Parallel Processing Letters,

Where Does the Speedup Go: Quantitative Modeling of Performance Losses in Shared-Memory Programs*

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

ABSTRACT

Even fully parallel shared-memory program sections may perform significantly below the ideal speedup of P on P processors. Relatively little quantitative information is available about the sources of such inefficiencies. In this paper we present a *speedup component model* that is able to fully account for sources of performance loss in parallel program sections. The model categorizes the gap between measured and ideal speedup into the four components *memory stalls, processor stalls, code overhead*, and *thread management overhead*. These model components are measured based on hardware counters and timers, with which programs are instrumented automatically by our compiler. The speedup component model allows us, for the first time, to quantitatively state the reasons for less-than-optimal program performance, on a program section basis. The overhead components are chosen such that they can be associated directly with software and hardware techniques that may improve performance. Although general, our model is especially suited for the analysis of loop-oriented programs, such as those written in the OpenMP API. We have applied this model to compare three parallel code generation schemes for the Polaris parallelizing compiler. It helps us answer questions such as, what sources of inefficiencies are present in compiler-parallelized programs. To discuss the question we have also implemented an alternative, thread-based code generation method.

1. Introduction

With the increasingly wide availability of shared-memory multiprocessor servers and workstations, the importance of compilers that can generate code for parallel programs and that can parallelize standard, sequential programs is growing steadily. There exists a large body of research in parallel compiler techniques, ranging from detecting parallelism to efficiently generating 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 still exist when detecting parallelism in science and engineering applications that contain sparse data structures and in C, C++, and Java programs, which contain pointers and irregular control flow.

In this paper we consider a problem that has been given relatively little attention in the past. We have observed that even highly parallel programs that exhibit good

*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.

cache locality can perform poorly on today's shared memory machines. For example, in the `TRFD` Perfect benchmark, two major loops consume 93% of the overall time in the serial code. The loops are fully parallel 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 only a speedup of 2.3.

Several reasons can be responsible for such behavior. Highly-parallel programs that do not perform well on SMP systems are often thought of as "not having enough locality," leading to many cache misses and thus stalls in the memory system. In [3], the cache behavior of serial programs was analyzed in detail. In [4], the cache behavior of parallel programs was quantified. However the impact of the cache behavior on program speedups has not been quantified. Another source of performance loss can be that parallel code is less-highly optimized by the compiler than the equivalent serial program. This can be due to conservative assumptions made by the code generating compiler, which can be caused by a lack of integration of the parallelizing compiler and the backend. To address this problem, [5] uses a common representation for both compilation steps. In [6], information is passed from the frontend to the backend compiler using a universal format. Again, the impact of this factor has not been quantified. A third reason for limited performance is synchronization overhead, or more general, thread management overhead. A large number of performance analysis contributions have measured this overhead in various applications. Loop-parallel applications typically have communication and barrier overheads (a.k.a. fork&join overhead) at the beginning and end of each parallel section. Again, no quantitative information is available as to how this impacts program efficiencies.

The present paper complements and continues work presented in [4], where we have analyzed the performance of several programs using a cache simulator and the measured number of executed instructions. We have found parallel programs that exhibited high cache locality but nevertheless exhibited low speedups. One of the symptoms was that these programs executed substantially higher numbers of instructions than in their original serial version. Several other reasons were identified and discussed qualitatively. In the present paper we have quantified these effects such that we can attribute a percentage of "loss of speedup" to them. To do so, we have developed modified metrics, which we have measured using hardware counters available on SPARC V9 processors. The hardware counters monitor 25 performance aspects such as cycles, the number of dynamic instructions, cache access statistics, bus transactions, and pipeline stalls [7].

The remainder of the paper is organized as follows. Section 2 describes the experimental setup and overall program execution results that we have obtained on the Sun Enterprise system. Section 3 presents the speedup component model and investigates the performance of the most time-consuming loops in each program in detail. Section 4 presents conclusions.

2. Experiment Setup and Overall Program Performance

For our experiments we have implemented three *output passes* for the Polaris parallelizing compiler. They generate parallel programs in three forms, using Sun directives, OpenMP, and thread-based programs, respectively. Our implementation of the MOERAE translator and microtasking library to generate thread-based code was described in [4]. The test suite consists of four 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 system. All these translations were performed by Polaris. Figure 1 gives an overview of our software system.

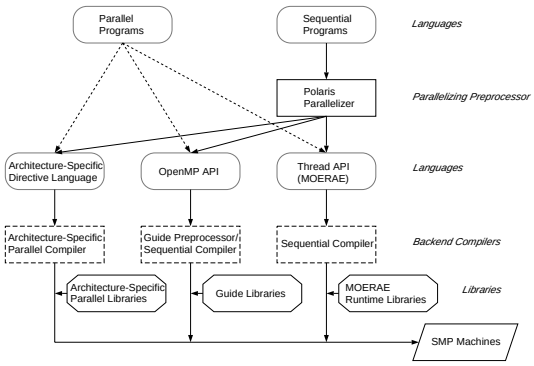
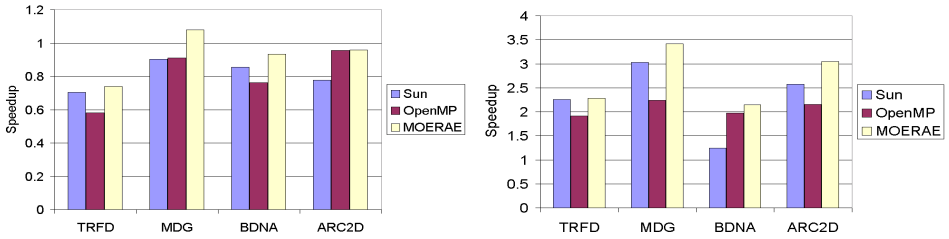


Fig. 1. Language and Translator System Used in Our Machine Environment.

Figure 2 shows the overall performance of our benchmarks on a Sun Enterprise 4000 system. The serial (non-parallelized) program defines *speedup=1* in our performance results. We used the programs `TRFD`, `MDG`, `BDNA`, and `ARC2D` from the Perfect Benchmarks suite [8]. The Sun Enterprise 4000 runs Solaris 2.6, has six 248 MHz UltraSPARC V9 processors, each with a 16KB L1 data cache and 1 MB unified L2 cache. Each code was compiled by the Sun 5.0 `C` and `Fortran` compilers with the flags `-xtarget=ultra2 -xcache=16/32/1:1024/64/1` and `-O5` for `Fortran`, and `-xO3` for `C` codes. Each code was parallelized by Polaris and transformed into the three parallel forms shown in Figure 1. For quantitative performance measurements, we used the available hardware counter (`TICK` register).

The overall results show that our thread-based MOERAE scheme outperforms the loop-based OpenMP and architecture-specific directive schemes in all measured benchmarks. Figure 2 (a) shows the parallel code executed on one processor relative to the serial execution time. This difference, which we term *parallelization overhead*, is substantial in most programs. Figure 2 (b) shows the speedup of compiler-parallelized codes using 4 processors with respect to the original sequential time. Although all three code variants exhibit the exact same parallelism, the individual



(a) P = 1. (b) P = 4.

Fig. 2. Speedup of Benchmarks on Sun Enterprise 4000. All speedups are calculated with respect to the original sequential time. P is the number of processors.

numbers differ substantially. The measured applications exploit a high degree of parallelism, but the achieved performance is far below the ideal speedup of four. Compared to the results previously reported in [4], there are small performance differences due to a new release of Sun's code generating compiler. In `MDG`, the improvement is significant and is due to a compiler bug fix, which prevented higher optimizations in the previous release. In the next section, we discuss the performance gap between measured and ideal speedups using our proposed performance model.

3. Detailed Performance Analysis and Modeling

The basis for our analysis is the loop-by-loop performance of the benchmarks on the Sun Enterprise 4000 machine, which is monitored by timers and hardware counters. As in Figure 2, all programs are measured in three different forms. They exploit the exact same parallelism (detected by the Polaris preprocessor), but differ in their representation and backend compilers as shown in Figure 1.

3.1. Overhead Categories

The objective of our quantitative model is to be able to fully account for the performance losses incurred by each parallel program section on a shared-memory multiprocessor system. We categorize overhead factors into four main components. Table 1 shows the categories and their contributing factors. In this paper we present only the major components. Where necessary, we will discuss additional factors when analyzing the performance of individual loops. The full details of this analysis is documented in [9]. We will use the notation $T_i(S)$ for overhead incurred in the sequential execution and $T_i(P)$ for overheads incurred in the parallel execution. The notion i stands for an overhead category.

Memory stalls reflect latencies incurred due to cache misses, memory access times and network congestion. *Processor stalls* account for delays incurred processor-internally. We collect both categories through hardware counters.

Code overhead means that executing a section of parallel code, excluding stalls,

Table 1. Overhead Categories of the Speedup Component Model.

Overhead Category	Contributing Factors	Description	Measured with
Memory stalls	IC miss	Stall due to I-Cache miss.	HW Cntr
	Write stall	The store buffer cannot hold additional stores.	HW Cntr
	Read stall	An instruction in the execute stage depends on an earlier load that is not yet available.	HW Cntr
	RAW load stall	A read needs to wait for a previously issued write to the same address.	HW Cntr
Processor stalls	Mispred. Stall	Stall caused by branch misprediction and recovery.	HW Cntr
	Float Dep. stall	An instruction needs to wait for the result of a floating point operation.	HW Cntr
Code overhead	Parallelization	Added code necessary for generating parallel code.	computed
	Code generation	More conservative compiler optimizations for parallel code.	computed
Thread management	Fork&join	Latencies due to creating and terminating parallel sections.	timers
	Load imbalance	Wait time at join points due to uneven workload distribution.	

takes longer than the equivalent serial section† It may have been introduced when parallelizing the program (e.g., by substituting an induction variable) or through a more conservative parallel code generating compiler. In our performance analysis we compute the time spent due to code overhead by subtracting all stall cycles reported by the hardware counters from the overall number of executed cycles.

$$T_{code\_overhead}(P) = T_{total\_measured\_cycle}(P) - T_{stalls}(P) - T_{ideal}(P) \quad (1)$$

where $T_{stalls}(P) = T_{memory\_stall}(P) + T_{processor\_stall}(P) + T_{thread\_management}(P)$ and $T_{ideal}(P) = \frac{T_{ideal}(S)}{P}$.

Thread management accounts for latencies incurred at the fork and join points of each parallel section. It includes the times for creating or notifying waiting threads, for passing parameters to them, and for executing barrier operations. It also includes the idle times spent waiting at barriers, which are due to unbalanced thread workloads. We measure these latencies directly through timers before and after each fork and each join point. Timers and hardware counters introduce overheads in the order of 10μs, which we subtract from our measurements for accuracy. However, the loops discussed here have long execution times, so that the instrumentation impact on performance is minimal.

Our overhead categories correspond to software and hardware issues, such that the presence of a certain category indicates an opportunity for improved techniques in the respective area. Memory stalls are best addressed through locality-enhancing software techniques. Processor stalls primarily point out opportunities for processor

†In [4] we have used the term *instruction efficiency*, which counts the difference in number of executed instructions between serial and parallel code. Instead, the term *code overhead* also includes the time taken by these instructions.

architecture improvements. Code overhead raises questions of compiler optimizations in both parallelizers and code generators. Thread management latencies can be reduced through highly-optimized runtime libraries and through improved balancing schemes of threads with uneven workloads.

3.2. The Speedup Component Model

Our goal is to display the above overhead categories in a form that allows us to easily see why a parallel region does not exhibit an ideal speedup of P on P processors. The basic idea is to identify overheads that increase between the serial and the parallel program execution. There is no code overhead and thread management in the serial program. However there are memory and processor stalls, and it is the increase in these categories that affects the speedup. We will then fill in the gap between the measured speedup and the ideal speedup with the overhead categories according to their magnitude. We call these the *speedup components*, denoted as $S_{loss}(i,P)$ for overhead i, and the overall display the *speedup component model*. Because of the presence of overheads in the serial program, it is possible that they *decrease* in the parallel program (e.g., this can occur if the parallelizer applies locality enhancement techniques not applied in the serial compilation). This leads to apparent superlinear speedups and will show up in our model as negative overhead components.

We compute the speedup components as follows. The ideal speedup on P processors is

$$S_{ideal\_speedup} = P = \frac{T_{ideal}(S)}{T_{ideal}(P)} = S_{measured\_speedup} + \sum_i S_{loss}(i,P) \qquad (2)$$

where i is an overhead component to be considered in the model. Then, with $T_{total\_measured\_cycle}(S) = T_{ideal}(S) + \sum_i T_i(S)$ and $T_{total\_measured\_cycle}(P) = T_{ideal}(P) + \sum_i T_i(P)$,

$$\begin{aligned} \sum_i S_{loss}(i,P) &= S_{ideal\_speedup} - S_{measured\_speedup} \\ &= \frac{T_{ideal}(S)}{T_{ideal}(P)} - \frac{T_{total\_measured\_cycle}(S)}{T_{total\_measured\_cycle}(P)} \\ &= \sum_i \frac{P \times T_i(P) - T_i(S)}{T_{total\_measured\_cycle}(P)} \end{aligned} \qquad (3)$$

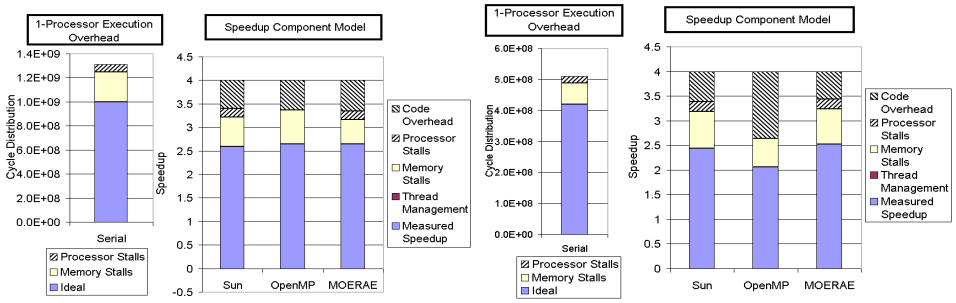
Hence, for each overhead i we have a speedup component $\frac{P \times T_i(P) - T_i(S)}{T_{total\_measured\_cycle}(P)}$ and the sum of these components make the difference between measured and ideal speedup.

We will use this speedup component model to discuss the performance of four benchmarks. For each benchmark we will display (1) the measured performance of the parallel code relative to the serial version, (2) the execution overheads of the serial code in terms of stall cycles reported by the hardware monitor, and (3) the

speedup component model for each of the three parallel variants. We will discuss details of the analysis where necessary to explain effects. However, for the full analysis with detailed overhead factors and a larger set of programs we refer the reader to [9].

3.3. TRFD

In `TRFD`, two loops `OLDA DO100`[‡] and `OLDA DO300` take more than 93% of the total execution time in the serial code. As shown in Figure 3, the MOERAE thread-based program form outperforms the OpenMP variant in `OLDA DO300`.



(a) `OLDA DO100`. (b) `OLDA DO300`.

Fig. 3. Performance of the Major Loops in `TRFD`. `P` is the number of processors. The speedup component model quantifies overheads responsible for the difference between measured and ideal speedups.

`TRFD` contains large, regular loops and the data cache hit ratio of the parallel program is about 96.2% for read and 98.4% for write in both serial and parallel executions. Figure 3 shows that the code overheads and memory system stalls degrade the speedup by about 1.5. The Figure says that both memory stalls and code overhead contribute substantially to the low speedup of these loops. It is important to note that we cannot conclude that removing all memory stalls would directly lead to a speedup of 3.1. While we can expect significant gains from locality-enhancing techniques, memory stalls may in part hide other overheads (e.g., processor stalls) that would become visible when the memory stalls are removed.

The OpenMP variants have no noticeable processor stalls, however they have higher memory stalls than the other variants. So, memory stalls may hide processor stalls in this case. In the serial version, memory stalls are the most important overhead component.

The code overhead of the parallel code is consistent with the fact that the benchmark has complex induction variables, whose substitution in the parallel code leads to significantly more complex expressions. The code overhead also reflects the ability of the backend compiler to apply advanced optimizations. In our analysis we

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

have found that the parallel program representation in the interface between preprocessor and backend compiler may have a substantial impact on these optimizations. For example, the semantics of shared-memory parallel program execution in general requires sequential consistency. That is, 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 4 is read-only inside the parallel loop. However, we have found that the OpenMP compiler translates this case most conservatively. By 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 at runtime 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 contributes to the inferior performance of `OLDA DO300` loop.

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% [9]. The OpenMP API offers another solution. The programmer can add `flush` directives at points in parallel code sections where values of shared variables need to be communicated to other processors. Effectively this means that compilers can assume a relaxed memory consistency model[§]

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

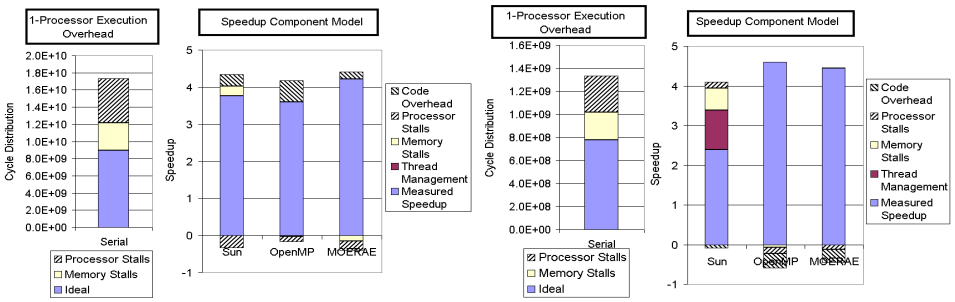
Fig. 4. Conservative Assumption by the Backend Compiler. The shared variable `factor` is read-only. However, the backend compiler does not detect this fact and disables the allocation of `factor` in a register.

3.4. MDG

In `MDG`, two parallel loops `INTERF DO1000` and `POTENG DO2000` take more than 98% of the total serial execution time. Figure 5 shows the speedups of these two

[§] The OpenMP definition [10] does not state the assumed memory consistency model explicitly.

loops. MOERAE and OpenMP yield the best performance in each loop. `MDG` has good locality of about 97% for read and 94% for write in both serial and parallel executions. The speedup component model reflects this situation.



(a) `INTERF DO1000`. (b) `POTENG DO2000`.

Fig. 5. Performance of the Major Loops in `MDG`. `P` is the number of processors. Negative components of the speedup component model (shown below to 0-line) explain superlinear behavior. The sum of all components always equals the ideal speedup (=4).

In `INTERF DO1000`, MOERAE has little code overhead and even a negative processor stall component. The serial version suffers from substantial processor stalls, which is mostly due to floating-point dependences. This overhead is less in the parallel code, leading to the negative overhead components. The Sun and the OpenMP variants have large parallel code overhead, and the Sun variant also incurs additional memory stalls.

In `POTENG 2000`, all three variants have negative code overhead, which is due to functions inlining inside the loop. The Sun variant suffers from load imbalance, which is part of the thread management overhead component. Although the loop scheduling scheme in the Sun variant assigns iterations equally to processors, these iterations have uneven execution times. Furthermore, the Sun variant shows a significant memory stall component, which is mainly caused by load stalls.

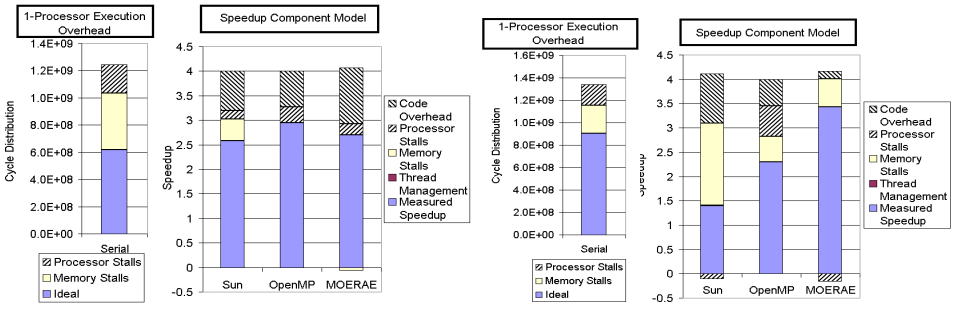
3.5. BDNA

`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 reduction operations 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 6.

In `ACTFOR DO240`, the code overhead is similar in all variants. Up to 1.0 in speedup loss can be attributed to this component, which is in part due to array reduction transformations. The Sun directive code has a large amount of memory stalls, mainly load stalls. All three code variants have similar cache accesses, hence we attribute this overhead to differences in the instruction scheduling scheme.

In `ACTFOR DO500`, the Sun variant has significant memory overheads, due to

both load and write stalls. The OpenMP variant incurs floating-point pipeline stalls and a large code overhead. This results from the conservative compiler code generation described in the Section . In both loops, the detailed comparison of the assembly code revealed that the instruction schedule of the Sun directive variant is inferior to the others. This may be caused by the fact that both the OpenMP and the MOERAE variant use the sequential compiler as a code generator, whereas the native directive codes appear to be generated with a parallel code generator. In `ACTFOR DO500`, the OpenMP variants incur significant floating-point pipeline stalls, which is part of the processor stall component.



(a) `ACTFOR DO240`. (b) `ACTFOR DO500`.

Fig. 6. Performance of the Major Loops in `BDNA`. `P` is the number of processors.

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. Depending on the reduction techniques used, the performance can vary significantly [4].

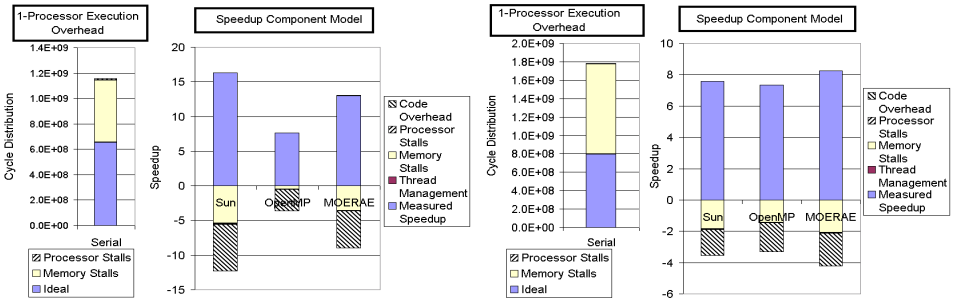
3.6. ARC2D

`ARC2D` consists of many small loops, each of which has a few milli-seconds average execution time. Figure 7 shows that the MOERAE and Sun code variants yield the best performance in most loops. The serial original code has significant memory stalls. Our detailed analysis revealed that these are mainly load stalls, there is a data cache read hit rate of 81.9% and a write hit rate of 55.5%.

In `FILERX DO19`, all parallel variants have a large negative code overhead and memory stall component. Inspecting the assembly code, we found that the only possible cause for this was loop unrolling applied in the parallel variants. The loop also has poor data locality. The Sun code has 77.0% data cache read hits and 2.41% store hit. The OpenMP version has 68.4% read hits and 4.12% store hits, while the MOERAE code has 75.6% read hits and 30.65% store hits.

The `STEPFX DO210` loop shows an even more highly superlinear speedup in all three code variants. This is due to the fact that Polaris applies loop interchanging to achieve stride-1 references. This optimization is not applied in the serial version.

Loop interchange further results in negative memory stall components.



(a) `FILERX DO19`. (b) `STEPFX DO210`.

Fig. 7. Performance of the Major Loops in `ARC2D`. `P` is the number of processors.

4. Conclusion

We have analyzed quantitatively the sources of performance loss in loop-parallel shared-memory programs. We have discussed four Perfect Benchmark programs in their automatically-parallelized form. We have compared parallel variants expressed in three different forms, in the OpenMP API, the native directive language from Sun Microsystems, and a new, thread-based variant. All programs are fully parallel, however their measured speedups on 4 processors range from an acceptable 3.5 to a poor 1.2.

For our analysis we have introduced a new *speedup component model*, which can fully account for the difference between measured and ideal speedup. It subdivides this performance gap into the 4 components *memory stalls, processor stalls, code overhead,* and *thread management*. These components account for the difference in overheads between the serial and the parallel program variants. Where these differences become negative, they represent sources of superlinear speedups.

We have found that, while memory stalls account for a significant speedup loss, a major secondary component is code overhead. It stems from the fact that (1) additional code is introduced for parallelizing a program (e.g., substituting induction variables) and (2) compilers tend to generate more conservative code for parallel program sections. The latter limitation was overcome by our thread-based API, although simple compiler optimizations and a relaxed memory consistency model could have the same effect in a loop-parallel API. A third cause for speedup losses are processor stalls. In our analysis we have identified several loops that exhibit significant stalls. They may serve as test cases for future, improved instruction scheduling schemes. However, we have also found situations where different compilation schemes lead to similar performance, but in one scheme memory stalls dominate while the other incurs mostly processor stalls. Hence, it would be incorrect to conclude that removing a speedup component directly results in the gain

represented by the component. Thread management is a fourth source of speedup loss. We have found situations where unbalanced workloads caused significant performance degradations. In the presented, most time-consuming loops, the fork/join overhead was negligible. However in many small loops this cost is higher and can add up to a significant overall program performance factor.

In addition to our performance results, a contribution of this work is the MOERAE thread-based translator and runtime library. The system is available for distribution together with the Polaris compiler infrastructure. It provides a portable environment, comparable to the OpenMP API. We have shown in this paper that analyzing performance effects of the translation system is often complex. MOERAE simplifies this analysis because 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] K. S. McKinley and O. Temam. A quantitative analysis of loop nested locality. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, October 1996.

[4] Seon Wook Kim, Michael Voss, and Rudolf Eigenmann. Performance analysis of parallel compiler backends on shared-memory multiprocessors. *Proceedings of Compilers for Parallel Computers (CPC2000)*, pages 305–320, January 2000.

[5] 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.

[6] 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.

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

[8] 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.

[9] Seon Wook Kim and Rudolf Eigenmann. Detailed, quantitative analysis of shared-memory parallel programs. Technical Report ECE-HPCLab-00201, HPCLAB, Purdue University, School of Electrical and Computer Engineering, 2000.

[10] OpenMP Forum, http://www.openmp.org/. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, October 1997.