Compiler-Based Tools for Analyzing Parallel Programs

Brian Armstrong Seon Wook Kim Insung Park Michael Voss
Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University *

Abstract

In this paper, we present several tools for analyzing parallel programs. The tools are built on top of a compiler infrastructure, which provides advanced capabilities for symbolic program analysis and manipulation. The tools can display characteristics of a program and relate this information to data gathered from instrumented program runs and other performance analysis tools. They also support an interactive compilation scenario, giving the user feedback on how the compilation process performed and how to improve it. We will present case studies demonstrating the tool use. These include the characterization of an industrial application and the study of new compiler techniques and portable parallel languages.

1 Introduction

Compilers play a crucial role in the development of parallel programs. In addition to their task of translating and optimizing programs, they provide useful infrastructures for building tools. Examples of such tools are program analysis instruments, which can gather performance information, program structure information, and statistics. Other examples are tools that can manipulate programs for various purposes, such as timing instrumentation, and augmentation for execution-driven simulation.

One objective of our work is to integrate performance analysis tools and compilers more closely. The term "tools" is commonly used for facilities that assist a user in gathering performance data and visualizing them. Examples of such tools are Pablo [Ree94], Paradyn [MCC+95], and PTOPP [EM93]. Although these tools could benefit from the advanced program analysis capabilities built into optimizing compilers, this level of integration does not yet exist.

Another objective of this paper is to advocate compilers themselves to be viewed as tools that interact with the programmer. Such tools can facilitate the interactive invocation of transformation passes, as was proposed in several previous projects [ASM89, BKK+89]. In addition, in out project, we consider interactivity by making the results of the compiler analysis accessible to the user in adequate forms. This leads to a better understanding of the program characteristics, the compilation process, and the resulting performance. The user, in turn, can improve the source code, better steer the compilation process, or directly modify the compiler-translated parallel program.

The tools presented in this paper are based on the Polaris compiler infrastructure. An overview of the Polaris project is given in [BDE+96]. This project provides both the environment and motivation of the described work. Polaris' primary goal is to advance the state of the art of automatic program parallelization. The project maintains a compiler, representing this state, and facilitates its use by the research community. In addition to many commonly known compilation techniques, Polaris includes advanced

*This work was supported in part by U. S. Army contract #DABT63-92-C-0033, by the NSF "petaflop point design" grant #9612133, and an NSF CAREER award. This work is not necessarily representative of the positions or policies of the U. S. Army or the Government.

capabilities for array privatization, symbolic and nonlinear data dependence testing, idiom recognition, inter-procedural analysis, and symbolic program analysis. Polaris represents a general infrastructure for analyzing and manipulating Fortran programs. This infrastructure is being used by many compiler research projects world-wide.

The utility of the Polaris infrastructure for building parallel programming tools has not been discussed previously. Motivation to do so is twofold: First, Polaris is being used for many application studies. The desire for better facilities for understanding the compiler-gathered program analysis information has arisen in these efforts. Second, several (non-compiler) tools that are Polaris-based have been or are being implemented. Sections 2 and 3 will present such tools. The URSA MINOR tool enables the user to inspect the results of the compilation process and the program characteristics gathered therein. This information can be viewed together with the results of other tools, such as timing and parallelism analyzers. The MAX/P tool permits the analysis of the maximum parallelism available in the executing program. The PERFOR tool models program performance and enables the user to estimate its scaling behavior. Section 4 then presents scenarios that make use of the described tools. For this purpose we chose two of our closely-related projects. In the first study we analyze and characterize an industrial computational application. The second study analyzes a program for identifying new compilation techniques and for evaluating compiler target languages. We will conclude with a discussion of tool features that proved most useful and show where extensions are worthwhile.

2 Ursa Minor: a browser for compilation and performance results

The goal of the URSA MINOR project is to provide tools that assist parallel programmers in effectively writing and tuning codes [PVAE97] Access to information from various sources is given to the user in a comprehensible way. These sources include tools such as compilers, profilers, and simulators. Users interact with URSA MINOR through a graphical interface, which can provide selective views and combinations of the data. URSA MINOR is implemented in Java, potentially making it accessible through the web.

URSA MINOR can act as the overall environment, making available information from many tools, including the ones discussed in subsequent sections. Currently, the tool can display information such as timing results from various program runs, the number of invocations of each loop, the calling structure, and the maximum degree of parallelism provided by MAX/P (described in Section 3.1). It also indicates whether a loop is serial or parallel as detected by Polaris. If it is serial, the reason given by the compiler can be displayed on-demand. Various displays are supported. For example, the data can be shown in a table format, each line being the name of an individual loop. Figure 1 shows such a view.

Another view provides the calling structure of a given program, which includes subroutine, function, and loop nest information as shown in Figure 2. Each rectangle represents either a subroutine, function, or loop. Clicking one of these will display the corresponding source code. In Figure 2 the user is inspecting the loop `ACTFOR_do240` in this way.

One option is to save the data in a form understood by generic spread-sheet programs. In Figure 3 we have read this form into the XESSS3 spread-sheet program. This allows one to exploit its many capabilities, such as diverse graphical representations. In Figure 3 the user has chosen an execution time graph for program `BDNA`, comparing the performance of Polaris with the compiler from Sun Microsystem. The URSA MINOR tool is especially useful for those users developing parallel programs, whereby the compiler-detected parallelism serves as a starting point. The timing information points out loops that do not yet exploit parallel processors well. If the loop is serial, the user can look at the reason Polaris failed to detect parallelism, which helps to find the right remedy. The maximum degree of parallelism shows how much potential for improvement there is in the loop. If the loop is nested in a larger loop, the user may focus on this larger loop. On the other hand, if it seems impossible to parallelize the loop, the user may inspect some of its time-consuming inner loops.

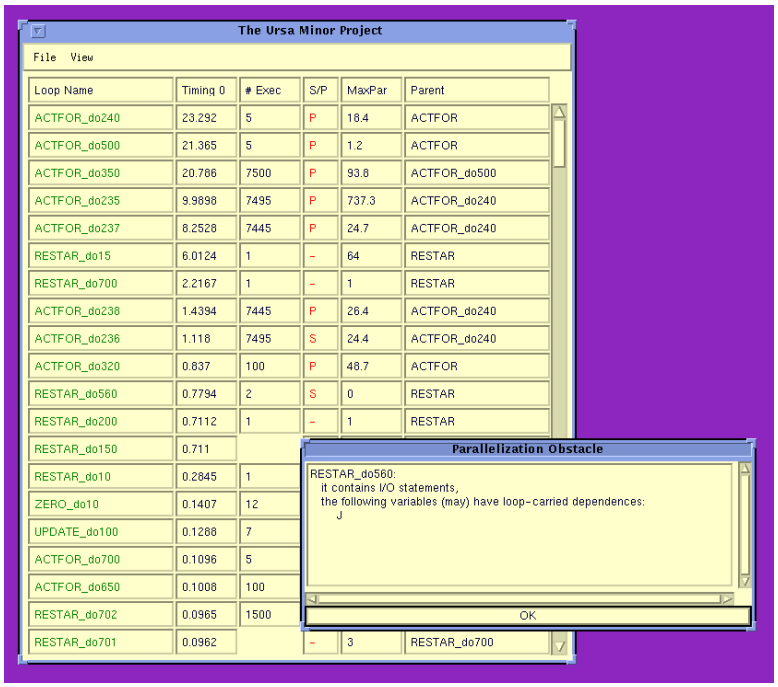

Figure 1: Loop Table View of the URSA MINOR Tool. Each column represents different data associated with the loops of the inspected program. The user has clicked on the loop `RESTAR_do560` to see the reason that inhibited parallelization.

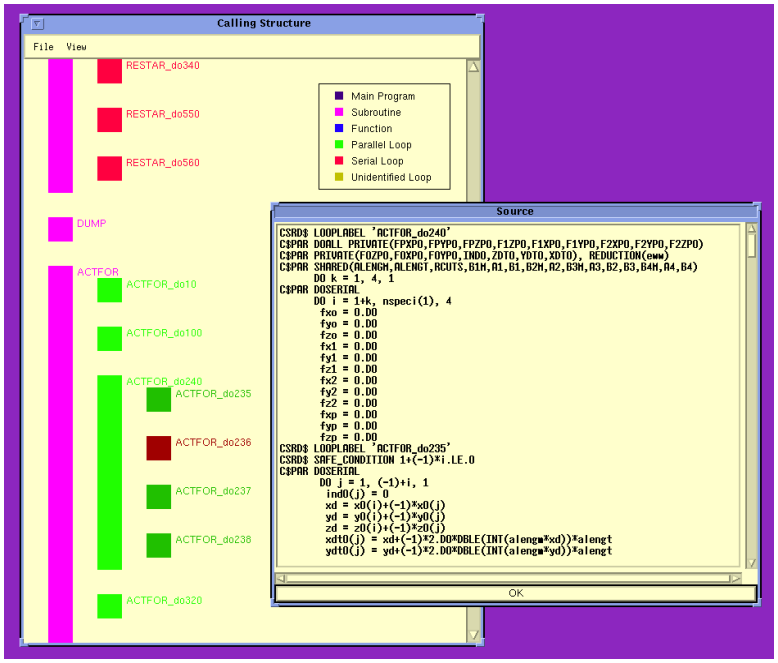

Figure 2: Annotated Call Graph View of the URSA MINOR Tool. The color-coded rectangles represents either a loop, a subroutine, or a function. The user has clicked on the loop `ACTFOR_do240` to see the source code with parallel directives.

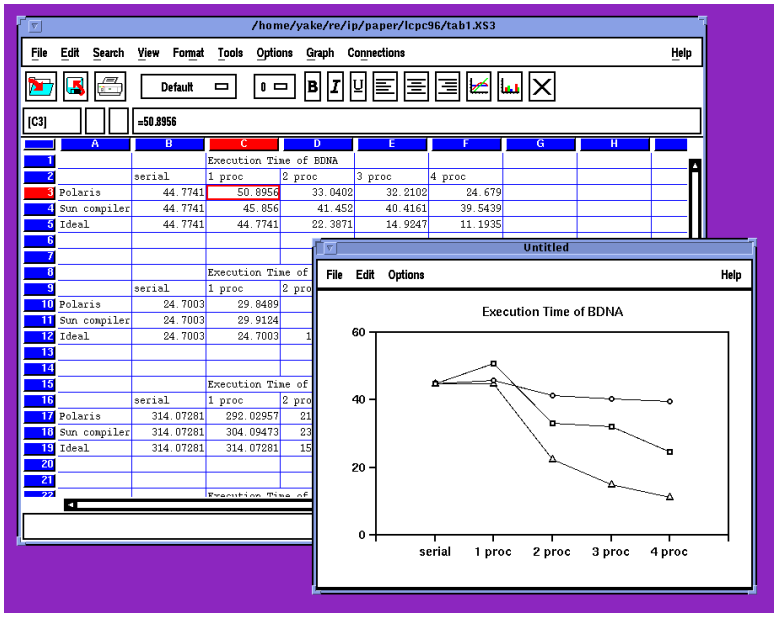

Figure 3: Spread-Sheet View of the Database Exported from the URSA MINOR Tool. The user has drawn a chart with the data obtained from three different runs of program `BDNA`. The lowest line in the chart shows ideal execution time.

The URSA MINOR tool collects and combines information from various sources. Timing information is gathered from instrumented program runs. The tool performing this instrumentation is another Polaris-based utility. Maximum parallelism estimates are supplied by the MAX/P tool. Information on which loops are serial or parallel is provided by the actual Polaris compiler. The URSA MINOR tool includes a subroutine and loop calling structure analyzer, also implemented using the Polaris infrastructure. Several modes are provided to read this information from the various original files, add to the existing information incrementally, store the entire database, or read from a previously saved database.

The URSA MINOR tool uses the basic Java language with standard Application Programming Interfaces (API's), allowing the tool to be portable. The windowing toolkits and utilities provided by Java enable us to focus on the practicality of the tool. One problem with using Java is that it does not allow local file access when run over the internet. This means that users cannot use URSA MINOR via the internet to operate on their own compilation and performance data. However the tool *can* be used to provide to remote users access to the database of program information at the tool home site. Creating and making available such a database of characteristics and performance data of a variety of benchmark programs is one of our goals.

3 Tools for projecting maximum parallelism and scaled performance

In this section we present tools that can look beyond current limitations of compilers and architectures. They may answer questions such as whether parallelism exists in a seemingly serial program section, and whether the performance of a program will scale up well for large numbers of processors and large input data sets. Two tools that can answer these very questions are MAX/P and PERFOR. Both tools make use of the Polaris infrastructure. The tools can run in stand-alone mode or can provide their results to the URSA MINOR environment for visualization.

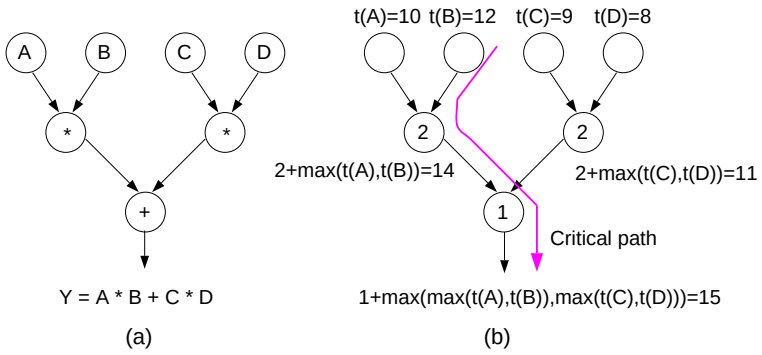

Figure 4: Computing *Earliest Available Time* for the Assignment, `Y = A*B+C*D`. (a) Operations, (b) Time Computation. The circles represent the operators and operands of a program statement, and the arrows represent the dependencies. The earliest available time for `Y` is 15, given that a multiplication takes 2 steps, an addition takes 1 step, and the operands `A, B ,C, D` are available at times 10, 12, 9, 8, respectively.

3.1 MAX/P: detecting maximum parallelism

A compiler is able to analyze the static behavior of a program. It can find characteristics of a program that are true for all possible input data sets and target machines. In contrast, the dynamic evaluation of a program can provide insights into the characteristics of programs and the predictions of behaviors that may be undetected by static analysis methods. Of great interest is understanding the dynamic behavior of parallelism, one of the most dominant factors of performance.

MAX/P is a Polaris-based tool, originally developed at the University of Illinois [KE97, Pet93]. It evaluates the inherent parallelism of a program at runtime. The inherent parallelism is defined as the ratio of the total number of operations in a program, or program section, to the number of operations along the critical path. The critical path is the longest path in the program's dataflow graph, which is computed by MAX/P during the program execution. The tool can find the minimum execution time of a program as it would execute with an unlimited number of parallel processors.

MAX/P operation

MAX/P operates in two steps. First, it instruments a Fortran source code and, second, it executes this program to evaluate the dynamic behavior. In essence it performs an execution-driven simulation, whereby it detects the dynamic parallelism. Figure 4 shows the concept of parallelism detection. MAX/P records, for each program variable, the *earliest available* time. When a variable is assigned a value, this time is computed as the maximum of the time recorded for each operand of the assignment plus the time taken by the operations. MAX/P determines from this information the maximum possible parallelism at time point `t` as the total number of assignments that have recorded time `t`.

MAX/P generates loop and routine-level summaries. They include execution times (the number of operations,) the number of loop invocations, the number of loop iterations, dependencies, and the degrees of parallelism.

MAX/P use and ongoing work

The MAX/P tool gives insight into the dynamic behavior, which cannot be detected by static analyses. It shows the maximum parallelism as an upper estimate for the potential performance gain that a user can expect from aggressively optimizing the code.

Of further interest is the comparison of parallelism detected by MAX/P and the Polaris parallelizer. In several programs we have found that Polaris detected parallelism very successfully, except for two cases: when Do-loops have multiple exits or entries, and when do-loops contain I/O statements. On

the other hand, Polaris sometimes detects parallelism where MAX/P does not, such as with loops that contain reduction operations. Polaris parallelizes such loops by taking advantage of the mathematical commutativity and associativity of reductions. Currently MAX/P does not include this capability.

In ongoing work we plan to combine the strengths of both tools. Implementing the reduction transformation in MAX/P is one step. In addition, Polaris' symbolic analysis capabilities can often find the exact value of program variables that determine the degree of parallelism. Our combined tool will detect parallelism statically where possible. It will then instrument for dynamic evaluation only the program parts where static detection of parallelism was not possible. In this way, the execution time of MAX/P can be reduced without any loss of information.

3.2 PERFOR: Predicting program scalability

Understanding the behavior of an application on large input datasets or large architectures is not only important for the application development but also for the advancement of computer systems. Unfortunately, evaluating the behavior of large-scope problems can be prohibitively time-consuming. The MAX/P tool, described previously, can take a week for fully evaluating an industrial-scope application. This is typical for a simulation tool. Facilities that can help reduce this time are crucial.

We are currently developing the "PERFOR" tool, which addressed this issue. It is being implemented using the Polaris infrastructure, as most tools described in this paper. Polaris can generate detailed information about an application, including the block-structure, code flow, numbers of instructions and their types, data dependencies, variable values, and loop ranges. Our tool will make use of this information for generating accurate estimates of how the application will perform and scale for systems that do not yet exist and for input data sets that would exhaust current machine resources.

Modeling performance

The PERFOR tool models an application's behavior in terms of resource usage functions. These functions express the loads the application places on the CPU, the disk, and the communication network. The tool determines these loads by analyzing the volumes of computation, communication, and I/O in the source program, and expressing the loads in terms of program input and architectural parameters. Basically, this is done by traversing all loop nests, determining the numbers of operations in each loop body and the loop iteration counts, and then expressing them in terms of input and architecture parameters. A thorough description of this model can be found in [AE97]. Section 4 gives an example of the resource usage functions generated for a Seismic processing application.

Implementing PERFOR using the Polaris infrastructure

Performance modeling with PERFOR relies on program knowledge, such as the loop nest structure, the loop iteration counts, operation counts, and the call sites of communication subroutines and disk accesses. Several Polaris-based tools support the PERFOR tool. The URSA MINOR tool provides subroutine and call graph information. Using Polaris' value propagation capability, the loop ranges can be described in terms of the parameters read from the input file. Then, one can formulate the functions expressing the nesting structure of the loops and their iteration numbers in terms of the application input parameters. Furthermore, operation counts in each loop body are gathered via a (Polaris-based) counting tool.

The PERFOR tool is currently being implemented. The goal is to fully automate the described performance model. In future work we plan to extend the model to larger application classes, hence relaxing some of the current restrictions regarding the analyzability of an application.

Table 1: Summary of Execution Times and Speedups of Seismic, Using MAX/P. The table shows inherent parallelism of the Seismic application. Serial and parallel execution times are given in terms of the number of operations on the critical execution path. Unlimited number of processors is assumed for the parallel execution.

Phase	Serial Execution Time	Parallel Execution Time	Speedup
1	7696662290	142189827	54.1
2	484867961	30706813	15.8
3	233098401	14781415	15.8

4 Tool use and case studies

We present two different scenarios using the tools described in the previous sections. In the first study we characterize the behavior and properties of a large-scope application, used in the Seismic industry. The second study aims at understanding the performance of several benchmark applications to determine compilation techniques that improve performance and to find languages that best serve as compiler target representations.

4.1 Analyzing maximum parallelism and performance scalability in a Seismic processing application

The Seismic application being studied is a program in the SPEChpc96 suite of real-application benchmarks [EH96]. It is a code used in the seismic processing industry for the search of oil and gas. The code consists of four applications ("phases"), which perform the seismic processes: "Data generation," "Stacking of data," "Time Migration," and "Depth migration." The entire code contains 20,000 lines of Fortran and C, and includes about 250 subroutines with intensive communication as well as intensive disk I/O. The benchmark includes 5 data sets, ranging from a small set, for testing purposes, to one larger than current machines can handle. For our measurements we use the smallest data set (approximately 100 megabytes in size,) providing program runs in the half-hour range, which is adequate for our purposes.

Parallelism analysis with MAX/P

MAX/P can instrument Fortran programs. One issue was to deal with the mix of Fortran and C languages used by Seismic. We did this by converting C routines to dummy statements, but which have the same dependencies and operation counts as the original ones. Then, after instrumentation, the original C programs were reinserted for the actual execution.

Table 1 shows the inherent parallelism of Seismic using the smallest (called *Test*) data set. Execution times are taken as the number of operations, and the speedup is defined as the ratio of the serial to the parallel execution time. The table shows the first three phases of Seismic. Phase 1 has more inherent parallelism than Phases 2 and 3. Obtaining this data is very time-consuming; Phase 1 in the *Test* data set, for example, takes approximately a week to run on a SPARC20 100 Mhz processor.

The processor activity histograms and the parallelism profile graphs can be drawn as shown in Figures 5 and 6 with the parallelism profiles generated by MAX/P. The shape of the processor activity histogram (PAH) helps to determine the homogeneity of parallel activity, and shows the processor utilization. The PAH describes the number of operations that can be executed concurrently, assuming an unlimited number of processors. Figure 5 (a), (b), and (c) show the PAH's in terms of maximum, minimum, and average number of operations in a certain time period. The PAH's reveal that Seismic's Phase 1 has a large amount of inherent parallelism. The execution time is divided into 512 time periods. Figure 5 (d) shows the effect of limited numbers of processors. The lower- and upper-bound speedup are the worst

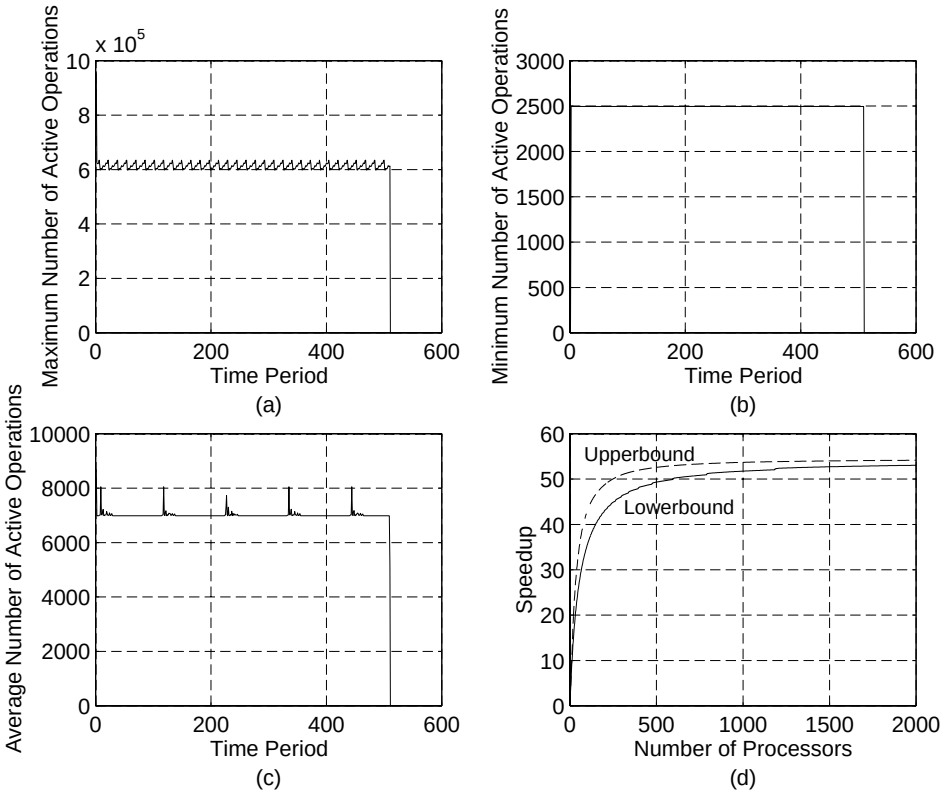

Figure 5: Parallelism of Phase 1 of the Seismic Processing Application with the *Test* Data Set. (a) Maximum Number of Active Operations. (b) Minimum Number of Active Operations. (c) Average Number of Active Operations. (d) Speedup. MAX/P gathers dynamic information such as inherent parallelism, and presents it in terms of processor activities and speedups.

and the best cases for different scheduling methods. Speedups are saturated around 500 processors, and amount to about 50. These speedups do not include other overheads such as memory and communication latencies. A further discussion can be found in [KE97].

Figure 6 shows the procedure-level parallelism. Each tick on the x-axis represents a subroutine call. The graphs visualize the number of consecutive calls to a subroutine and their parallelism. This view is not yet fully supported by the tool.

Performance modeling and prediction with PERFOR

The goal of this study is to come to an understanding of the application's performance behavior for large numbers of processors and data sets that are larger than current machines can accommodate.

As described in Section 3.2, the PERFOR tool expresses computation, communication, and I/O volumes in terms of the application's input data parameters. The data space in Seismic consists of seismic samples in traces along a number of lines [MH93]. When executed on more than one processor, the lines are partitioned into groups of traces for each processor. Thus, the data size depends on the number of samples in a trace, the number of traces in a group, the number of groups in a line, and the number of lines.

The following equation gives an example of a function generated in the process of modeling Seismic's performance. (The reader is not expected to understand the function's semantics. Rather, it is meant to illustrate the kind and shape of expressions PERFOR generates internally.) The equation expresses

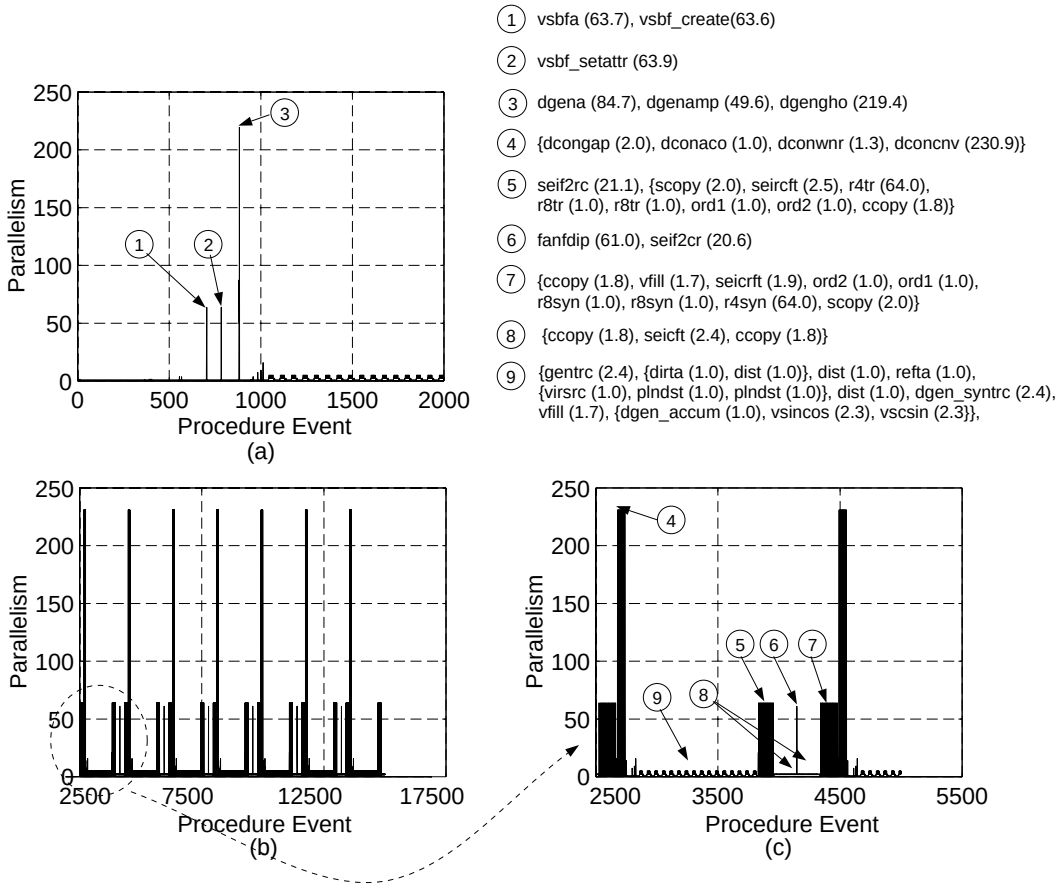

Figure 6: Procedure Parallelism of Phase 1 of Seismic with the *Test* Data Set. (a) Procedure calls $1-2000$. (b) Procedure calls $2500-15000$. (c) Procedure calls $2500-5000$. Procedure names are given and their degrees of parallelism are in parentheses. The x-axes of the graphs represent the sequence of procedure calls. The y-axes indicate the parallelism in each called procedure.

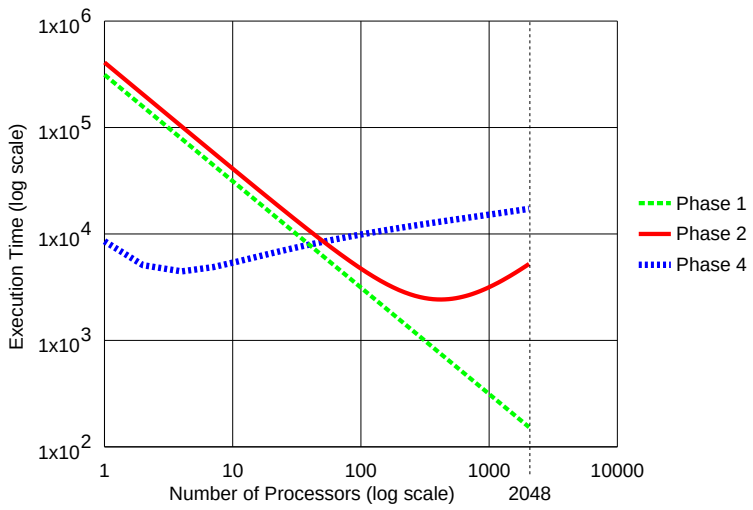

Figure 7: Extrapolations for Three Passes (Phases) in the Seismic Processing Application with a 3 TB Data Size. While Phase 1's execution time decreases steadily with increasing processor numbers, Phases 2 and 4 have an optimal number of processors, beyond which the performance degrades, due to communication and I/O.

the total number of computational operations for a given section of code. P is the number of processors, Ops_x is the number of operations found in the body of loop x, and I_y stands for the value of a particular program input parameter y. A full model consists of several functions similar to the one here. These functions make up an application's "signature," which abstracts the important aspects of computation, communication and I/O. Figure 7 gives an example "Performance Forecast Diagram" generated from these functions. Again, this figure merely illustrates the type of display and experiments enabled by PerFor. A thorough discussion of the experiments is beyond the scope of this paper.

$$Ops_{\texttt{Total}} = I_{\texttt{numLines}} \times \left(\begin{array}{l} \frac{I_{\texttt{tracesPerGroup}}}{P} \times \left(\begin{array}{ll} & Ops_{\texttt{STAK-LINE-1}} \\ + & I_{\texttt{samplesPerTrace}} \times Ops_{\texttt{STAK-LINE-1.1}} \end{array} \right) \\ + \quad P \left(\begin{array}{ll} & Ops_{\texttt{STAK-LINE-2}} \\ + & \frac{I_{\texttt{tracesPerGroup}}}{P} \times \left(\begin{array}{ll} & Ops_{\texttt{STAK-LINE-2.1}} \\ + & I_{\texttt{samplesPerTrace}} \times Ops_{\texttt{STAK-LINE-2.1.1}} \end{array} \right) \end{array} \right) \end{array} \right) \quad (1)$$

4.2 Tool use in compiler analysis and evaluation

The goal of our second study is to identify new optimizing compiler techniques. Compiler optimizations and transformations have to be driven by the needs of real applications. For identifying such needs, it is important for the compiler developer to be able to quickly and effectively characterize applications.

In a current project we are conducting such characterization studies. To illustrate the use of the presented tools we will describe an effort that compares parallel directive languages for their suitability as a portable compiler output representation. Another objective is to identify compiler transformations that can express programs in these output forms effectively. We will do this by comparing the machine-specific directive languages on the SGI Power Challenge and the Sun SPARC multiprocessors with the portable Guide directive set [KAI97]. Guide implements the parallel model proposed by the ANSI X3H5 committee and provides compilers for this language on most of today's available SMP platforms.

Figure 8 compares the Polaris-parallelized programs expressed in the Guide dialect with those expressed in the native directive set from Sun Microsystems. It also shows the performance of the code automatically parallelized by the native Sun parallelizer. The measurements are taken on a four processor SPARCstation 20 machine. The performance differences are now being analyzed to determine compiler transformations that work best for the portable Guide target language

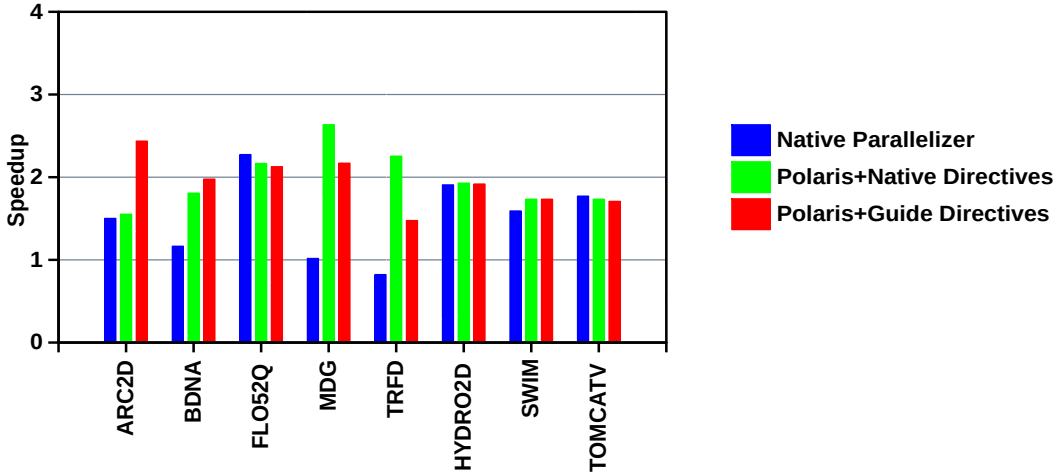

Figure 8: Speedups of Several Benchmark Programs on a 4 Processor SPARCstation 20. The three bars represent measurements of three different program variants. The blue bars are programs parallelized by the native compiler from Sun Microsystems. The green bars represent Polaris-parallelized programs expressed in the native Sun directives, while the red bars represent Polaris-parallelized programs expressed in Guide directives.

The most notable differences in the performance of these benchmarks are seen in the programs `ARC2D`, `BDNA`, `MDG` and `TRFD`, all of which are codes from the Perfect Club Benchmark suite. We will first describe several compilation issues that we identified as making a significant difference. Then we will describe our use of tools for these studies.

Analyzing the interdependence of compiler optimizations

One issue in using a portable directive set such as Guide, is to determine when a high-level transformation will make a positive performance difference. Even if the transformation is beneficial per se, it may have a secondary effect of either facilitating or inhibiting low-level optimizations performed by the back-end compiler. These secondary effects can have a significant performance impact. We will discuss techniques that influenced the back-end compiler's ability to analyze and optimize a program: forward substitution, induction variable substitution, and load-based scheduling. We will discuss the effect of these compilation techniques and the tools used for their investigation. Solutions to these problems involve both coordination strategies for compilation techniques and new language constructs for expressing performance-critical results of program analyses. However, this is beyond the scope of this paper and we refer to [Vos97].

Loop interchanging and forward substitution: In `ARC2D`, three of the most time consuming loop nests are `FILERX_do19`, `STEPFX_do210` and `STEPFX_do230`. Figure 9 shows that these loops make up approximately $\frac{1}{3}$ to $\frac{1}{2}$ of the total execution time of the program. The performance differences between the code versions are caused by the interplay of *forward substitution* and *loop interchanging*.

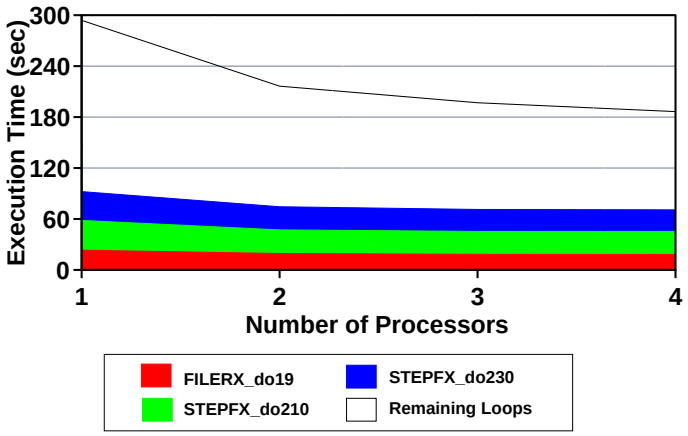

Figure 9: Cumulative Execution Time of the Most Time-Consuming Loops in ARC2D. These timings were taken of the code generated by the native Sun parallelizer and executed on a 4-processor SPARCstation 20.

In Figure 8, the performance of these different versions of `ARC2D` is given. The Polaris+Guide version significantly outperforms both other versions of the code. To get a better understanding of these trends, our tools are used to look at loop by loop timings.

The loop timings and speedups are shown in Figure 10. Examining loop nest `FILERX_do19`, it became apparent that, when using the Sun directive set, the loop interchanging transformation was not performed by the Sun back-end compiler. This interchanging, which was performed in both other versions of the code, ensures stride-1 accesses to arrays within the loop nest, increasing cache locality.

In the loops `STEPFX_do210` and `STEPFX_do230` neither the Polaris+Sun directives version nor the Autopar versions had loop interchanging applied. Looking at the original serial source code, it is seen that these two loop nests are imperfectly nested. The Sun Autopar version cannot interchange such loops. However, Polaris converts the loop into a perfect nest, for which the Sun backend compiler applies the interchanging. Polaris performs this transformation through a combination of *forward substitution* and *dead code elimination*. One issue with forward substitution is to decide when to apply it, as it may create expressions of increased complexity, hence create overhead. However, in the loops we inspected, forward substitution, combined with dead-code elimination, had a consistent positive effect.

Induction variable substitution The issue of the increased complexity of the resulting expressions is raised even higher when the compiler performs *induction variable substitution*. For example, Figure 11 shows a simple induction expression as well as a transformed, parallel form. Induction variable substitution must first recognize variables of this form and then substitute them with a closed form solution [PE95].

This transformation would allow the resulting loop shown in Figure 11 to be executed in parallel. Unfortunately, if there are many enclosing loops and complex induction variables, the closed form solutions may become rather large. In fact in the program `TRFD`, the induction variable closed forms are multiple lines of Fortran code long. The resulting performance of the code then becomes highly dependent on the back-end compiler's ability to optimize (i.e., reduce) these expressions. If little or no common subexpression elimination or strength reduction is done, a large overhead may result.

Figure 12 shows the execution time of `TRFD` on both a SPARCstation 20 and a Silicon Graphics (SGI) Power Challenge. On both machines these complex expressions cause overhead. The 1-processor parallel version executes approximately 2 times slower than the original serial version on both the SPARCstation and Power Challenge.

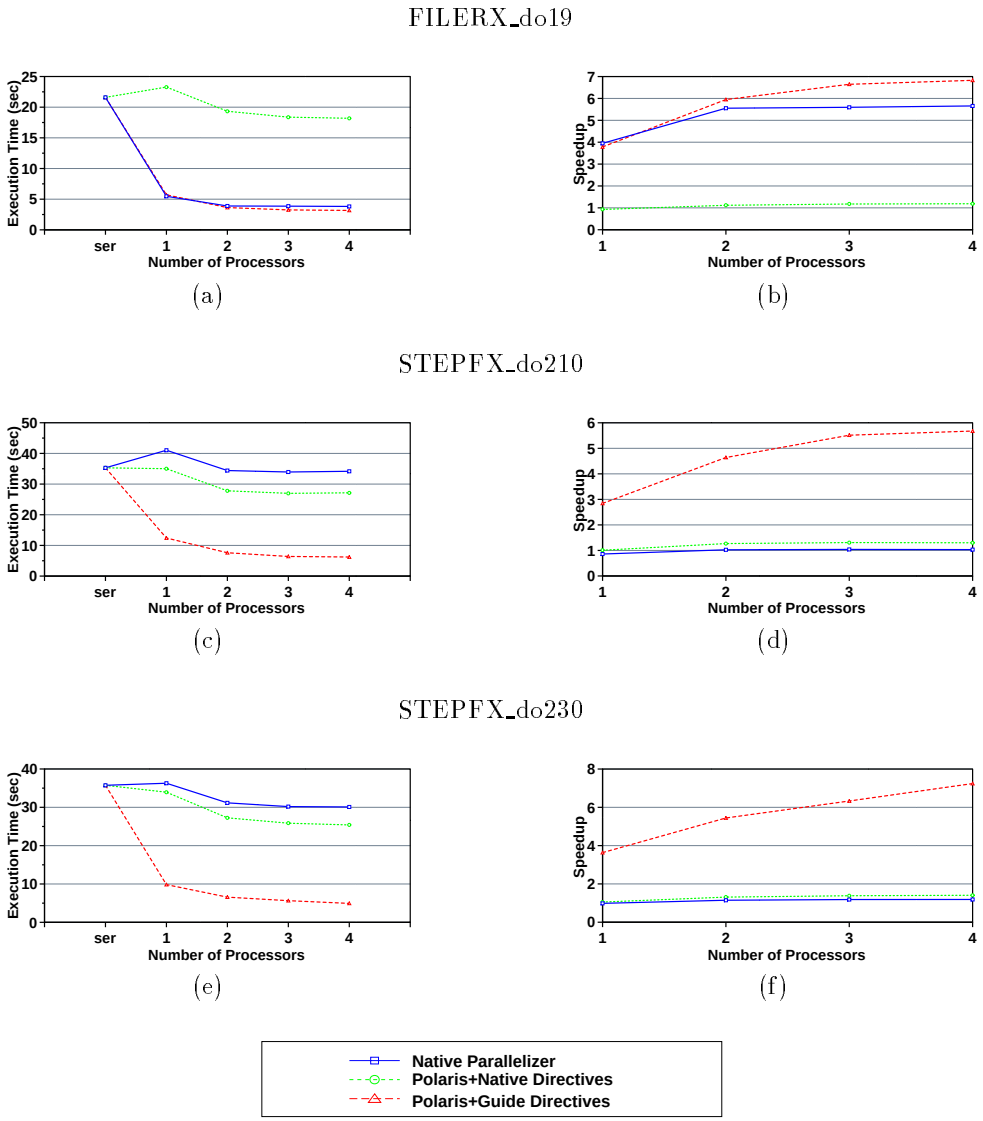

Figure 10: Performance of Important ARC2D Loops on the 4-Processor SPARCstation 20. Loop execution times and speedups are shown for the code as automatically parallelized by the Sun native parallelizer and for two Polaris-generated forms: one using the native Sun directive language and one using a portable Guide directive language

```
S1: X = 0
S2: DO I = 1,n
S3:       X = X + 2*I
S4:       A(X) = ...
S5: ENDDO
```

(a) Original

```
S1: X = 0
S2: DO I = 1,n
S3:       A(I + I**2) = ...
S4: ENDDO
```

(b) Transformed

Figure 11: An Example of Induction Variable Substitution. Generalized induction variables can lead to complex subscript expressions, potentially causing significant overhead.

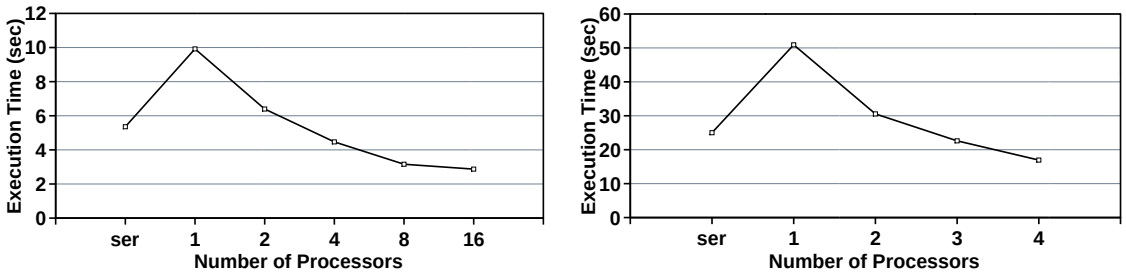

Figure 12: The Execution Time of TRFD: (a) on the 4 Processor SPARCstation 20 and (b) on the 16 Processor Power Challenge. On both machines there is a significant *parallelization overhead* when comparing the serial execution time with the 1-processor parallel variant.

These results are meaningful, because if one were to execute on just 2 processors of either machine, the overhead would cause a slowdown with respect to the serial version. Therefore parallelization would only be profitable if more than 2 processors were available.

The Autopar version of `TRFD` runs slower than the serial version. This is because the most time-consuming loops in this program need a combination of advanced induction variable recognition, and non-linear data dependence testing for their parallelization. Currently, only the Polaris compiler includes these capabilities.

Load-based scheduling

Loop scheduling methods accounted for another significant performance difference in the studied loops. This effect was caused by triangular loops. These are loop nests in which an inner loop iteration count depends on an outer loop index. Such loops have varying (increasing or decreasing) iteration times, potentially causing load-imbalance. For the programs `MDG` and `BDNA` the differences in performance shown in Figure 8 are primarily due to differences in scheduling techniques. In the original Guide output pass in Polaris, the default scheduling technique is always used. This technique is block scheduling with the block size equal to $\frac{1}{p}$ of the iteration space, where p is the number of participating processors. However, for generating Sun directives, Polaris chose interleaved scheduling, which lead to better performance in triangular loops.

For the two most time-consuming loops in `MDG`, the effect of changing the scheduling technique can be observed. Figure 13 demonstrates that the time spent in each processor is more balanced when using an interleaved technique (called *cyclic*). It further shows that a block size of 1 is best.

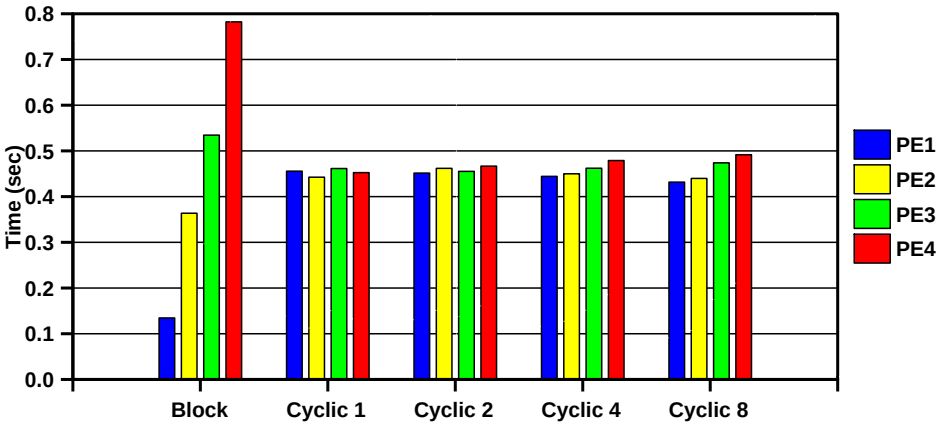

Figure 13: Per-Processor Execution Times of the `MDG` Main Loop on a 4-CPU SPARCstation 20, Using Different Scheduling Schemes (cyclic n = block cyclic with block size n). Triangular loops cause a significant loop imbalance when using block scheduling. Cyclic scheduling resolves this issue.

4.3 Important tool capabilities for program analysis

The tools described in this paper proved essential for characterizing computational applications, for identifying shortcomings of compiler techniques and parallel languages, and for finding potential improvements.

In our study we usually started by instrumenting a program with loop-by-loop timing code and generating execution profiles. By executing the serial program and the parallel program on various numbers of processors, we obtained information about the importance of loops, the overhead introduced by parallel transformations, and speedup figures. The tools helped us to quickly find important and related information, and combine the results from several versions of the code. This includes

- information about the most time-consuming loops,
- inner and outer subroutines and loops,
- data on what loops were parallelized or not parallelized,
- potential parallelism where the compiler did not detect it,
- quantitative timing information, from which we derived speedups, parallelization overhead factors, and instrumentation costs.
- source code of the inspected loops and subroutines.

In addition to understanding the performance that is measurable on current systems, our tools can predict characteristics such as upper limits of parallelism and the scaling behavior of an application. All of these tools are based on the Polaris infrastructure, exploiting both its program analysis and manipulation capabilities. Profiles and detailed timing information are generated through Polaris-instrumented program runs. Loop structure and parallelization information result from Polaris' analysis passes. Potential parallelism is analyzed through a Polaris-based execution-driven simulator. And the program scaling behavior is determined from the compiler-derived loop structure and operation count information.

We found it particularly useful to keep a close relationship between the compiler development project and the effort providing tools for the compiler project. This led to synergy and insights about the desirable boundaries between compilers and other programming tools.

Tool improvements

Tools that are being used quickly accumulate a list of desirable extensions. The following areas seem particularly beneficial.

Our program instrumentation for timing measurements is done at the source level, which provides flexibility in selecting the instrumentation points of a program and in varying the code to be inserted. While the associated overhead was negligible in most cases, occasional perturbation had to be removed. We did this by estimating the overhead based on the known cost per instrumentation function and the measured number of function calls. This computation could be automated, generating a warning if the overhead is excessive.

Several extensions to the information that the URSA MINOR tool can display are planned. The ranges of variable values and array accesses are known internally to Polaris, gathered through its symbolic analysis package [BE95]. Making this information available to the user will be important for determining data access overlaps, from which one can derive parallelizable code sections and locality patterns. The same information helps determine the memory use (working set sizes), which is important for estimating memory requirements and cache performance. Furthermore, Polaris can output data dependences that inhibited parallelism, giving the user hints on where to focus attention in detecting further parallelism.

One long-term goal of our work is to establish a well-defined methodology of programming parallel machines and to provide supporting tools. For such a methodology, the information supplied by current program and performance analysis tools is only a starting point. In one effort, we have begun to define ways of transforming this raw information into terms that are closer to the user [Eig93], leading to suggested improvements for programs and architectures. The URSA MINOR environment is a means to implement such a framework.

Finally, in section 3.1 we have mentioned ongoing work on the MAX/P tool. Integrating the detection of dynamic and compiler-detected parallelism is one goal. It can lead to drastic reductions in the tool's execution time. Another on-going project is to automate the generation of the Do-loop and procedure parallelism profile graphs shown in Figure 6.

5 Conclusions

We have presented new parallel programming tools that facilitate the interactive use of optimizing compilers and that integrate compilers with performance analysis tools. The tools are implemented using the Polaris compiler infrastructure. We have described related projects, in which the tools helped to understand real applications, to identify new compilation techniques, and to evaluate programming languages.

The tools facilitate the development of parallel programs by providing to the user compiler-gathered program information and decision making facts of the optimizer. This information is combined with data collected from other programming tools, such as timing data gathered by performance analyzers and simulators that evaluate the program behavior under idealistic assumptions.

The presented tools are a first generation of a comprehensive environment that provides an integrated tool view. In our initial experiences we have found the following tool aspects particularly useful: providing program information on time-consuming sections, the loop/subroutine call structure, compiler-detected parallelism, potential parallelism where the compiler failed, and quantitative timing information. The tools have also become useful as a general program browsing utility.

We have identified several enhancements that serve as starting points to a second generation of this environment. Enhancements include the visualization of additional compiler-analyzed data, (such as array access information, symbolic variable ranges, data dependences,) lower-overhead instrumentation facilities, combined compile-time and runtime parallelism detection, and automated performance forecasting support. In addition, we will provide decision making support that can reason about the collected information and suggest improvements for applications, compilers, and architectures. In order to succeed in the design of such an integrated tool environment, one has to monitor and drive its progress with

real application studies. Such studies are ongoing in projects that are tightly related to our tool design effort.

References

[AE97] Brian Armstrong and Rudolf Eigenmann. Performance Forecasting: Characterization of Applications on Current and Future Architectures. *Technical Report ECE-HPCLab-97202*, Purdue University, School of Electrical and Computer, Engineering, High-Performance Computing Laboratory, February 1997.

[ASM89] Bill Appelbe, Kevin Smith, and Charles McDowell. Start/Pat: A Parallel-Programming Toolkit. *IEEE Software*, 6(4):29–38, July 1989.

[BDE+96] 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.

[BE95] William Blume and Rudolf Eigenmann. Symbolic Range Propagation. *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, April 1995.

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The ParaScope Editor: An Interactive Parallel Programming Tool. In *International Conference on Supercomputing*, pages 540–550, 1989.

[EH96] Rudolf Eigenmann and Siamak Hassanzadeh. Benchmarking with Real Industrial Applications: The SPEC High-Performance Group. *IEEE Computational Science & Engineering*, 3(1):18–23, Spring 1996.

[Eig93] Rudolf Eigenmann. Toward a Methodology of Optimizing Programs for High-Performance Computers. *Conference Proceedings, ICS'93, Tokyo, Japan*, pages 27–36, July 20-22, 1993.

[EM93] Rudolf Eigenmann and Patrick McClaughry. Practical Tools for Optimizing Parallel Programs. *Presented at the 1993 SCS Multiconference, Arlington, VA*, March 27 - April 1, 1993.

[KAI97] Kuck and Illinois. Associates Inc, Champaign. The KAP/Pro Toolset. http://www.kai.com/kpts/, 1997.

[KE97] Seon-Wook Kim and Rudolf Eigenmann. Max/P: Detecting Maximum Parallelism in a Fortran Program. *Technical Report ECE-HPCLab-97201*, Purdue University, School of Electrical and Computer, Engineering, High-Performance Computing Laboratory, 1997.

[MCC+95] 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 Tools. *IEEE Computer*, 28(11), November 1995.

[MH93] C. C. Mosher and S. Hassanzadeh. ARCO Seismic Processing Performance Evaluation Suite, User's Guide. *Technical report*, ARCO, Plano, TX., 1993.

[PE95] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 444–448, 1995.

[Pet93] Paul Marx Petersen. Evaluation of Programs and Parallelizing Compilers Using Dynamic Analysis Techniques. *PhD thesis*, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1993.

[PVAE97] Insung Park, Michael J. Voss, Brian Armstrong, and Rudolf Eigenmann. Interactive Compilation and Performance Analysis with Ursa Minor. In *Workshop of Languages and Compilers for Parallel Computing*, August 1997.

[Ree94] Daniel A. Reed. Experimental Performance Analysis of Parallel Systems: Techniques and Open Problems. In *Proc. of the 7th Int' Conf on Modeling Techniques and Tools for Computer Performance Evaluation*, pages 25–51, 1994.

[Vos97] Michael J. Voss. Portable Loop-Level Parallelism for Shared Memory Multiprocessor Architectures. *Master's thesis, School of Electrical and Computer Engineering, Purdue University*, 1997. (in preparation).