Performance Analysis Using SvPablo

Luiz De Rose *
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801, U.S.A.
(derose@cs.uiuc.edu)

Abstract

In this tutorial we present SvPablo, a graphical source code browser and performance visualizer that integrates the University of Illinois Pablo project's dynamic performance instrumentation software with HPF and C compilers and with the MIPS R10000 hardware performance mechanism. The following topics will be covered in the tutorial: an introduction to the SvPablo environment, including an overview of the Self-Defining Data Format, used to store the performance files; the instrumentation and visualization of HPF and C programs; the integration with the MIPS R10000 hardware performance counters; and finally an experiment using a real life application running on an SGI Origin 2000, that demonstrates the usefulness of SvPablo for tuning application programs.

Resumo

Esse tutorial descreve a interface gráfica SvPablo que foi desenvolvida na University of Illinois para instrumentação e visualização de desempenho de programas. SvPablo integra o software de instrumentação de desempenho Pablo com compiladores C e HPF, e com o mecanismo de instrumentação de hardware disponível nos microprocessadores MIPS R10000. Os seguintes tópicos vão ser abordados nesse tutorial: uma introdução a interface gráfica SvPablo, incluindo uma visão geral do Self-Defining Data Format, que é usado nos arquivos de desempenho; instrumentação e visualização de programas HPF e C; integração com o mecanismo de instrumentação de hardware do MIPS R10000; e finalmente um experimento usando uma aplicação real, rodando numa SGI Origin 2000, que demonstra a utilidade do SvPablo para o ajuste de aplicações científicas.

*This work was supported in part by Army contract DABT63-91-K-0004.

1 Introduction

Developing an application program that achieves high performance on a scalable parallel system requires a cycle of experimentation and refinement in which one first identifies the key program components responsible for the bulk of the program's execution time and then modifies the program in the hope of improving its performance. For this cycle to be effective not only must performance data be accurate, it must be directly tied to the source program and to the underlying architecture.

Traditional performance analysis tools capture dynamic performance data from the executing code (e.g., by instrumenting MPI communication primitives) and rely on the application software developer to map the resulting performance data to source code constructs. However, the use of high-level languages like HPF and sophisticated parallelizing compilers means that an application software developer's mental model of a program and the actual code that executes on a particular parallel system are quite different. Modern analysis tools must provide the requisite performance data and suggestions for performance improvements at the level of an abstract, high-level program. Thus, they must integrate dynamic performance data with information recorded by the compiler that describes the mapping from the high-level source to the resulting low-level, explicitly parallel code [1].

Furthermore, performance analysis becomes even harder due to the complexity of new parallel architectures, such as DSM architectures, which have multi-level memory hierarchies and exploit speculative execution through branch predictions. Therefore, it is extremely important the integration of performance analysis tools with hardware performance mechanisms, to provide meaningful performance feedback to a programmer, so the behavior of the application program can be understood and its performance be improved.

In this tutorial we will present SvPablo, a graphical user interface tool for instrumenting source code and browsing runtime performance data. SvPablo, derived from the phrase *"Source view Pablo"*, integrates the University of Illinois Pablo project's dynamic performance instrumentation software with PGI HPF (pghpf) [5], the Portland Group's commercial HPF compiler, and the MIPS R10000 [4] hardware performance counters [3, 6]. In addition, using the same interface, SvPablo allows the interactive instrumentation of C programs, with ongoing work for the interactive instrumentation of Fortran 77 and Fortran 90 programs.

The rest of this document briefly describe the topics that will be covered in the tutorial. An overview of SvPablo and the Self-Defining Data Format (SDDF) are presented in Section 2. The instrumentation and visualization of HPF and C programs are introduced in Sections 3 and 4 respectively. Section 5 addresses the hardware performance integration. Finally, Section 6 concludes this tutorial with an example using a real application on an SGI Origin 2000, demonstrating the usefulness of SvPablo for tuning application programs.

2 Overview

SvPablo was designed to provide performance data capture, analysis, and presentation for applications executing on a variety of sequential and parallel platforms and written in a wide range of languages. The current release supports ANSI C programs and HPF programs. In addition, work is in progress to support Fortran D95, Fortran 77, and Fortran 90.

Source code instrumentation using SvPablo can be interactive or automatic, depending on the language. C, Fortran 77, and Fortran 90 programs are interactively instrumented using the SvPablo tool, while HPF and Fortran D programs are automatically instrumented by the compiler. Performance browsing is accomplished by correlating the performance data gathered at runtime with the source code, performing statistical analyses, and creating a *performance file*. This performance file, stored in the Pablo Self-Defining Data Format (SDDF) [2], is used as input to the SvPablo tool which presents the performance information as it relates to the original source code.

The Pablo SDDF is a performance data description language that specifies both data record structures and data record instances. This data meta-format provides the following requirements, dictated by the nature of performance data:

- Compactness: performance data files are often quite large, so compactness is a concern.
- Portability: performance data may be collected on one machine and analyzed on another. Consequently, SDDF is able to accommodate potential differences in byte order, word length, and floating point representation.
- Generality: depending on the system component being studied (application software, system software, or hardware) as well as on the underlying architecture-specific characteristics (shared or distributed memory), the "interesting" performance data will vary. Therefore, a diverse set of event types is supported by SDDF.
- Extensibility: the set of performance event types will grow as the performance analysis environment is extended, so SDDF supports the addition of new event types.

Compactness and portability are obtained with the two possible representations for SDDF files, a binary representation that is more compact and an ASCII representation that is completely portable and human-readable. Generality and extensibility are provided by the meta-format approach which makes it easy to have a diverse set of event types and simple to add new event types as they are needed.

The SvPablo tool's interaction model, shown in Figure 1, is based on the notion of a project. Associated with a project are a set of application source files and one or more performance contexts. Each performance context may include an instrumentation specification and a corresponding set of performance data. An instrumentation specification is sometimes referred to as a *configuration*, and contains the source code points where performance measurements probes are inserted. The performance data is generated when the instrumented

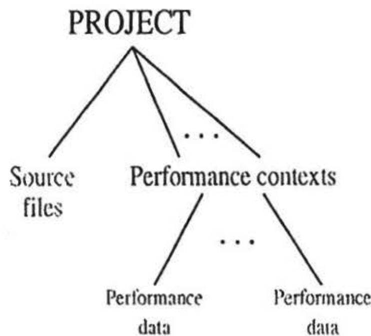

Figure 1: SvPablo model

code is run and the resulting summary files are merged and analyzed statistically. Because these are statistics, rather than detailed event traces, SvPablo can measure the performance of programs that execute for hours or days on hundreds of processors.

3 Instrumentation and Visualization of HPF programs

SvPablo works with the commercial HPF compiler from the Portland Group Incorporated (PGI). As illustrated in Figure 2, the HPF instrumentation/visualization process starts with the compilation of the HPF program. The PGI HPF compiler automatically inserts calls to the SvPablo instrumentation library at the beginning and end of every procedure, and for each executable line in the program. Therefore, every line and procedure executed by the program contributes to the runtime performance information.

After compiling and linking the HPF program, the user runs the instrumented executable code which generates a set of *per-process summary files* at the end of the run. One summary file is generated for each process activated by the program, and each file contains trace information that was summarized during runtime for the corresponding active process.

The next step in the instrumentation/visualization process is the merge and statistical analysis of the trace data. After this step, a single performance file is generated. This SDDF file, referred to here as the *HPF performance file*, contains dynamic performance statistics for all routines and lines executed in the program. The HPF performance file is used as input to the SvPablo tool which presents the performance information as it relates to the original source code.

Different metrics are collected for procedure and line statistics, as presented in Tables 1 and 2. In addition, as described in Section 5, a set of hardware performance events can be captured when running on a MIPS R10000 microprocessor.

Figure 3 shows SvPablo's main window, which displays the performance data for an HPF

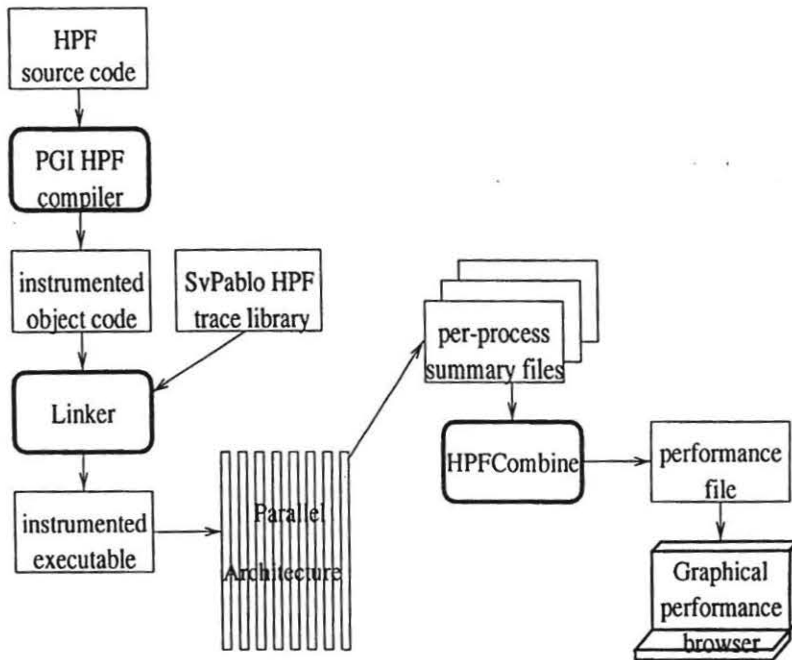

Figure 2: Overall view of the HPF instrumentation

program that simulates the dynamics of the shallow-water equations. The window is divided in six panes, containing the following information:

- `Project Description`: displays a textual summary of the current project.
- `Source Files`: lists the source files for the current project.
- `Performance Contexts`: lists the performance contexts for the current project.
- `Routines in Source File`: lists the routines defined in a selected source file[1].
- `Routines in Performance Data`: lists the routines appearing in a selected performance context, together with graphical presentation of per-routine performance data.
- `Source File`: displays the name of the current source file, the source code, and per-event performance data presented graphically on the individual source code lines.

[1] This pane is used only for interactive instrumentation.

Metric name	Description
Count	number of procedure activations
Exclusive Duration	time in seconds spent in the procedure, excluding calls to other procedures
Inclusive Duration	total time in seconds spent in the procedure
Send Msg Duration	time in seconds spent sending messages
Receive Msg Duration	time in seconds spent receiving messages

Table 1: Procedure statistics metrics

Metric name	Description
Count	number of occurrences of the line
Duration	total time in seconds spent executing the line
Exclusive Duration	time in seconds spent in the line excluding procedure calls
Message Send Duration	time in seconds spent sending messages
Message Send Count	number of messages sent
Message Send Size	total number of bytes in messages sent
Message Receive Duration	time in seconds spent receiving messages
Message Receive Count	number of messages received
Message Receive Size	total number of bytes in messages received

Table 2: Line statistics metrics

The SvPablo browser provides a hierarchy of color-coded performance displays, including a high-level routine profile and source code scrollboxes. The color columns graphically summarize the number of calls and the cumulative time for the routines, and the metrics for the lines executed in the program (one column for each line statistics metric). Clicking the mouse pointer in the color column area next to the routine name in the pane `Routines in Performance Data` or next to the line in the `Source File` pane results in one of the following displays, depending on the button used:

- **Left button:** displays the value associated with the color box under the pointer
- **Middle button:** displays a dialog prompting for different distributions of values within the color range (linear, quadratic, or exponential).
- **Right button:** displays a legend describing each column and the colors associated with the minimum and maximum values for the individual columns

In addition, pop-up dialogs showing other statistics and detailed information about a particular routine or a particular line, including per-processor metrics, can be obtained by clicking the mouse on the routine name or the line.

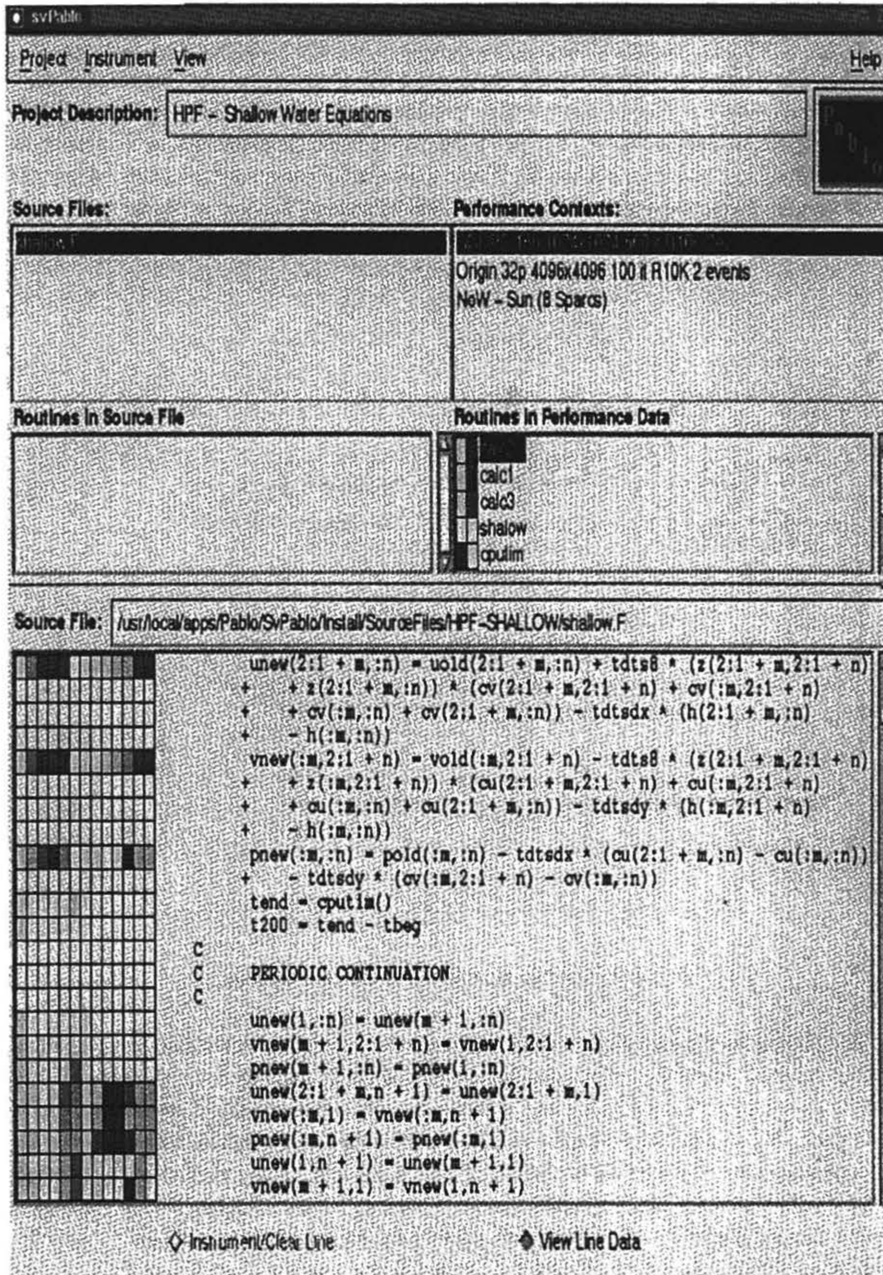

Figure 3: Performance data from the shallow-water equations program

4 Instrumentation and Visualization of C programs

SvPablo allows the interactive instrumentation of ANSI C programs. As illustrated in Figure 4, the C instrumentation/visualization process starts with the creation of a project, followed by the instrumentation of selected *constructs* (sometimes referred to as *events*) in the C source files, and then generation of an instrumented executable program. After the interactive instrumentation of the C program, the remaining process is similar to the one described for HPF.

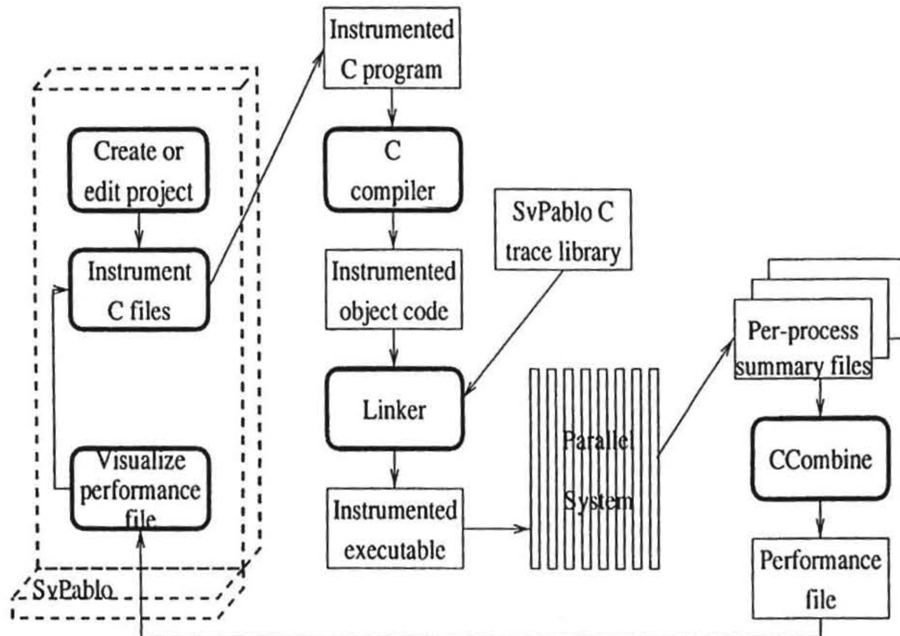

Figure 4: Overall view of the C instrumentation

For each C source file selected to be instrumented, SvPablo parses the file to identify instrumentable constructs[2], displays all functions defined and called by the code in the `Routines in Source File` pane, and displays the file contents in the scrollable `Source Code` pane. Within the `Source Code` pane, lines containing instrumentable constructs (events) are marked with a ">" symbol to the left of the source line. These events can be selected to be instrumented via the `Instrument` menu, shown in Figure 5, or by clicking the mouse on the corresponding line. A "⇝" symbol appears on the line to the left of the ">" symbol indicating that it has been marked for instrumentation. Notice that Figure 5 shows the

[2]In the current implementation, *function calls* and *outer loops* are the instrumentable constructs.

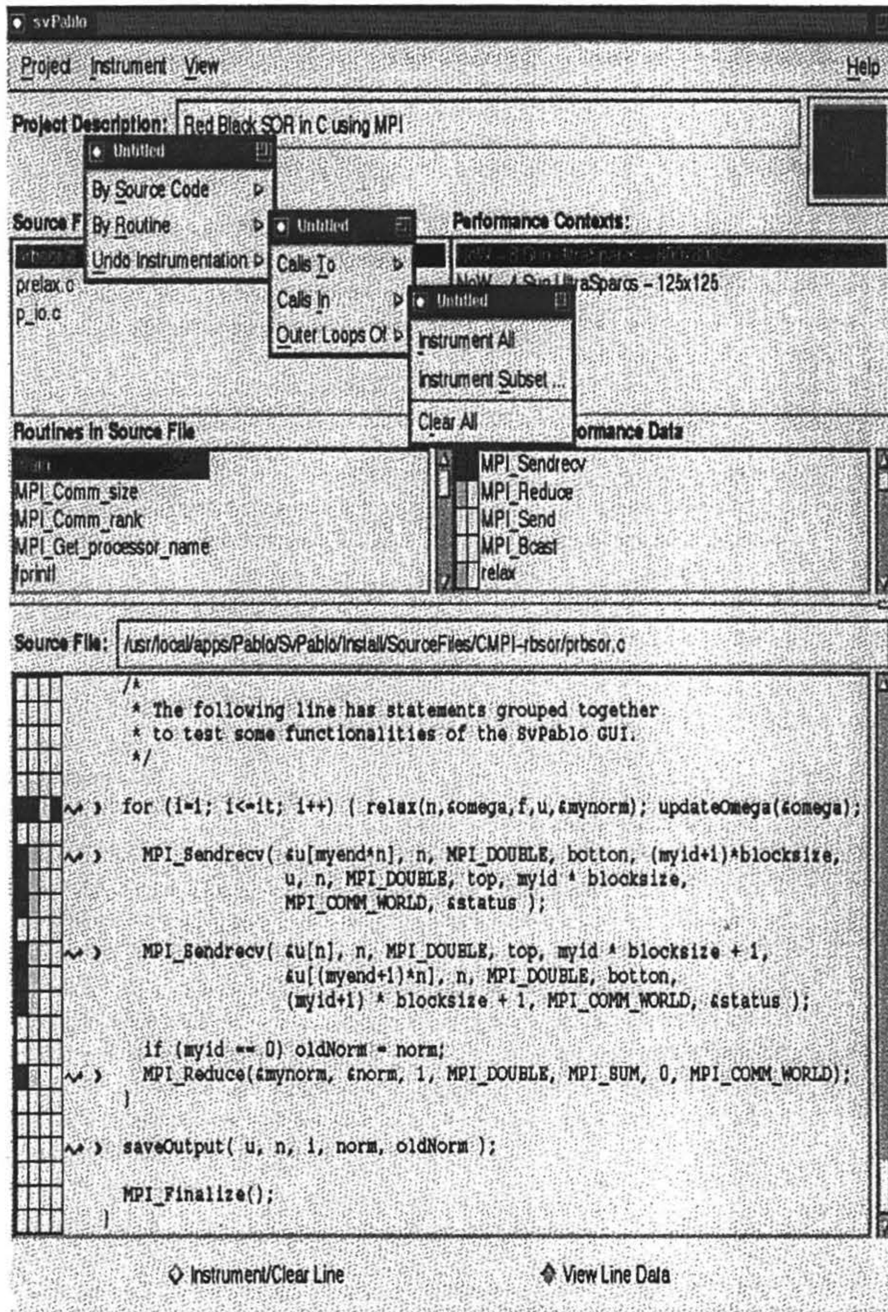

Figure 5: Instrumentable constructs and performance data in the file prbsor.c

SvPablo main window for a project after the instrumented program was compiled and run, thus it also displays the program's performance data.

5 Hardware Performance Integration

Hardware performance monitoring is integrated into SvPablo with the use of the MIPS R10000 hardware performance counters. The MIPS R10000 microprocessor provides detailed information on the behavior of the chip through its hardware performance facility. This performance facility provides two hardware performance counters, each one able to track up to 16 different events, as presented in Table 3.

Counter zero			Counter one		
No.		event	No.		event
0		Cycles	16		Cycles
1		Instructions issued	17		Instructions graduated
2		Load/prefetch/sync issued	18		Load/prefetch/sync graduated
3		Stores issued	19		Stores graduated
4		Store conditional issued	20		Store conditional graduated
5		Failed store conditional	21	*	Floating-point instructions (grad)
6		Branches decoded	22		Write back from data cache to secondary cache
7		Write back from secondary cache to System interface	23		TLB refill exceptions
8		Single-bit ECC errors on secondary cache data	24		Branches mispredicted
9	*	Instruction cache misses	25	*	Data cache misses
10	*	Secondary cache misses (inst.)	26	*	Secondary cache misses (data)
11		Secondary cache way mispredicted (instruction)	27		Secondary cache way mispredicted (data)
12		External intervention requests	28		External intervention hits
13		External invalidation requests	29		External invalidation hits
14		Virtual coherency	30		Upgrade requests on clean secondary cache lines
15	*	Instructions graduated	31		Upgrade requests on shared secondary cache lines

Table 3: MIPS R10000 Counters (* denotes default events captured by *SvPablo*)

To enable capture of more than two events during program execution, the operating system kernel maintains a set of 32 virtual counters, multiplexing the physical counters

across these. When this multiplexing approach is used, the kernel switches events at every clock with each event being counted once every n clock cycles, where n is the number of events selected for each counter. This multiplexing sacrifices accuracy, that is inversely proportional to the number of events selected from each counter, but increases coverage, since it is possible for a program to access the information from all events.

Through the SvPablo interface, the user can select the hardware events to be instrumented by providing at runtime an ASCII file containing the events of interest. If this file is not available during runtime, the SvPablo data capture library uses a default set of events, marked with an asterisk in Table 3. The SvPablo data capture library configures the R10000 processor to query the virtual counters, recording this data with extant application measurements. In addition to presenting the data obtained from the counters, SvPablo uses data from selected events to synthesize new metrics, such as MFLOPS per processor and branches mispredicted percentage, for each executed line.

6 Application Tuning Example

In this section we demonstrate the usefulness of SvPablo for tuning application programs. As an example, we use a numerical model to simulate cloud and density current dynamics. This model is a three-dimensional, non-hydrostatic, finite difference, convective cloud model which utilizes a quasi-compressible version of the Navier-Stokes equations. The program was originally written in CM Fortran for the CM5 and was translated to HPF to run on the SGI Origin 2000. The current version has approximately 9000 lines. In this example, we executed two versions of the program (baseline and modified) on an Origin 2000, using 8 processors.

Figure 6 shows SvPablo's main window after the generation of the performance data for the baseline program. We started tuning our application by selecting the routine `s_mix`, which had the largest cumulative time (197.20 seconds, as shown in the corresponding metric display). The metrics corresponding to each column in the pane `Source File` and the range of values for each metric (for the lines of the file `s_mix.hpf`) is presented in Figure 7(A).

Scrolling down the source code, we observe that most of the execution time for the routine `s_mix` was spent computing the array `fs`, when performing the operations `X - Mixing` and `Y - Mixing`. This is easily identified due to the color-coded performance displays. In this example, we observe that most of the execution time for each computation of `fs` is due to communication (represented by the last 2 columns to the right in the `Source Code` pane). As shown by the specific metric displays, out of the 97.68 seconds for the total execution time for the highlighted line, 90.66 seconds was due to message receives and 7.06 seconds was due to message sends. We also observe that this statement had in excess of 5.6 million data cache misses and each processor achieved at most 0.05 MFLOPS.

An analysis of the loop indicates that the reason for this poor performance could be due to the circular shift of the array `flx`. This array is computed in the previous statement of the

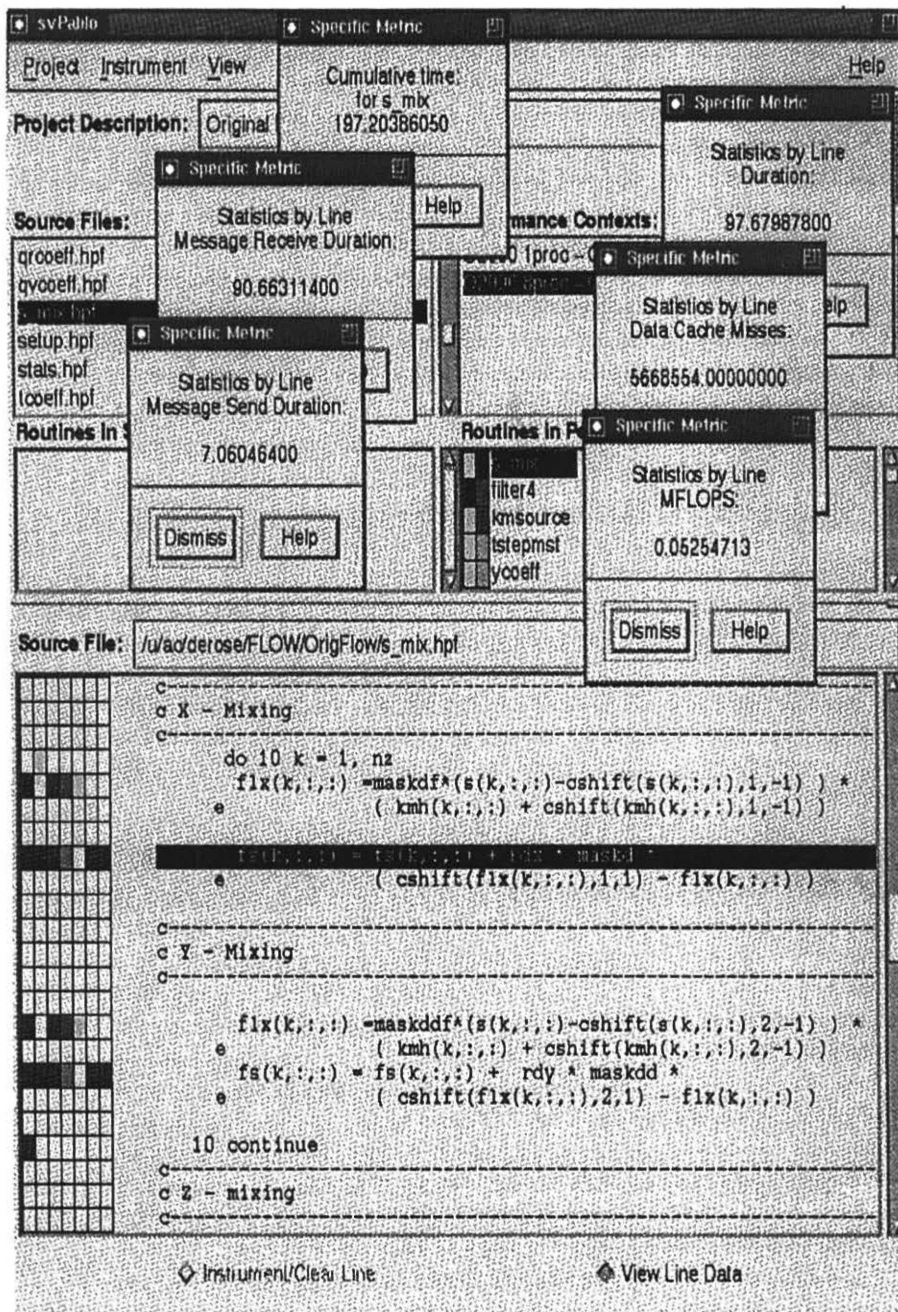

Figure 6: Performance data from the baseline program

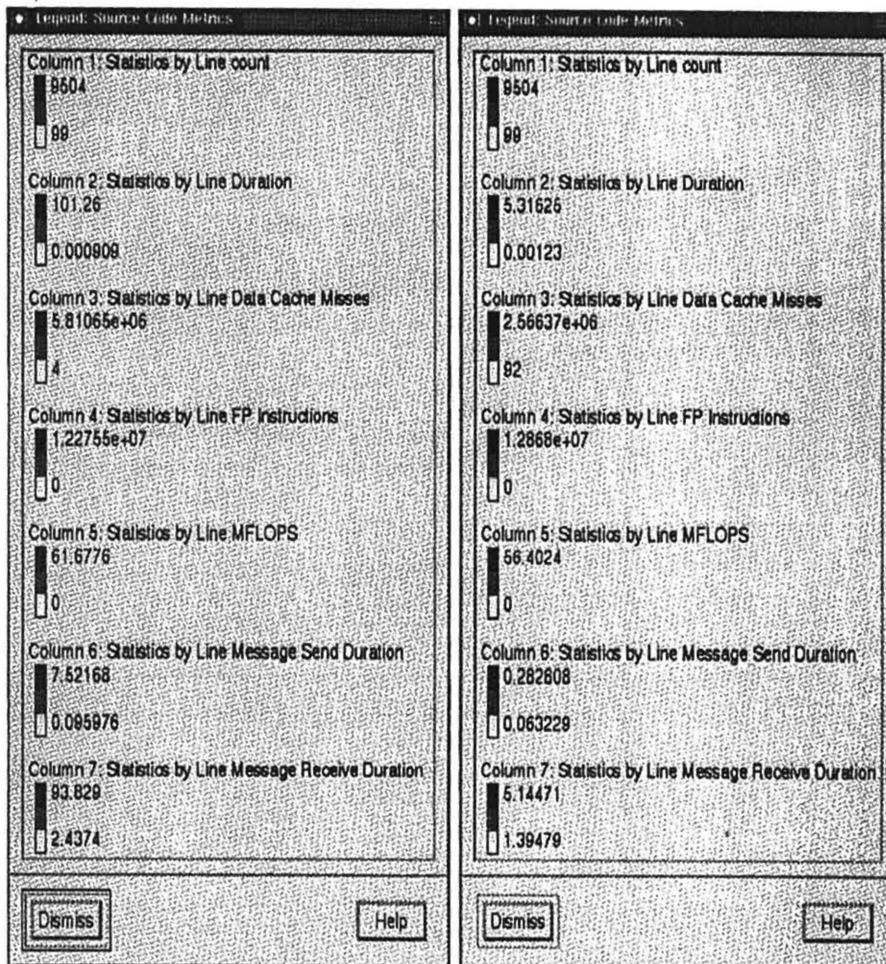

Figure 7: Performance metrics and range of values for the baseline program (A) and for the modified program (B)

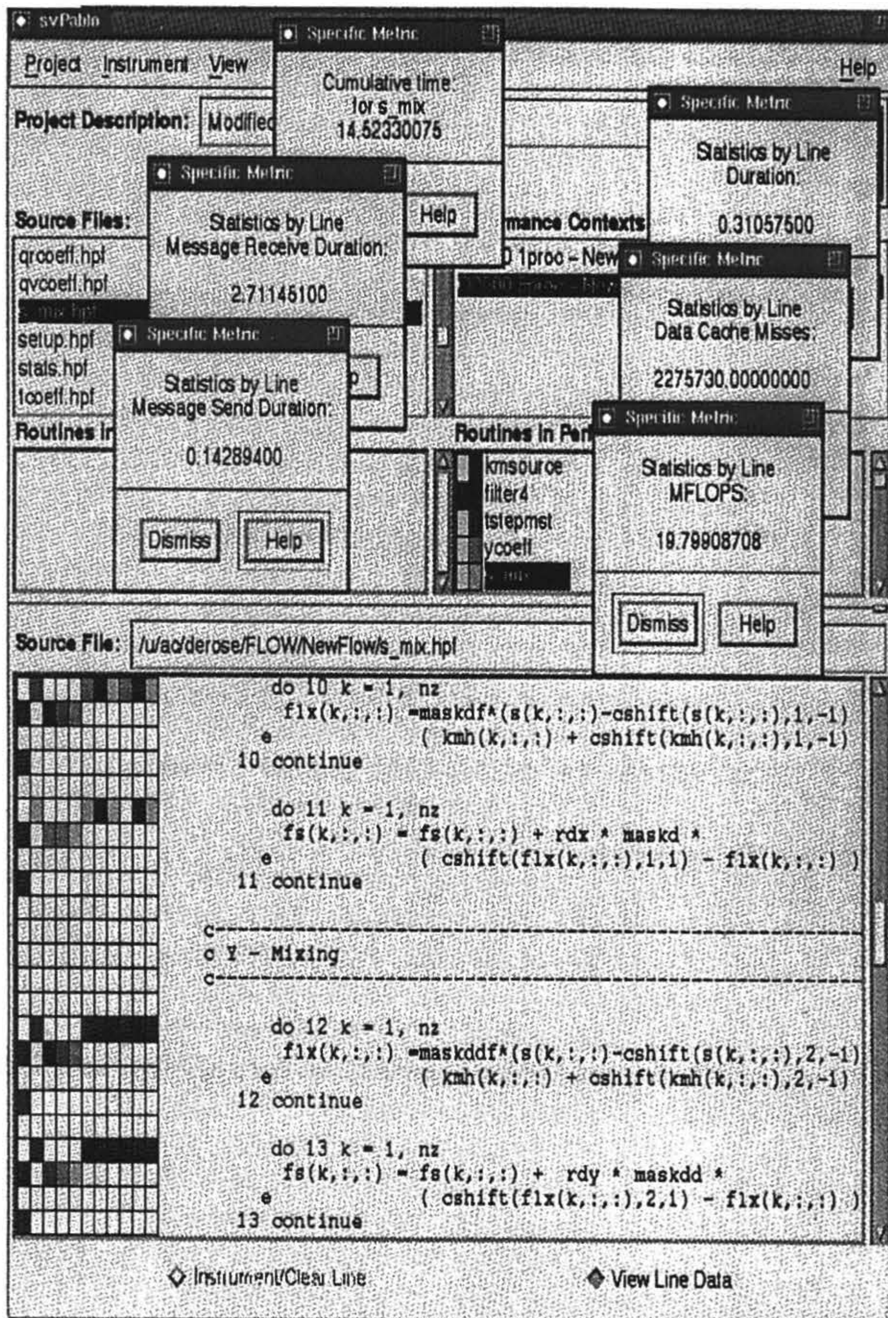

Figure 8: Performance data from the modified program

loop, so the processors have to wait during each iteration of the loop for the circular shift to occur. To improve the performance of this routine, we split the original loop into four loops, one for each statement inside of the original loop. By splitting the loop, we expected that due to prefetching, the data migration would start as soon as each iteration of the previous loop was completed.

Figure 8 displays the performance data for the modified version of the program, with the metrics and range of values for each column being presented in Figure 7(B). We observe that as expected, after splitting the original loop, the communications occurred between loops, reducing considerably the duration of message sends and message receives (to 0.14 and 2.71 respectively for the same highlighted statement from Figure 6). In addition, we observe a reduction in the number of data cache misses, with the total time for the statement being only 0.31 seconds and performance close to 20 MFLOPS per processor.

Due to the loop splitting and the moving of the communications, the total time of the loop control was increased by a factor of three (from approximately 6 seconds accounted to execute the single loop control statement in the baseline program to roughly 18 seconds to execute all four loop control statements in the modified program). However, even with this increase in execution time for the loop control, the total execution time for the routine `s_mix` dropped by one order of magnitude.

References

[1] ADVE, V., MELLOR-CRUMMEY, J., WANG, J.-C., AND REED, D. Integrating Compilation and Performance Analysis for Data-Parallel Programs. *Proceedings of Supercomputing'95* (November 1995).

[2] AYDT, R. The Pablo Self-Defining Data Format. Tech. rep., Department of Computer Science at the University of Illinois at Urbana-Champaign, April 1994.

[3] MIPS TECHNOLOGIES INC. *Definition of MIPS R10000 Performance Counters*, 1996. http://www.sgi.com/MIPS/products/r10k/Perf_Cnt/R10K_PF_Count.doc.html.

[4] MIPS TECHNOLOGIES INC. *R10000 Microprocessor User's Manual*, version 2.0 ed., October 1996. http://www.sgi.com/MIPS/products/r10k/UMan_V2.0/R10K_UM.cv.html.

[5] THE PORTLAND GROUP, INC. *PGHPF User's Guide*, 1994.

[6] ZAGHA, M., LARSON, B., TURNER, S., AND ITZKOWITZ, M. Performance Analysis Using the MIPS R10000 Performance Counters. *Proceedings of Supercomputing'96* (November 1996).