A Performance Advisor Tool for Novice Programmers in Parallel Computing*

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

Abstract

Optimizing a parallel program is often difficult. For novice programmers, who lack the knowledge and intuition of advanced parallel programmers, it can be a very strenuous task. We have developed a framework that addresses this problem by automating the analysis of static program information and performance data, and offering active suggestions to novice programmers. Our tool enables experts to transfer programming experience to new users. It complement today's parallelizing compilers in that it helps to tune the performance of a compiler-optimized parallel program. To show its applicability, we present two case studies that utilize this system. By simply following the suggestions of our system, we were able to reduce the execution time of benchmark programs by as much as 39%.

1 Introduction

Parallelization, performance data analysis and program tuning are very difficult tasks for novice parallel programmers. Tools such as parallelizing compilers and visualization systems help facilitate this process. Today's state-of-the-art parallelization and visualization tools provide efficient automatic utilities and ample choices for viewing and monitoring the behavior of parallel applications.

Nevertheless, tasks such as identifying performance problems and finding the right solutions have remained cumbersome to many programmers. Meaningful interpretation of a large amount of performance data is challenging and takes significant time and effort. Once performance bottlenecks are found through analysis, programmers need to study code regions and devise remedies to address the problem. Programmers generally rely on their knowhow and intuition to accomplish these tasks. Experienced programmers have developed a sense of "what to look for" in the given data in the presence of performance problems. Tuning programs requires dealing with numerous individual instances of code segments. Categorizing these variants and finding the right remedies also demand sufficient experience from programmers. For novice programmers there are few choices other than empirically acquiring knowledge through trials and errors. Even learning parallel programming skills from experienced peers takes time and effort. No tools exists that help transfer knowledge from experienced to novice programmers.

*This work was supported in part by NSF grants #9703180-CCR, #9872516-EIA, and #9975275-EIA. This work is not necessarily representative of the positions or policies of the U. S. Government.

We believe that tools can be of considerable use in addressing these problems. We have developed a framework for an automatic performance advisor called MERLIN that allows performance evaluation experience to be shared with others. It analyzes program and performance characterization data and presents users with interpretations and suggestions for performance improvement. MERLIN is based on a database utility and an expression evaluator implemented previously [1]. With MERLIN, experienced programmers can guide novice programmers in handling individual analysis and tuning scenarios. The behavior of MERLIN is controlled by a knowledge-based database called a *performance map*. Using this framework, we have implemented several performance maps reflecting our experiences with parallel programs.

The contribution of this paper is to provide a mechanism and a tool that can assist novice programmers in parallel program performance tuning. Related work is presented in Section 2. Section 3 describes MERLIN in detail. In Section 4 and 5, we present two case studies that utilize the tool. First, we apply this system to the analysis of data gathered from hardware counters. Next, we present an application of MERLIN for the automatic analysis of timing and static program analysis data. Section 6 concludes the paper.

2 Related Work

Tools provide support for many steps in a parallelization and performance tuning scenario. Among the supporting tools are those that perform automatic parallelization, performance visualization, instrumentation, and debugging. Many of the current tools are summarized in [2, 3]. Performance visualization has been the subject of many previous efforts [4, 5, 6, 7, 8, 9], providing a wide variety of perspectives on many aspects of the program behavior. The natural next step in supporting the performance evaluation process is to automatically analyze the data and actively advise programmers. However, providing such support has been attempted by only few researchers.

The terms "performance guidance" or "performance advisor" are used in many different contexts. Here we use them to refer to taking a more active role in helping programmers overcome the obstacles in optimizing programs through an automated guidance system. In this section, we discuss several tools that support this functionality.

The SUIF Explorer's Parallelization Guru bases its analysis on two metrics: parallelism coverage and parallelism granularity [10]. These metrics are computed and updated when programmers make changes to a program and run it. It sorts profile data in a decreasing order to bring programmers' attention to most time-consuming sections of the program. It is also capable of analyzing data dependence information and highlighting the sections that need to be examined by the programmers.

The Paradyn Performance Consultant [6] discovers performance problems by searching through the space defined by its own search model (named W^3 space). The search process is fully automatic, but manual refinements to direct the search are possible as well. The result is presented to the users through a graphical display.

PPA [11] proposes a different approach in tuning message passing programs. Unlike the Parallelization Guru and the Performance Consultant, which base their analysis on runtime data and traces, PPA analyzes program source and uses a deductive framework to derive the algorithm concept from the program structure. Compared to other programming tools, the

suggestions provided by PPA are more detailed and assertive. The solution, for example, may provide an alternative algorithm for the code section under inspection.

The Parallelization Guru and the Performance Consultant basically tell the user where the problem is, whereas the expert system in PPA takes the role of a programming tool a step further toward an active guidance system. However, the knowledge base for PPA's expert system relies on an understanding of the underlying algorithm based on pattern matching. Having an algorithm-based expert system applicable to general parallel algorithms is impractical.

Our approach is different from the others, in that it is based on a flexible system controlled by a performance map, which any expert programmer can write. An experienced user states relationships between common performance problems, characterization data signatures that may indicate sources of the problem, and possible solutions related to these signatures. The performance map may contain complex calculations and evaluations and therefore can act flexibly as either or both of a performance advisor and an analyzer. In order to select appropriate data items and reason about them, a pattern matching module and an expression evaluation utility are provided by the system. Experienced programmers can use this system to help novice programmers in many different aspects in parallel programming, allowing an efficient transfer of knowledge to inexperienced programmers. For example, if a novice programmer encounters a loop that does not perform well, the user may activate a performance advisor to see the expert's suggestions on such phenomenon. Our system does not stop at pointing to problematic code segments. It presents users with possible causes and solutions to the best of its knowledge.

3 MERLIN: Performance Advisor

MERLIN is a graphical user interface utility that allows users to perform automated analysis of program characterization and performance data. This data can include dynamic information such as loop timing statistics and hardware performance statistics, as well as compiler-generated data such as control flow graphs and listings of statically applied techniques. It can be invoked from the URSA MINOR performance evaluation tool [1]. The activation of MERLIN is as simple as clicking a mouse on a problematic program section from this tool.

Through compilation, simulation, and execution, a user gathers various types of data regarding a number of different code blocks within a target program. Upon activation, MERLIN performs various analysis techniques on the data at hand and presents its conclusions to the user. Figure 1 shows an instance of the MERLIN interface. It consists of an analysis text area, an advice text area, and buttons. The analysis text area displays the diagnostics MERLIN has performed on the selected program unit. The advice text area provides MERLIN's solution to the detected problems with examples, if any. Diagnosis and the corresponding advice are paired by a number (such as `Analysis 1-2`, `Solution 1-2`).

MERLIN navigates through a database that contains knowledge on diagnosis and solutions for cases where certain performance goals are not achieved. Experienced programmers write performance maps based on their knowledge, and novice programmers can view their suggestions by activating MERLIN. Figure 2 shows the structure of a typical map used by this framework. It consists of three "domains." The elements in the *Problem Domain* corresponds to general performance problems from the viewpoint of programmers. They can represent a poor speedup, a large number of stalls, etc., depending upon the performance data types targeted by the performance map writer. The *Diagnostics Domain* depicts possible causes of these problems, such

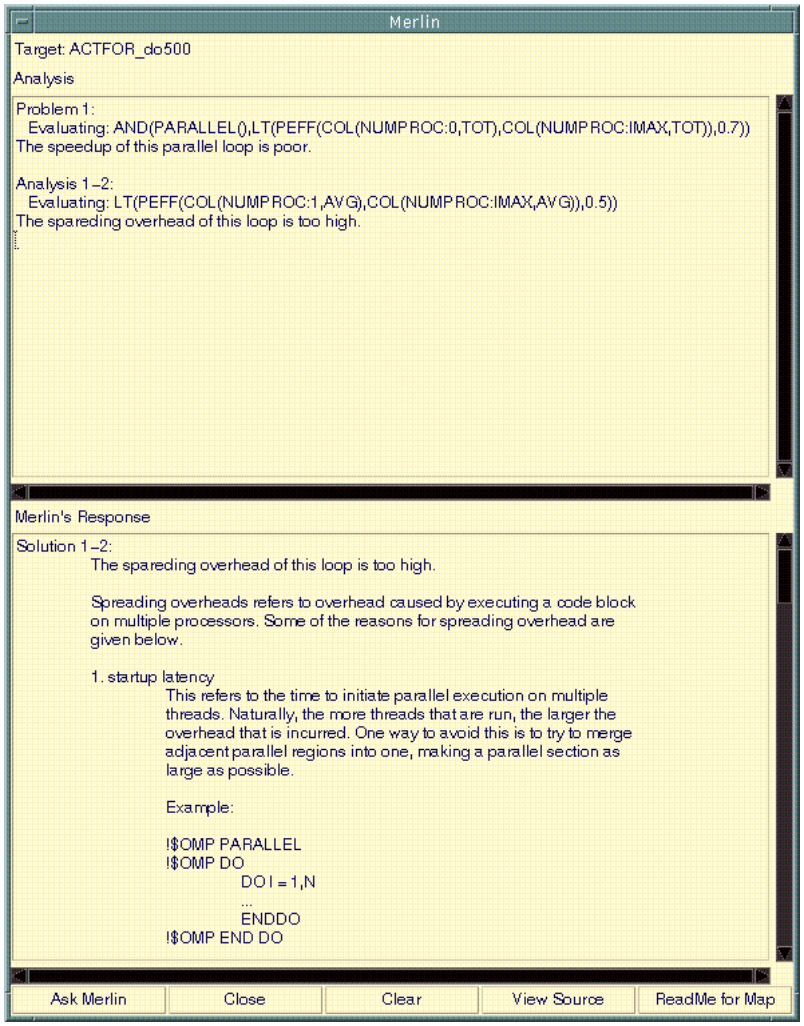

Figure 1: The user interface of MERLIN in use. MERLIN provides the solutions to the detected problems. This example shows the problems addressed in loop `ACTFOR DO500` of program `BDNA`. The button labeled `Ask Merlin` activates the analysis. The `View Source` button opens the source viewer for the selected code section. The `ReadMe for Map` button pulls up the ReadMe text provided by the performance map writer.

as floating point dependencies, data cache overflows, etc. Finally, the *Solution Domain* contains possible remedies. These elements are linked by *Conditions*. Conditions are logical expressions representing an analysis of the data. If a condition evaluates to true, the corresponding link is taken, and the element in the next domain pointed to by the link is explored. MERLIN invokes an expression evaluator for the evaluation of these expressions. When it reaches the Solutions domain, the suggestions given by the map writer are displayed and MERLIN moves on to the next element in the Problem domain.

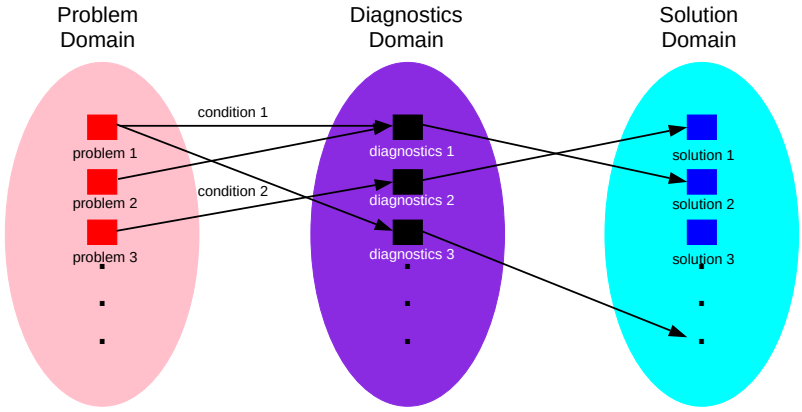

Figure 2: The internal structure of a typical MERLIN performance map. The Problem Domain corresponds to general performance problems. The Diagnostics Domain depicts possible causes of the problems, and the Solution Domain contains suggested remedies. Conditions are logical expressions representing an analysis of the data.

For example, in the default map of the URSA MINOR tool, one element in the Problem domain is entitled "poor speedup." The condition for this element is "the loop is parallel and the parallel efficiency is less than 0.7." The link for this condition leads to an element in the Diagnostics Domain labeled "poor speedup symptoms" with conditions that evaluate the parallelization and spreading overheads. When these values are too high, the corresponding links from these conditions points to suggestions for program tuning steps, such as serialization, fusion, interchange, and padding. The data items needed to compute this expression are fetched from URSA MINOR's internal database using the pattern matching utility. If needed data are missing, (e.g., because the user has not yet generated hardware counter profiles,) MERLIN displays a message and continues with the next element. The performance map is written in URSA MINOR's generic input text format [1]. It is structured text of data descriptions that can be easily edited, so map writers can use any text editor. MERLIN reads this file and stores it internally. When a user chooses a loop for automatic analysis, MERLIN begins tracing the conditions in the Problem domain.

MERLIN differs from conventional spreadsheet macros in that it is capable of comprehending static analysis data generated by a parallelizing compiler. MERLIN can take into account numeric performance data as well as program characterization data, such as parallel loops detected by the compiler, the existence of I/O statements, or the presence of function calls within a code block. This allows a comprehensive analysis based on both performance and static data available for the code section under consideration.

A MERLIN map enables efficient cause-effect analyses of performance and static data. It fetches the data specified by the map from the URSA MINOR tool, performs the listed operations and follows the links if the conditions are true. There are no restrictions on the number of elements and conditions within each domain, and each link is followed independently. Hence, multiple perspectives can be easily incorporated into one map. For instance, stalls may be caused by poor locality, but it could also mean a floating point dependence in the pipeline CPU. In this way, MERLIN considers all possible causes for performance problems separately and presents an inclusive set of solutions to its users. At the same time, the remedies suggested by MERLIN assist users in "learning by example." MERLIN enables users to gain expertise in an efficient manner by listing performance data analysis steps and many example solutions given by experienced programmers.

MERLIN is able to work with any performance map as long as it is in the proper format. Therefore, the intended focus of performance evaluation may shift depending on the interest of the user group. For instance, the default map that comes with MERLIN focuses on a performance model based on parallelization and spreading overhead. Should a map that focuses on architecture be developed and used instead, the response of MERLIN will reflect that intention. Furthermore, the URSA MINOR environment does not limit its usage to parallel programming. URSA MINOR coupled with MERLIN can be used to address many topics of the optimization processes in various engineering practices.

As mentioned above, MERLIN is accessed through the URSA MINOR performance evaluation tool [1]. The main goal of URSA MINOR is performance optimization through the interactive integration of performance evaluation with static program analysis information. It collects and combines information from various sources, and its graphical interface provides selective views and combinations of the gathered data. URSA MINOR consists of a database utility, a visualization system for both performance data and program structure, a source searching and viewing tool, and a file management module. URSA MINOR also provides users with powerful utilities for manipulating and restructuring the input data to serve as the basis for the users' deductive reasoning. URSA MINOR can present to the user and reason about many different types of data (e.g., compilation results, timing profiles, hardware counter information), making it widely applicable to different kinds of program optimization scenarios. The ability to invoke MERLIN greatly enhances the functionality of URSA MINOR.

4 Case Study 1: Hardware Counter Data Analysis

In our first case study, we discuss a performance map that uses the *speedup component model* introduced in [12]. The model fully accounts for the gap between the measured speedup and the ideal speedup in each parallel program section. This model assumes execution on a shared-memory multiprocessor and requires that each parallel section be fully characterized using hardware performance monitors to gather detailed processor statistics. Hardware monitors are now available on most commodity processors.

With hardware counter and timer data loaded into URSA MINOR, users can simply click on a loop from the URSA MINOR table view and activate MERLIN. MERLIN then lists the numbers corresponding to the various overhead components responsible for the speedup loss in each code section. The displayed values for the components show overhead categories in a form that allows users to easily see why a parallel region does not exhibit the ideal speedup of p on p

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 completed.	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.	

processors. MERLIN then identifies the dominant components in the loops under inspection and suggests techniques that may reduce these overheads. An overview of the speedup component model and its implementation as a MERLIN map are given below.

4.1 Performance Map Description

The objective of our performance map 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.

Memory stalls reflect latencies incurred due to cache misses, memory access times and network congestion. MERLIN will calculate the cycles lost due to these overheads. If the percentage of time lost is large, locality-enhancing software techniques will be suggested. These techniques include optimizations such as loop interchange, loop tiling, and loop unrolling. We found, in [13], that loop interchange and loop unrolling are among the most important techniques.

Processor stalls account for delays incurred processor-internally. These include branch mispredictions and floating point dependence stalls. Although it is difficult to address these stalls directly at the source level, loop unrolling and loop fusion, if properly applied, can remove branches and give more freedom to the backend compiler to schedule instructions. Therefore, if processor stalls are a dominant factor in a loop's performance, MERLIN will suggest that these two techniques be considered.

Code overhead corresponds to the time taken by instructions not found in the original serial code. A positive code overhead means that the total number of cycles, excluding stalls, that are consumed across all processors executing the parallel code is larger than the number

used by a single processor executing the equivalent serial section. These added instructions may have been introduced when parallelizing the program (e.g., by substituting an induction variable) or through a more conservative parallel code generating compiler. If code overhead causes performance to degrade below the performance of the original code, MERLIN will suggest serializing the code section.

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. Thread management latencies can be reduced through highly-optimized runtime libraries and through improved balancing schemes of threads with uneven workloads. MERLIN will suggest improved load balancing if this component is large.

Ursa Minor combined with this MERLIN map displays (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 the parallel code. 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 [12].

4.2 Experiment

For our experiment we translated the original source into OpenMP parallel form using the Polaris parallelizing compiler [14]. The source program is the Perfect Benchmark `ARC2D`, which is parallelized to a high degree by Polaris.

We performed our measurements on a Sun Enterprise 4000 with six 248 MHz UltraSPARC V9 processors, each with a 16KB L1 data cache and 1MB unified L2 cache. Each code variant was compiled by the Sun v5.0 Fortran 77 compiler with the flags `-xtarget=ultra2 -xcache=16/32/1:1024/64/1 -O5`. For hardware performance measurements, we used the available hardware counter (`TICK` register) [15].

`ARC2D` consists of many small loops, each of which has a few milli-seconds average execution time. Figure 3 shows the overheads in the loop `STEPFX DO210` of the original code, and the speedup component graphs generated before and after applying a loop interchange transformation.

MERLIN calculates the speedup component model using the data collected by a hardware counter, and displays the speedup component graph. Figure 3 (b) shows the speedup component graph of the Polaris-parallelized code. It shows that *code overhead* degrades the speedup by a factor of 2. MERLIN suggests several recipes to reduce this overhead. In addition, MERLIN's advice is to reduce the memory latency. This is based on an additional performance map that detects that the program spends more than 25% of its total execution time in memory stalls. From several suggested recipes the user tries loop interchanging, which results in significant, now superlinear speedup. Figure 3 (c) shows that the memory stall component has become negative, which means that there are fewer stalls than in the original, serial program. The negative component explains why there is superlinear speedup. The speedup component model further shows that the code overhead component has drastically decreased from the original parallelized

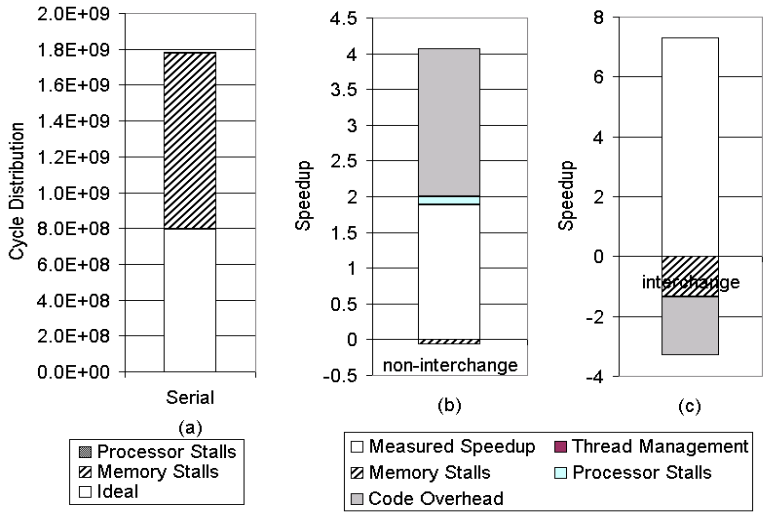

Figure 3: Performance analysis of the loop `STEPFX DO210` in program `ARC2D`. The graph (a) on the left shows the overhead components in the original, serial code. The graphs (b) and (c) show the speedup component model for the parallel code variants on 4 processors before and after loop interchanging is applied. Each component of this model represents the change in the respective overhead category relative to the serial program. MERLIN is able to generate the information shown in these graphs.

program. The code is even more efficient than in the serial program, further contributing to the superlinear speedup.

In this example, the use of the performance map for the speedup component model has significantly reduced the time spent by a user analyzing the performance of the parallel program. It has helped explain both the sources of overheads and the sources of superlinear speedup behavior.

5 Case Study 2: Simple Techniques to Improve Performance

In this section, we present a performance map based solely on execution timings and static compiler information. Such a map requires program characterization data that a novice user can easily obtain. The map is designed to advise novice programmers in improving the performance of programs achieved by a parallelizing compiler such as Polaris [14]. Parallelizing compilers significantly simplify the task of parallel optimization, but they lack knowledge of the dynamic program behavior and have limited analysis capabilities. These limitations often lead to marginal performance gains. Therefore, good performance from a parallel application is often achieved by a substantial amount of manual tuning. In this case study, we assume that novice programmers have used a parallelizing compiler as the first step to optimize the performance of the target program and that its static analysis information is available. The performance map presented in this section aims at increasing this initial performance.

Table 2: Optimization technique application criteria.

Techniques	Criteria
Serialization	speedup < 1
Loop Interchange	# of stride-1 accesses $<$ # of non stride-1 accesses
Loop Fusion	speedup < 2.5

5.1 Performance Map Description

Our goal in this study is to provide users with a set of simple techniques that may help enhance the performance of a parallel program based on data that can be easily generated. This includes timing and static program analysis data.

Based on our experiences with parallel programs, we have chosen four techniques that are (1) easy to apply and (2) may yield considerable performance gain. These techniques are serialization, loop interchange, and loop fusion. They are applicable to loops, which are often the focus of the shared memory programming model. All of these techniques are present in modern compilers. However, compilers may not have enough knowledge to apply them most profitably [13], and some code sections may need small modifications before the techniques become applicable automatically.

We have devised criteria for the application of these techniques, which are shown in Table 2. If the speedup of a parallel loop is less than 1, we assume that the loop is too small for parallelization or that it required extensive modification. Serializing it prevents performance degradation. Loop interchange may be used to improve locality by increasing the number of stride-1 accesses in a loop nest. Loop interchange is commonly applied by optimizers; however, our case study shows many examples of opportunities missed by the backend compiler. Loop fusion can likewise be used to increase both granularity and locality. The criteria shown in Table 2 represent simple heuristics and do not attempt to be an exact analysis of the benefits of each technique. We simply assumed the threshold of the speedup as 2.5 to apply the loop fusion.

5.2 Experiment

We have applied these techniques based on the criteria presented above. We have used the same machine as in Section 4. The OpenMP code is generated by the Polaris OpenMP backend. The results on five programs are shown. They are `SWIM` and `HYDRO2D` from SPEC95, `SWIM` from SPEC2000, and `ARC2D` and `MDG` from the Perfect Benchmarks. We have incrementally applied these techniques starting from serialization. Figure 4 shows the speedup achieved by the techniques. The improvement in execution time ranges from -1.8% for fusion in `ARC2D` to 38.7% for loop interchange in `SWIM'2000`. For `HYDRO2D`, application of the MERLIN suggestions did not noticeably improve performance.

Among the codes with large improvement, `SWIM` from SPEC2000 benefits most from loop interchange. It was applied under the suggestion of MERLIN to the most time-consuming loop, `SHALOW DO3500`. Likewise, the main technique that improved the performance in `ARC2D` was loop interchange. `MDG` consists of two large loops and numerous small loops. Serializing these

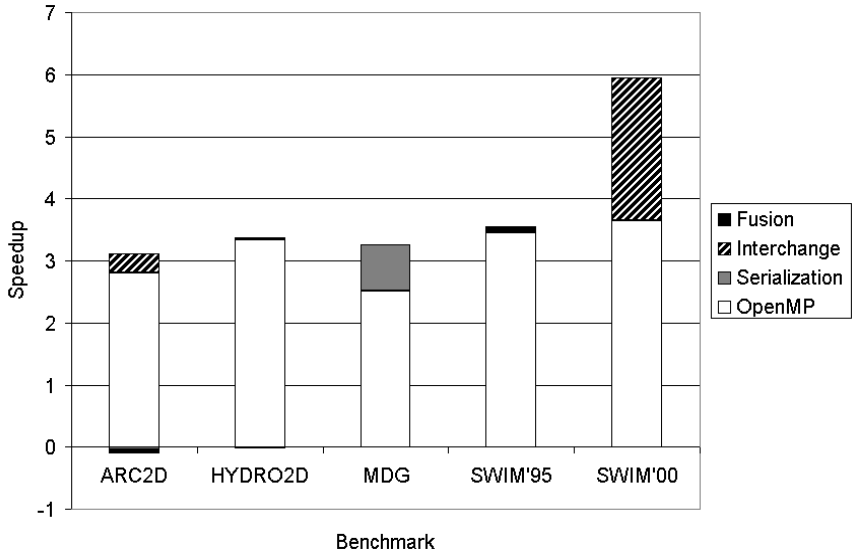

Figure 4: Speedup achieved by applying the performance map. Detailed numbers can be seen in Table 3. The speedup is with respect to one-processor run with serial code on a Sun Enterprise 4000 system. Each graph shows the cumulative speedup when applying each technique.

small loops was the sole reason for the performance gain. Table 3 shows a detailed breakdown of how often techniques were applied and their corresponding benefit.

Using this map, considerable speedups are achieved with relatively small effort. Novice programmers can simply run MERLIN to see the suggestions made by the map. The map can be updated flexibly without modifying MERLIN. Thus if new techniques show potential or the criteria needs revision, expert programmers can easily incorporate changes.

6 Conclusions

We have presented a framework and a tool, MERLIN, that addresses an important open issue in parallel programming: Guiding novice programmers in the process of tuning parallel program performance. It is a utility with a graphical user interface that allows inexperienced programmers to examine suggestions made by expert programmers on various subjects. While equipped with a powerful expression evaluator and pattern matching utility, MERLIN's analysis and advice are entirely guided by a performance map. Any experienced programmers can write a performance map to help new programmers in gaining experience. MERLIN is accessed through a performance evaluation tool, so performance visualization and data gathering are done in conjunction with this performance advisor.

We have presented two case studies that utilize MERLIN. In the first study, MERLIN is used to compute various overhead components for investigating performance problems. In the second study, we have introduced a simplified performance map that can still effectively guide users in improving the performance of real applications. The results show that the relatively small effort to run MERLIN can lead to a significant gain in speedup.

With the increasing number of new users of parallel machines, the lack of experience and

Table 3: A detailed breakdown of the performance improvement due to each technique.

Benchmark	Technique	Number of Modifications	% Improvement
ARC2D	Serialization	3	-1.55
	Interchange	14	9.77
	Fusion	10	-1.79
HYDRO2D	Serialization	18	-0.65
	Interchange	0	0.00
	Fusion	2	0.97
MDG	Serialization	11	22.97
	Interchange	0	0.00
	Fusion	0	0.00
SWIM'95	Serialization	1	0.92
	Interchange	0	0.00
	Fusion	3	2.03
SWIM'00	Serialization	0	0.00
	Interchange	1	38.69
	Fusion	1	0.03

transfer of programming knowledge from advanced to novice users is becoming an important issue. A novel aspect of our system is that it alleviates these problems through automated analysis and interactive guidance. With advances in performance modeling and evaluation techniques as exemplified in this paper, parallel computing can be made amenable to an increasingly large community of users.

References

[1] Insung Park, Michael J. Voss, Brian Armstrong, and Rudolf Eigenmann. Parallel programming and performance evaluation with the URSA tool family. *International Journal of Parallel Programming*, 26(5):541–561, November 1998.

[2] J. Brown, A. Geist, C. Pancake, and D. Rover. Software tools for developing parallel applications. 1. code development and debugging. In *Proc. of Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[3] J. Brown, A. Geist, C. Pancake, and D. Rover. Software tools for developing parallel applications. 2. interactive control and performance tuning. In *Proc. of Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[4] Michael T. Heath. Performance visualization with ParaGraph. In *Proc. of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 221–230, May 1994.

[5] Daniel A. Reed. Experimental performance analysis of parallel systems: Techniques and open problems. In *Proc. of the 7th Int' Conf on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 25–51, 1994.

[6] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[7] J. Yan, S. Sarukkai, and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software-Practice and Experience*, 25(4):429–461, April 1995.

[8] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.

[9] B. Topol, J. T. Stasko, and V. Sunderam. PVaniM: A tool for visualization in network computing environments. *Concurrency Practice and Experience*, 10(14):1197–1222, December 1998.

[10] W. Liao, A. Diwan, R. P. Bosch Jr., A. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Proc. of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48, August 1999.

[11] K. C. Li and K. Zhang. Tuning parallel program through automatic program analysis. In *Proc. of Second International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 330–333, June 1996.

[12] Seon Wook Kim and Rudolf Eigenmann. Detailed, quantitative analysis of shared-memory parallel programs. Technical Report ECE-HPCLab-00204, HPCLAB, Purdue University, School of Electrical and Computer Engineering, 2000.

[13] 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.

[14] 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.

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