Performance Analysis of Compiler-Parallelized Programs on Shared-Memory Multiprocessors[†]

Seon Wook Kim Michael Voss Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285

Abstract

Shared-memory multiprocessor (SMP) machines have become widely available. As the user community grows, so does the importance of compilers that can translate standard, sequential programs onto this machine class. Substantial research has been done to develop sophisticated parallelization techniques, which can detect and exploit parallelism in many real applications. However, the performance of compiler-parallelized applications can be below expectations. The speedups of even fully-parallel codes on today's shared-memory multiprocessors can be significantly less than the number of processors. In this paper we will investigate reasons for such performance behavior. We will focus on three specific issues: (1) We will determine whether it is appropriate for the preprocessor to express the detected parallelism in the common loop-oriented form, (2) we will determine sources of inefficiencies in fully parallel SMP programs that exhibit good cache locality, and (3) we will discuss the portability of these programs across SMP machines.

In our experiments we have extended the Polaris compiler, so that it can generate thread-based code directly. We compare the performance of this code with Polaris' loop-parallel OpenMP output form and with architecture-specific directive languages available on the Sun Enterprise and the SGI Origin systems. We have analyzed in detail the performance of several parallel Perfect Benchmarks. Our main findings are as follows. (1) Overall, there is no significant performance disadvantage of the loop-parallel representation. (2) However, substantial performance differences are attributable to the *instruction efficiency*, which is influenced by the data sharing semantics of parallel constructs. (3) Both the OpenMP and the thread-based program forms are functionally portable, but can result in substantially different performance on the two machines.

[†]This work was supported in part by DARPA contract #DABT63-95-C-0097 and NSF grants #9703180-CCR and #9872516-EIA. This work is not necessarily representative of the positions or policies of the U. S. Government.

I. Introduction

A. *Trends in Parallelizing Compilers for SMP's*

With the increasingly wide availability of shared-memory multiprocessor servers and workstations, the importance of compilers that can translate standard, sequential programs onto this machine class is growing steadily. There exists a large body of research in parallelizing compiler techniques, ranging from detecting parallelism to efficiently executing parallel code on diverse machine organizations.

Automatic program parallelization has been most successful in Fortran code, as demonstrated by the Polaris and SUIF compilers [1, 2]. Substantial challenges exist when detecting parallelism in science and engineering applications that contain sparse data structures [3] and in C, C++, and Java programs, which contain pointers and irregular control flow [4, 5, 6].

B. *Performance of the Backend Compiler*

In this paper we consider an additional problem, which has been given less attention in the past. We have observed that even highly parallel programs that exhibit good cache locality can perform poorly on today's shared memory machines. For example, in the `TRFD` Perfect benchmark, two major loops consume 94% of the overall time when using a serial code. The loops can be fully parallelized and have negligible cache misses. Using Amdahl's law, we can expect a speedup of 3.3 on 4 processors. However, the best measurement we have obtained on a real machine was a speedup of 2.6.

These observations raise questions of the backend compilers' performance. For the purpose of this paper we consider a backend compiler the second compilation step, which follows the parallelizing preprocessing step. Typically, the preprocessor performs sophisticated program analyses and optimizations, while the backend applies more straightforward code generation techniques. The two compilers are usually not integrated. One approach to a higher integration has been taken by the Promis compiler [7], in which a common representation serves both compilation steps. Another approach is to pass frontend information to the backend compiler using a universal format [8].

In the present paper we address three specific questions related to the performance of backend compilers and their interface to parallelizing preprocessors.

1. *Is parallel loop semantics the right form for expressing parallelizing preprocessor output?* Most parallelizing preprocessors rewrite sequential loops into parallel loops. Could they perform better if they translated the source into a form closer to the object code? To answer this question, we have extended the Polaris compiler so that it generates thread-based code directly. Our results indicate that, overall, no significant loss is attributable to the loop-parallel representation. However, we have also found that the two translation methods can lead to significant performance differences in individual code sections. One reason for this is that parallel loop constructs do not give data dependence information, which can impact backend compiler optimizations.

2. *What are the sources of performance loss in parallel shared-memory programs?* Highly-parallel programs that do not perform well on SMP systems are often thought of as "not having enough locality." In our work we have analyzed a number of overhead factors that impact the program performance. We have found that both the synchronization time (such as the barrier time at the beginning and end of a parallel loop) and the *instruction efficiency* (a code with higher instruction efficiency executes fewer instructions than a less efficient code) can make a significant difference.

3. *Is the output of a parallelizing preprocessor portable?* The Polaris compiler can generate output in both the OpenMP parallel language standard and the thread-based form. We have compared the portability of these two representations in terms of their performance on two different machines, a Sun Enterprise 4000 and an SGI Origin 2000 systems. We have found that, while functional portability is provided, there are significant performance differences of the same parallel program compiled with the respective backend compilers of the two machines.

The remainder of the paper answers these questions more quantitatively. It is organized as follows. Section II describes the experimental setup and overall program execution results that we have obtained on the Sun Enterprise and SGI Origin systems. Section III introduces MOERAE, the thread-based translator that we have implemented as part of our project. Section IV investigates the performance of the most time-consuming loops in each program in detail. Section V presents conclusions.

II. Experiment Setup and Overall Results

For our experiments we have implemented two Polaris *output passes* for generating OpenMP and thread-based programs, respectively. Our implementation of the MOERAE translator and microtasking library will be described in Section III. The test suite consists of five programs from the Perfect Benchmarks, which can be parallelized to a high degree by Polaris. We have translated these programs into OpenMP form, into thread-based MOERAE form, and – for comparison – into the native, machine-specific directive languages available on the target systems. All these translations were performed by Polaris. Figure 1 gives an overview of our compilation system. In our experiments we have measured the overall and the loop-by-loop performance on the Sun Enterprise 4000 and on the SGI Origin 2000 systems. We have also measured the serial (non-parallelized) program, which defines *speedup=1* in our performance results.

Figure 2 shows the speedups relative to the serial execution time of the original codes. The five programs: `TRFD`, `MDG`, `BDNA`, `ARC2D`, and `FLO52` from the Perfect Benchmarks suite were used in our evaluation [9, 10]. The codes were run on a Sun Enterprise 4000 (Solaris 2.5) and an SGI Origin 2000 (IRIX 6.4). The Sun Enterprise 4000 has six 248 MHz UltraSPARC Version 9 processors, each with a 16 KB L1 data cache and 1 MB unified L2 cache. The SGI Origin 2000 has 128 195MHz R10000 processors, each with a 32 KB L1 data cache and 4MB unified L2 cache. Each code was parallelized by Polaris and transformed into three

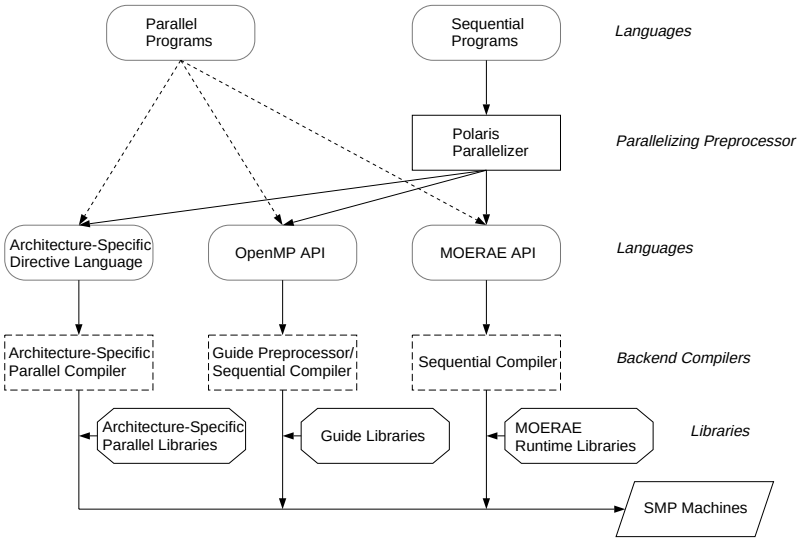

Figure 1: Language and Translator System Used in Our Machine Environment.

parallel forms: (1) using native, architecture-specific directives, (2) using OpenMP parallel loop directives, and (3) using the MOERAE scheme, respectively.

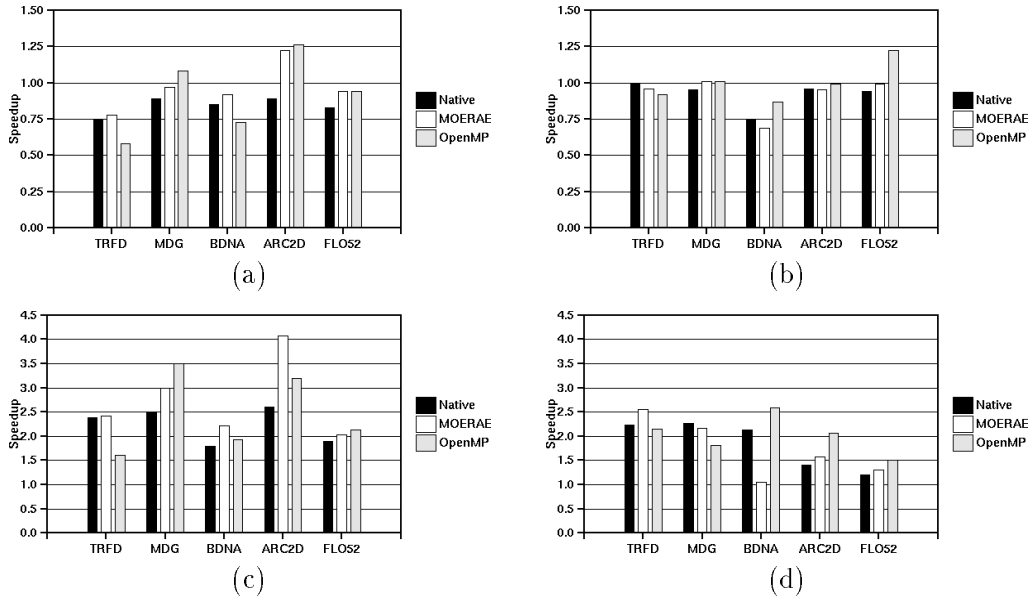

Figure 2: Speedup of Benchmarks as Executed on a Sun Enterprise 4000 and an SGI Origin 2000: (a) 1 processor speedup on the Enterprise, (b) 1 processor speedup on the Origin, (c) 4 processor speedup on the Enterprise and (d) 4 processor speedup on the Origin. All speedups are calculated with respect to the original sequential time on each machine.

On the Sun Enterprise and on the SGI Origin, the overall results show comparable

performance of the MOERAE and the OpenMP schemes. In only one case is the 4-processor performance of the machine-specific directive version the best, by a small margin. Although all three code variants exhibit the exact same parallelism, the individual numbers differ substantially. Furthermore, the speedups on the SGI system are generally lower than on the Sun Enterprise. However, we have measured the absolute performance on the SGI Origin to be on average 2.6 times higher than on the Sun Enterprise. Before we investigate these performance results in detail, we describe the new Polaris output pass and microtasking libraries, MOERAE, in the next section. In Section IV, we will then analyze the performance of the backend compilers, and Section V concludes the paper.

III. MOERAE: Portable, Thread-Based Interface

The MOERAE system expresses loop-level parallelism in Fortran programs. MOERAE consists of two major components: a compiler pass to translate sequential programs into a portable thread-based form, and a runtime library for managing these parallel threads. The program transformations are performed by a modified version of the Polaris [1] compiler. We have added a *postpass* to Polaris, which transforms parallel loops into subroutines and replaces the original loops with calls to a scheduler. At runtime, the scheduler dispatches the subroutines to the participating threads. The master thread executes the modified main program, and the child threads execute the newly-created subroutines, as orchestrated by the scheduler. Figure 3 illustrates this scheme.

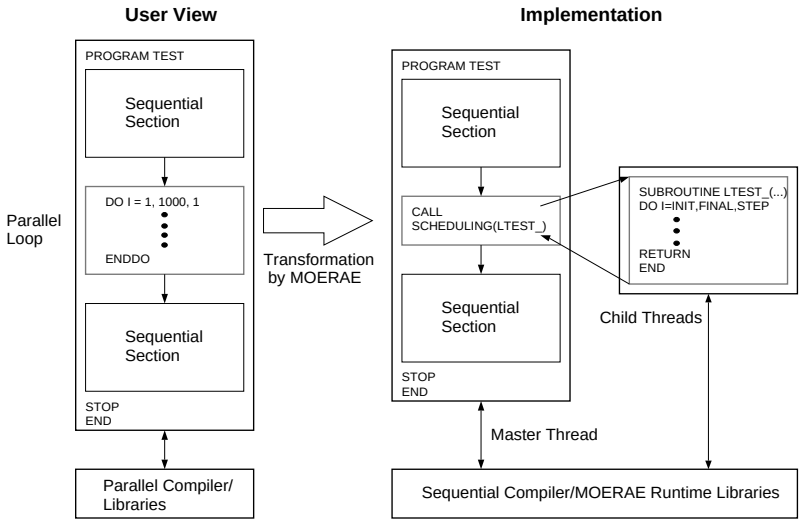

Figure 3: Overview of the MOERAE Translator and Runtime Libraries.

The key factor in achieving performance with MOERAE was a simple API consisting of 6 runtime functions for scheduling and dispatching parallel tasks (*initialize_thread*, *block_scheduling*, *interleaved_scheduling*, *lock*, *unlock*, and *num_threads*). This design contrasts with the rich functionality of the libraries used by the OpenMP implementation, and

has allowed us to provide efficient mechanisms. The runtime library uses a microtasking scheme using Pthread packages in order to reduce the overhead of creating threads each time a parallel section is encountered. That is, at the beginning of the program all of the threads are created (*initialize_thread*). Using spin-wait, the threads sleep during serial program sections and wake up for each parallel section. MOERAE supports blocked and interleaved scheduling schemes (*block_scheduling*, *interleaved_scheduling*). It provides lock/unlock functions to create critical sections (*lock*, *unlock*). For portability, the runtime library is implemented using the Pthreads package and is also available on Solaris threads. Our current implementation on the SGI IRIX 6.4 uses `sproc()` processes, because Pthreads are not yet fully supported.

IV. Detailed Performance Analysis

Our analysis is based on the measured loop-by-loop performance of the benchmarks on the Sun and the SGI machines. In addition, we used the ParaSim parallel program simulator [11], from which we obtained the number of instructions issued and cache hit ratios. The benchmarks are simulated using parameters comparable to the measured machines. As in Figure 2, all programs are measured in three different forms. They exploit the same parallelism (detected by the Polaris preprocessor) but differ in their representation and backend compilers. All programs are compiled with -O5 optimization on the Sun and -O3 on the SGI systems.

In our analysis we will focus on four potential overhead factors that prevent a parallel loop from having ideal speedup: (1) small parallel loops can be dominated by the barrier delays at loop start and end (fork/join overhead), (2) irregular or low numbers of iterations can cause load imbalance, (3) cache misses may increase when parallelizing a program, and (4) the instruction efficiency.

The term *instruction efficiency* needs introduction. A parallel code with low instruction efficiency has a higher number of executed instructions than the serial code. This can be due to additional instructions inserted by the parallelizer and/or more conservative code generation by the backend. Instruction efficiency has proven to be a useful measure in our performance analysis, although instruction execution times and pipeline stalls are not factored in. We will point out situations where these additional effects need to be considered.

A. *TRFD*

In `TRFD`, two loops `OLDA DO100`[1] and `OLDA DO300` take more than 94% of the total execution time in the serial code. As shown in Figure 4, the MOERAE thread-based program form outperforms the others on both Sun and SGI machines.

`TRFD` contains large, regular loops and the cache hit ratio of the parallel program is about 97%. Figure 5 shows the relative number of instructions of the code variants. There is about 30% overhead between the serial and the best parallel variant. This is consistent

[1] Our notation means the loop with label 100 in subroutine `OLDA`.

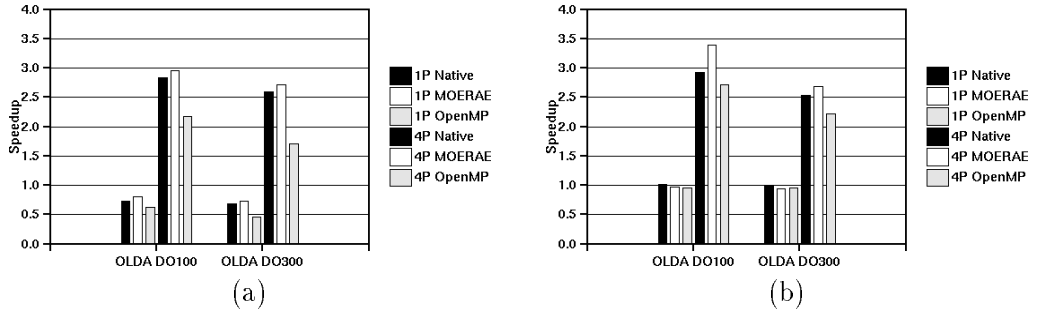

Figure 4: Speedup of the Major Loops in `TRFD` as Executed on (a) the Sun Enterprise 4000 and (b) the SGI Origin 2000.

with the fact that the benchmark has complex induction variables, whose substitution in the parallel code leads to significantly more complex expressions. The difference in executed instructions accounts for the difference in speedups of the code variants. In fact, factoring in the instruction efficiency models the obtained speedups closely.

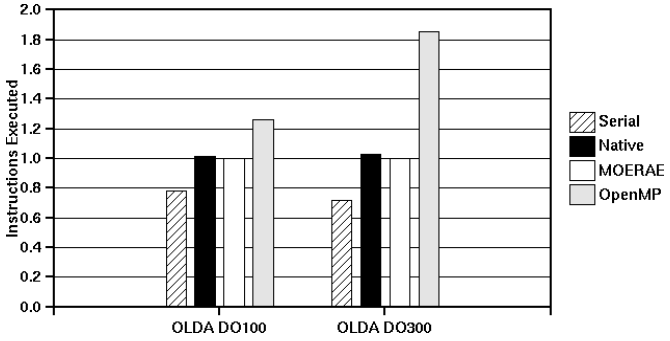

Figure 5: The Number of Instructions Executed in the Major Loops of `TRFD` on the Sun Enterprise 4000. The values are normalized to instructions executed in the MOERAE code. We refer to a code with low relative number of instructions as having high *instruction efficiency*.

The instruction efficiency also reflects the ability of the backend compiler to apply advanced optimizations. In our analysis we have found that the parallel program representation in the interface between preprocessor and backend compiler can have a substantial impact on these optimizations. The semantics of shared-memory parallel program execution requires that data written by one processor can be read immediately by another processor. A conservative compilation of such a program would disable all register allocation of shared data (because the shared variable could be overwritten by a different processor at any time, hence the register value would become stale). Simple analysis methods could improve these conservative assumptions. For example, one could easily determine that the variable `factor` in Figure 6 is read-only inside the parallel loop. However, we have found that both the native and the OpenMP compiler translate this case most conservatively. In contrast, MOERAE knows when Polaris generates loops that are dependence-free, hence

unrestricted register allocation is possible. The sequential backend compiler of MOERAE performs this optimization in the usual way. One reason that this optimization does not have an even bigger performance impact, is that after the first access, the variable `factor` is in the cache, hence the second access is a cache hit. On the other hand, if a compiler tends to pass program constants as subroutine parameters, the called subroutine will see them as shared variables, amplifying the described effect. This is the case in our OpenMP compiler, and it accounts for the inferior performance of the shown loops. A simple solution to this problem would be for the preprocessor to add directives indicating that the loop or the accessed variables are dependence-free. We have experimented with such improved interfaces between preprocessor and backend compiler and have found significant improvements of up to 53% [12].

```
C$OMP PARALLEL DO
      DO i=1,n
         a(i) = b(i) * factor
         c(i) = d(i) * factor
         e(i) = f(i) * factor
      ENDDO
```

Figure 6: Conservative Assumption by the Backend Compiler. The shared variable `factor` is read-only. However, the backend compiler does not detect this fact and disables the allocation of `factor` in a register.

In summary, in `TRFD` the thread-based MOERAE representation leads to better performance than the loop-based OpenMP and the native directive forms. The primary reason is in the more highly optimized code generated by the backend compiler, which is enabled by the MOERAE representation. Functional portability between the two machines is provided and the obtained speedups on the two machines are comparable.

B. MDG

In `MDG`, two parallel loops `INTERF DO1000` and `POTENG DO2000` take more than 95% of the total execution time in the serial code. Figure 7 shows the speedups of these two loops. OpenMP yields the best performance on the Sun Enterprise, and MOERAE on the SGI Origin. However the differences are not as pronounced as in `TRFD`.

Similar to `TRFD`, the cache hit ratio is close to 97% and the loops are large and regular. The difference in performance is again due to the better instruction efficiency as indicated by Figure 8. However, in contrast to `TRFD`, the instruction efficiencies are generally better and can even be higher than in the original code. We attribute this to optimizations performed in the parallelizing preprocessor (a combination of partial inline expansion and constant propagation). The number of executed instructions in `INTERF DO1000` using the native Sun directive version is larger than the others because of an instruction scheduling inefficiency. The Sun directive version uses an indirect addressing mode rather than direct addressing. In `POTENG DO2000` the native Sun directive version could not be parallelized due to an error

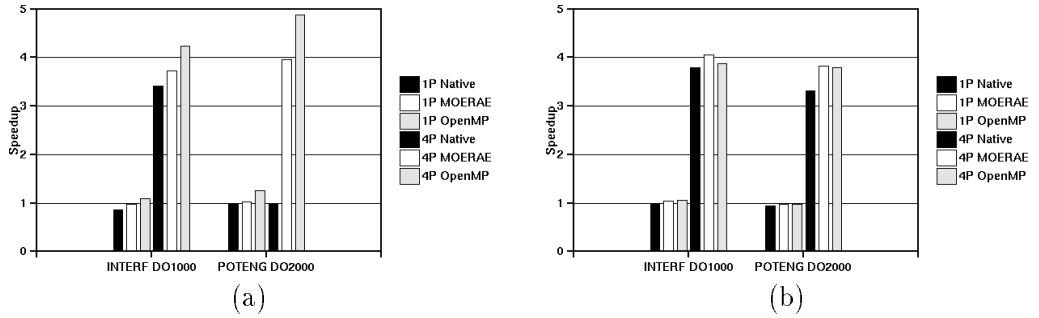

Figure 7: Speedup of the Major Loops in `MDG` as Executed on (a) the Sun Enterprise 4000 and (b) the SGI Origin 2000.

in the native compiler. However, the number of executed instructions in this loop still increases slightly by 3% in the resulting code.

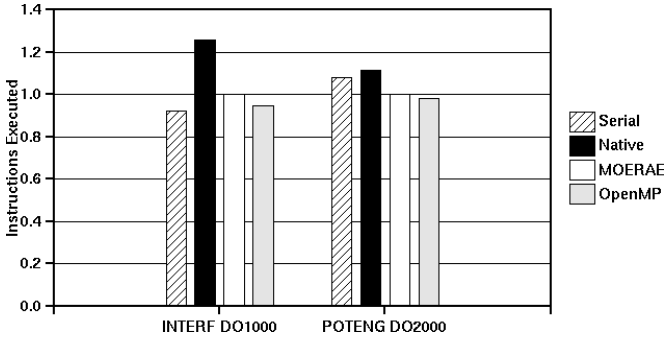

Figure 8: The Number of Instructions Executed in the Major Loops of `MDG` on the Sun Enterprise 4000. The values are normalized to instructions executed in the MOERAE code.

On the SGI system, the overall performance of MOERAE is better than OpenMP and similar to the native directive form, even though the performance of the major loops is similar. One reason for this is the many small parallel loops in `MDG`, which incur a substantial fork/join overhead on the SGI system. We have developed techniques that address this problem in related work [13]. However, the present results do not include this optimization.

In summary, `MDG` does not favor any particular parallel program representation. Instruction efficiency and small-loop overheads account for the observed performance differences. Functional portability is provided, but the performance on the two machines differs noticeably.

C. ARC2D

`ARC2D` consists of many small loops, each of which has a few micro-seconds average execution time. Figure 9 shows that the three code variants yield comparable performance. The loops in Figure 9 consume about one third of the total execution time. Two loops show a highly superlinear speedup. This is due to the fact that Polaris applies loop interchanging

to achieve stride-1 references in these loops. This optimization is not applied in the serial version. On the Sun Enterprise, the overall performance of MOERAE is best, but on the SGI Origin OpenMP is best. `ARC2D` differs from `TRFD` and `MDG` in that the cache hit ratio is 91.6%. A miss rate of 8.4% is significant and negatively impacts the program speedup. In fact, stalls due to cache misses on store instructions consume over half of the total execution time.

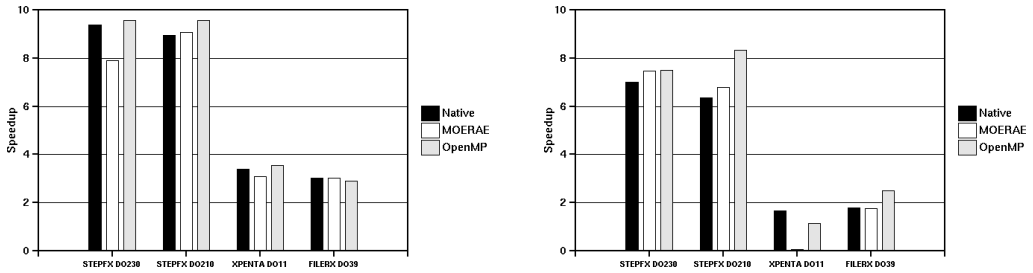

Figure 9: The Speedups of the Major Loops in `ARC2D` executed on (a) the Sun Enterprise 4000 and (b) the SGI Origin 2000.

In addition, the loops have short execution times and are executed many times. This repeated incurring of fork/join overheads likewise will degrade speedup. The fork/join overhead is larger on the SGI system, which is consistent with our measurements. They show generally lower speedup levels on this machine. The same holds for the overall performance shown in Figure 2. Figure 10 shows the measured executed instructions, which does not match the measured execution time well. It is evident that there are other factors in `ARC2D` that must determine the performance differences in the versions, although the measured speedups do show that there are large gaps in the optimization done by the backend compilers.

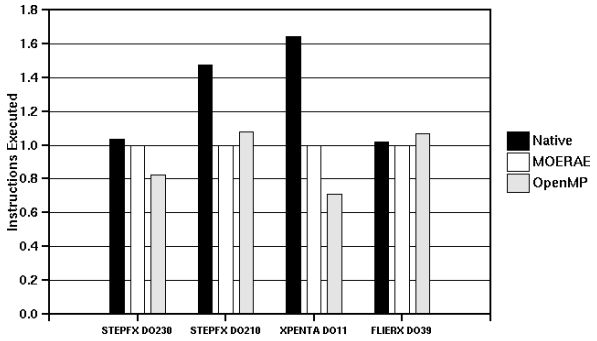

Figure 10: The Number of Instructions Executed in the Major Loops of `ARC2D` on the Sun Enterprise 4000. The values are normalized to instructions executed in the MOERAE code.

In summary, as for the other codes, the parallel-loop and thread-based representations show comparable performance, although substantial differences exist for individual loops. Cache misses, fork/join, and instruction efficiency impact the performance of the parallel

code. As for MDG, functional portability is provided, but the speedups obtained on the two measured machines differ.

D. FLO52

As with ARC2D, the major loops in FLO52 have small execution times. Figure 11 shows the performance of the various parallel forms. On the Sun system, the native directive form performs better than the others except for one loop, with MOERAE's performance falling between that of the native directive code and OpenMP. On the Origin, the OpenMP version is best, with the native directive form again falling between the other two. But in the overall performance on both machines, the OpenMP versions are best.

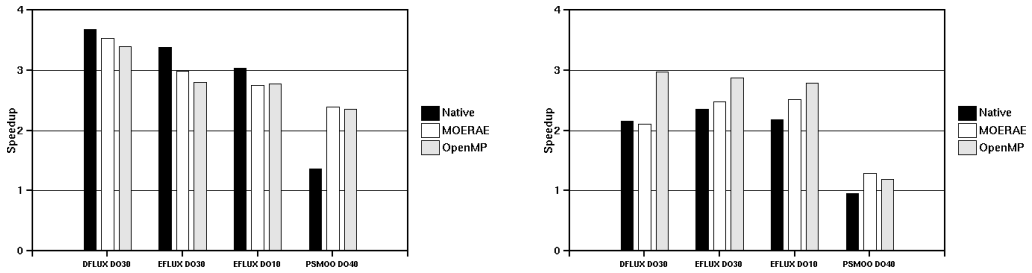

Figure 11: The Speedups of the Major Loops in FLO52 executed on (a) the Sun Enterprise 4000 and (b) the SGI Origin 2000.

Again, due to the short loop execution times, the fork/join overheads can have significant impact on loop performance. The Origin performance is lower in all measurements, which we attribute to this fact. The fork/join overhead also has a significant effect on the overall execution time of this program, with the overall program speedup on the Origin degrading to a 1.5 at best.

The cache hit ratio in this program is close to 94%, hence the miss rate may negatively impact the program speedup. The number of instructions executed in the major loops of FLO52 is shown in Figure 12. Although most code variants agree in the number of executed instructions, there are two outliers. The low instruction efficiency in the native directive version of loop PSMOO DO40 is due to the conservative code generation, explained in Figure 6. It accounts for the low speedup of this loop on the Sun Enterprise (Figure 11 (a)). However, the loop DFLUX DO30 shows a good speedup despite the apparent low instruction efficiency. The reason for the low number of instructions in the serial version is the code generated for an expression of the form $x - y$. It is generated as only one subtraction instruction in the serial code, whereas in all other versions the expression is compiled as two instructions, a negation and an addition. Despite the code differences the instruction cycles for the two variants are the same [14].

In summary, the loop-parallel and thread-based program forms agree in their performance. Fork/join overhead and instruction efficiency are the primary overhead factors of the parallel performance. Porting the code from the Sun Enterprise to the SGI Origin incurs

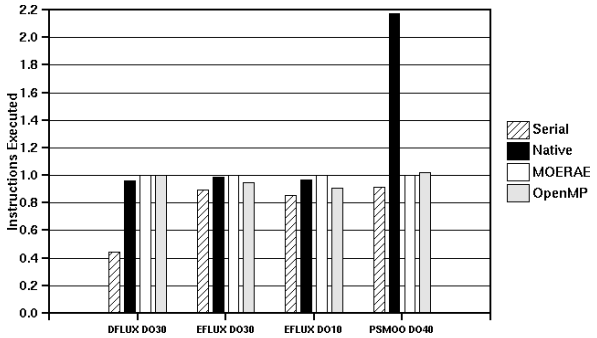

Figure 12: The Number of Instructions Executed in the Major Loops of `FLO52` on the Sun Enterprise 4000. The values are normalized to instructions executed in the MOERAE code.

a substantial performance decrease, which we attribute to the higher fork/join overhead.

E. BDNA

The `BDNA` benchmark is different from the other programs in that it contains a substantial number of reduction operations. Both scalar reductions and array reductions (in which the reduction variable is an array) are used. The Polaris parallelizer, together with the MOERAE output pass, can translate reductions into several forms, potentially resulting in substantial performance differences. Because of this we will describe this aspect of `BDNA` in particular and compare the performance of these different schemes.

Polaris generates three forms for expressing reduction operations, called *blocked reduction*, *privatized reduction*, and *expanded reduction* [15, 16]. Briefly, in blocked reductions all reduction statements are enclosed within critical sections such that they are performed atomically within a parallel loop. In contrast, in both privatized and expanded reductions, the reduction variables are replicated versions of the original variable. The replication is done either by array privatization or by array expansion. Each processor performs the reduction into its local version of the replicated variable. At the end of the loop, a global reduction is performed from the replicated variable into the original reduction variable. For privatized reductions this is done within a critical section at the end of the parallel loop in a *postamble*. For expanded reductions the global reduction can be done after the parallel loop. Before the parallel loop body, a *preamble* sets the replicated reduction variable to zero. Most compilers for parallel languages can transform scalar reductions. Polaris is able to also recognize array reductions. MOERAE's task includes the transformation of such operations into the proper parallel form.

`BDNA` has two major parallel loops, `ACTFOR DO500` and `ACTFOR DO240`, which take more than 80% of the total serial execution time. The loop `ACTFOR DO500` performs reductions on three arrays and one scalar variable, and the loop `ACTFOR DO240` has 12 array reductions and one scalar reduction. The performance of these loops is shown in Figure 13.

Table 1 compares the performance of the privatized and the expanded reduction schemes in `ACTFOR DO500` in the MOERAE versions. Expanded reductions are about 50% better than the privatized reductions in execution time. Even if the array size of the reduction

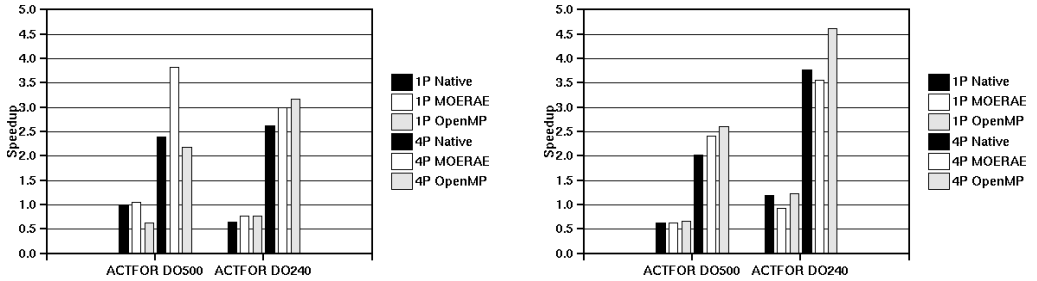

Figure 13: Speedup of the Major Loops in `BDNA` as Executed on (a) the Sun Enterprise 4000 and (b) the SGI Origin 2000.

Table 1: Composition of Execution Time (Seconds) in Loop `ACTFOR DO500` when Using Privatized Reductions and Expanded Reductions in MOERAE on the Sun Enterprise 4000.

Reduction	Statements	Number of Processors		
		1	2	4
Privatized	Memory Allocation	0.000186	0.000358	0.000605
	Initialization	0.000109	0.000108	0.000098
	Computation	2.479811	1.189530	0.600789
	Summation	0.000192	0.000290	0.000345
	Free	0.000020	0.000011	0.000011
Expanded	Memory Allocation	0.000293	0.000255	0.000284
	Initialization	0.000100	0.000223	0.000345
	Computation	1.486813	0.760792	0.396877
	Summation	0.000163	0.000205	0.000261
	Free	0.000019	0.000012	0.000015

variables is large, the overhead of the preamble (which allocates memory and initializes it to zero) and the postamble (which does the final summation and frees memory) are negligible compared to the loop computation time.

The initialization and summation in an expanded reduction are parallelized. The actual computation dominates the overall execution time. The overhead of memory allocation in the privatized reduction is larger than that in an expanded reduction, because the privatized reduction allocates separately for each thread from a common pool, guarded by a lock/unlock pair. The overhead of initialization in the expanded reduction is larger than for privatized reductions. In expanded reductions, during initialization the shared array is being accessed, requiring interprocessor communication, while in privatized reductions a local copy is accessed. The overhead of summation in the privatized reduction is larger because of the overhead of lock and unlock operations. In the overall program performance, the choice of reduction operation makes a substantial 10% difference. In Figure 13 all programs use the expanded reduction scheme.

Similar to other benchmarks, `BDNA` has a high cache hit ratio and the execution time

is primarily influenced by the number of executed instructions. There is a substantial difference in speedup for the `ACTFOR DO500` loop on the Sun system between the MOERAE and the other program forms. However, Figure 14 indicates better instruction efficiency for the serial and the native versions than the MOERAE version. A detailed analysis of the assembly code has revealed that the generated code in the native and OpenMP directive forms lead to significant pipeline stalls, which is not the case in the MOERAE variant. The execution time does not match the number of executed instructions. We have found that in neither case the base address of the reduction array is placed in a register. In the privatized reduction code the memory load and use of this address is done in two consecutive instructions, while in the expanded reduction case, the load is moved upward in the instruction stream. The latter allows the pipelined SPARC architecture [17] to overlap this memory load with the subsequent instructions. For the code generator, the difference between the two reduction schemes is merely that of a subroutine parameter versus an address on the local stack. This raises code generation and register allocation issues beyond the scope of this paper.

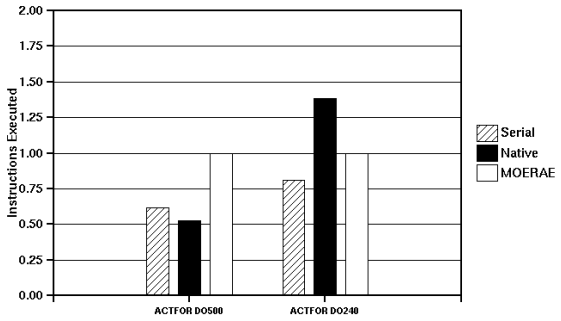

Figure 14: The Number of Instructions Executed in the Major Loops of `BDNA` on the Sun Enterprise 4000. The values are normalized to instructions executed in the MOERAE code.

In summary, all program variants perform comparably. Instruction efficiency varies substantially in this program. Among the measured schemes for parallel reductions, the expanded reduction form performed the best. As for most programs, portability is provided but the performance differs substantially between the Sun Enterprise and the SGI Origin systems.

V. Conclusion

In this paper we have investigated three questions that are related to the performance of compiler-parallelized programs. The questions are (1) whether or not the common use of parallel loop representations between the preprocessor and backend compiler has a negative performance effect, (2) why fully parallel programs that exhibit good cache locality perform less than ideal on current SMP machines, and (3) to what extent the output of parallelizing compilers can be ported across SMP platforms.

To answer these questions we have implemented two output passes to the Polaris parallelizing preprocessor, which generate parallel loop directives and a thread-based parallel form, respectively. We have described the latter pass, the MOERAE translator and runtime library, in this paper. We have analyzed the performance of five Perfect benchmarks, which can be parallelized to a high degree by Polaris. Our analysis included detailed measurements on a loop-by loop basis on both a Sun Enterprise and an SGI Origin machine.

We have found that, overall, there is no significant performance degradation attributable to the loop-parallel representation. Although we have found substantial performance differences at individual loops as well as at the overall program level, no scheme outperformed the other consistently. One important reason for the performance difference is the instruction efficiency. That is, the code generated by the different compilers results in substantially different numbers of executed instructions. We have found that an important cause is the lack of data dependence information in the parallel loop representation between the preprocessor and the backend compiler, which leads to conservative optimizations in the backend compiler. Another cause for performance differences was the fork/join overhead in small parallel loops. Of minor importance was the cache behavior. Three of the five benchmarks had negligible cache misses. We have also found that different schemes for implementing reduction operations can make a significant difference for individual loops. In one of the five applications, this made a 10% overall performance difference.

We have also found that both the OpenMP and the MOERAE thread-parallel program versions are functionally portable between the two SMP platforms. However, there are substantial performance differences. One reason for this is the different fork/join overhead of the two machines, which results in a different threshold for profitable parallel loops.

In addition to our performance results, a contribution of this work is the MOERAE translator and runtime library. The system is available for distribution together with the Polaris infrastructure. It provides a portable environment, comparable to the OpenMP language. We have shown in this paper that analyzing performance effects of the translation system is often complex. MOERAE simplifies this analysis since it uses only standard, sequential compilers as a backend. Compared to a compiler for parallel directive languages, this gives its users more direct insight into the performance behavior of parallel programs.

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.

[2] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, pages 84–89, December 1996.

[3] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. *Proceedings of the 7th ACM International Conference on Supercomputing (ICS'93)*, pages 416–424, 1993.

[4] Xingbin Zhang and Andrew A. Chien. Dynamic pointer alignment: tiling and communication optimizations for parallel pointer-based computations. *Proceedings of the Sixth ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming (PPoPP'97), pages 37–47, 1997.

[5] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI'90)*, pages 296–310, 1990.

[6] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 106–117, 1998.

[7] Carrie Brownhill, Alex Nicolau, Steve Novack, and Constantine Polychronopoulos. The PROMIS compiler prototype. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, 1997.

[8] Sangyeun Cho, Jenn-Yuan Tsai, Yonghong Song, Bixia Zheng, Stephen J. Schwinn, Xin Wang, Qing Zhao, Zhiyuan Li, David J. Lilja, and Pen-Chung Yew. High-level information - an approach for integrating front-end and back-end compilers. *Proceedings of the 1998 International Conference on Parallel Processing (ICPP'98)*, August 1998.

[9] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[10] William Blume and Rudolf Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[11] Seon Wook Kim, Michael Voss, and Rudolf Eigenmann. A methodology and a tool for cache characterization of loop-parallel programs. Technical Report ECE-HPCLab-99201, HPCLAB, Purdue University, School of Electrical and Computer Engineering, 1999.

[12] Seon Wook Kim and Rudolf Eigenmann. Interface issues between parallelizing preprocessors and code generators. Technical Report ECE-HPCLab-99208, HPCLAB, Purdue University, School of Electrical and Computer Eng ineering, 1999.

[13] Michael J. Voss and Rudolf Eigenmann. Generating portable shared-memory applications using OpenMP. Technical Report ECE-HPCLab-98207, HPCLAB, Purdue University, School of Electrical and Computer Engineering, 1998.

[14] David L. Weaver and Tom Germond. *The SPARC Architecture Manual, Version 9.* SPARC International, Inc., PTR Prentice Hall, Englewood Cliffs, NJ 07632, 1994.

[15] Jee Ku. The design of an efficient and portable interface between a parallelizing compiler and its target machine. Master's thesis, University of Illinois at Urbana-Champaign, Department of Electrical Engineering, December 1995.

[16] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing (ICS'95)*, pages 444–448, 1995.

[17] Richard P. Paul. *SPARC Architecture, Assembly Language Programming, & C.* Prentice Hall, Englewood Cliff, NJ, 1994.