizations achieved improvements. In the case of SWIM, the baseline translation degraded in going from one to four nodes. By comparison, the optimized version shows a major improvement with speedups of 1.53 on two nodes and 2.14 on four nodes. APPLU still exhibits performance degradation in going from two to four nodes, but the degradation is less compared to the baseline code.

For both of these codes, *selective touch* optimizations were useful. In SWIM, the application of this transformation was by parallelizing an array update that was serial in the original code. For APPLU, we used work-sharing on an inner loop (though the outer loop was specified as the OpenMP parallel loop) to control the pattern in which writes occurred to the shared memory.

For the C applications, we found opportunities to apply all three optimizations. ART gained considerably after privatizing several arrays that were not declared as private in the original code. In ART, the main OpenMP parallel loop is already specified for dynamic scheduling. However, in EQUAKE, we derived a benefit from making certain parallel loops dynamically scheduled, though the original OpenMP directives did not specify this. For both these codes, the optimizations improved the

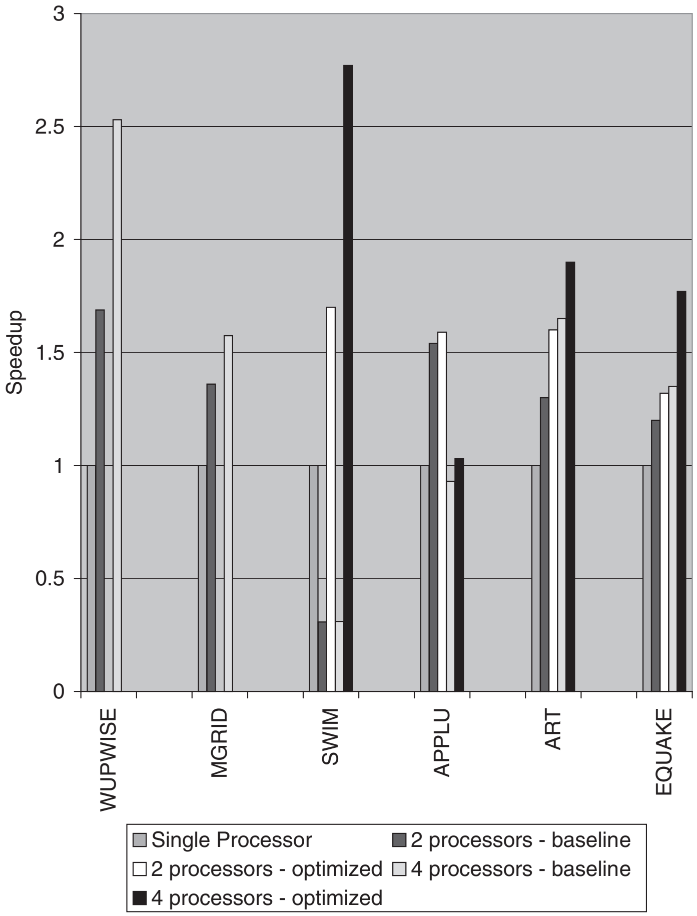

Fig. 10. Performance of four Fortran 77 and two C benchmarks from the SPEC OMPM2001 benchmark suite after baseline translation and optimization.

speedups on two nodes to almost match the baseline speedups on four nodes, and we achieved further speedups on four nodes.

4.5. Ongoing Work: Other Schemes for Improving OpenMP Performance on Software DSM Systems

In our ongoing project we are exploring transformations that can further optimize OpenMP programs on a Software DSM system. We expect the following techniques to have a significant performance impact. The realization of such a compiler is one objective of our ongoing project.

Data Prefetch and Forwarding. Prefetch is a fundamental technique for overcoming memory access latencies and has been shown to be useful in case of Software DSM systems.(17) Closely-related to prefetch is *data forwarding*, which is producer-initiated. Forwarding has the advantage that it is a one-way communication (producer forwards to all consumers) whereas prefetching is a two-way communication (consumers request data and producers respond). An important issue in prefetch/forwarding is to determine the optimal prefetch point. Prefetch techniques have been studied previously,(18) albeit not in the context of OpenMP applications for Software DSM systems. We expect prefetch/forwarding to significantly lower the cost of OpenMP END PARALLEL region constructs, as it reduces the need for coherence actions at that point.

Barrier Elimination. Two types of barrier eliminations will become important. It is well known that the usual barrier that separates two consecutive parallel loops can be eliminated if permitted by data dependences.(19) Similarly, within parallel regions containing consecutive parallel loops, it may be possible to eliminate the barrier separating the individual loops.

Page Placement. Software DSM systems may place memory pages on fixed home processors or the pages may migrate between processors. Fixed page placement leads to high overhead if the chosen home is not the processor with the most frequent accesses to this page. Migrating pages can incur high overhead if the pages end up changing their home frequently. In both cases the compiler can help direct the page placing mechanism. It can estimate the number of accesses made to a page by all processors and choose the best home.

Automatic Data Distribution. Data distribution mechanisms have been well researched in the context of distributed memory multiprocessors and languages such as HPF. Many of these techniques are directly applicable to a Software DSM system. Automatic data distribution is easy in regular data structures and program patterns. Hence, a possible strategy is to apply data distribution with explicit messaging in regular program sections and rely on Software DSM mechanism for other data and program patterns. This idea has been pursued in Ref. 20, and we will combine our approaches in a collaborative effort. Planned enhancements include other data and loop transformation techniques for locality enhancement.

Adaptive Optimization. A big impediment for all compiler optimizations is the fact that input data is not known at compile-time.

Consequently, compilers must make *conservative assumptions* leading to suboptimal program performance. The potential performance degradation is especially high if the compiler's decision making chooses between transformation variants whose performance differs substantially. This is the case in Software DSM systems, which typically possess several tunable parameters. We are building on a prototype of a dynamically adaptive compilation system,(21) called ADAPT, which can dynamically compile program variants, substitute them at runtime, and evaluate them in the executing application.

In ongoing work we are creating a compilation system that integrates the presented techniques. As we have described, several of these techniques have been proposed previously. However, no quantitative data of their value on large, realistic OpenMP applications on Software DSM systems is available. Evaluating these optimizations in the context of the SPEC OMP applications is an important objective of our current project.

Our work complements efforts to extend OpenMP with latency management constructs. While our primary emphasis is on the development and evaluation of compiler techniques for standard OpenMP programs, we will consider small extensions that may become critical in directing compiler optimizations.

5. CONCLUSIONS

In this paper, we have described our experiences with the automatic translation and further hand-optimization of realistic OpenMP applications on a commodity cluster of workstations. We have demonstrated that the OpenMP programming paradigm may be extended to distributed systems, such as clusters. We have discussed issues arising in automating the translation of moderate-sized OpenMP applications with an underlying layer of Software DSM and proposed solutions to these issues. We have also presented several optimizations in the context of the page-based Software DSM, which improve the performance for the studied applications considerably.

As part of a quantitative performance evaluation, we have found that a baseline compiler translation, similar to the one used for OpenMP programs on SMP machines, yields speedups for some of the codes but unacceptable performance for others. We then evaluated the performance after applying certain optimizations, such as access privatization, selective touch, and dynamic scheduling. We found that, across the board, these optimizations have a palpable benefit.

Currently, our optimizations are limited to source-level transformations on the applications within the scope of the API provided by the

Software DSM system. In the next phase of optimizations, we intend to adapt the Software DSM system to the needs of the compiler and applications. We also intend to couple our Software-DSM-based implementation and optimizations with hybrid message passing alternatives.[20] We anticipate that these unified approaches will enable commodity clusters to be used for a larger class of applications, as well as greatly simplify the program development for networked computing environments.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under Grants 9703180, 9975275, 9986020, and 9974976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. OpenMP Forum, OpenMP: A Proposed Industry Standard API for Shared Memory Programming, "www.openmp.org" Technical Report (October 1997).
2. Message Passing Interface Forum, MPI: Message passing interface standard, www.mpi-forum.org (1999).
3. S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, C. Amza, A. L. Cox, and W. Zwaenepoel, Treadmarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, **29**(2):18–28 (February 1996).
4. J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, *Proc. of the European Workshop on OpenMP (EWOMP '99)* (October 1999).
5. R. Eigenmann, G. Gaertner, W. B. Jones, V. Aslot, M. Domeika, and B. Parady, SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance, *Proc. of WOMPAT 2001, Workshop on OpenMP Applications and Tools*, Lecture Notes in Computer Science 2104, pp. 1–10 (July 2001).
6. R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson, J. Bircsak, P. Craig, and C. Offner, Extending OpenMP for NUMA Machines, *Proc. of the IEEE/ACM Supercomputing '2000: High Performance Networking and Computing Conference (SC'2000)* (November 2000).
7. V. Schuster and D. Miles, Distributed OpenMP, Extensions to OpenMP for SMP Clusters, *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT'2000)* (July 2000).
8. T. S. Abdelrahman and T. N. Wong, Compiler Support for Data Distribution on NUMA Multiprocessors, *J. Supercomputing*, **12**(4):349–371 (October 1998).
9. High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 1.0*, Technical Report CRPC-TR92225, Houston, Texas (1993).
10. M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi, OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP, *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001)* (July 2001).
11. L. Smith and M. Bull, Development of Mixed Mode MPI / OpenMP Applications, *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2000)* (July 2000).

12. M. Booth and K. Misegades, Microtasking: A New Way to Harness Multiprocessors, *Cray Channels*, pp. 24–27 (1986).
13. A. Basumallik, S.-J. Min, and R. Eigenmann, Towards OpenMP Execution on Software Distributed Shared Memory Systems, *Int'l. Workshop on OpenMP: Experiences and Implementations (WOMPEI'02)*, Lecture Notes in Computer Science 2327, Springer-Verlag (May 2002).
14. P. Keleher, A. L. Cox, and W. Zwaenepoel, Lazy Release Consistency for Software Distributed Shared Memory, *Proc. of the 19th Ann. Int'l. Symp. on Computer Architecture (ISCA'92)*, pp. 13–21 (1992).
15. V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance, *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001)*, Lecture Notes in Computer Science 2104, pp. 1–10 (July 2001).
16. C. Amza, A. L. 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).
17. A. Lo, T. Mowry, and C. Chan, Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory, *Proc. Fourth Int'l. Symposium on High-Performance Computer Architecture (HPCA'98)* (February 1998).
18. E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies, *Proceedings of ICS'90, Amsterdam, The Netherlands*, Vol. 1, pp. 342–353 (June 1990).
19. C. W. Tseng, Compiler Optimizations for Eliminating Barrier Synchronization, *Proc. of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'95)* (July 1995).
20. J. Zhu and J. Hoeflinger, Compiling for a Hybrid Programming Model Using the LMAD Representation, *Proc. of the 14th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC2001)* (August 2001).
21. M. J. Voss and R. Eigenmann, High-Level Adaptive Program Optimization with ADAPT, *Proc. of the Symposium on Principles and Practice of Parallel Programming* (2001).

Printed in Belgium