SvPablo: A Multi-Language Architecture-Independent Performance Analysis System

Luiz De Rose and Daniel A. Reed [*]
{derose,reed}@cs.uiuc.edu

Department of Computer Science
University of Illinois
Urbana, Illinois 61801 USA

Abstract

In this paper we present the design of SvPablo, a language independent performance analysis and visualization system that can be easily extended to new contexts with minimal changes to the software infrastructure. At present, SvPablo supports analysis of applications written in C, Fortran 77, Fortran 90, and HPF on a variety of sequential and parallel systems. In addition to capturing application data via software instrumentation, SvPablo also exploits hardware performance counters to capture the interaction of software and hardware. Both hardware and software performance data are summarized during program execution, enabling measurement of programs that execute for hours or days on hundreds of processors. This performance data is stored in a format designed to be language transparent and portable. We demonstrate the usefulness of SvPablo for tuning application programs with a case study running on an SGI Origin 2000.

Keywords: Performance analysis and visualization; Parallel and distributed systems; Software instrumentation; Hardware monitoring; Run-time summarization; SDDF Meta-format.

Technical area: Software Tools.

[*]This work was supported in part by the Defense Advanced Research Projects Agency under DARPA contracts DABT63-94-C0049, F30602-96-C-0161, DABT63-96-C-0027, and N66001-97-C-8532; by the National Science Foundation under grants NSF CDA 94-01124 and ASC 97-20202; and by the Department of Energy under contracts DOE B-341494, W-7405-ENG-48, and 1-B-333164.

1 Introduction

Developing applications that achieve high performance on current parallel and distributed systems requires multiple iterations of performance analysis and refinement. In each cycle, analysts first identify the key program components responsible for the bulk of the program's execution time and then modify the program to improve its performance.

For this methodology to be effective, not only must performance data be accurate, it must be directly tied to the source program and to the underlying architecture. Without such ties, application developers must deduce the effects of compiler transformations on source code and the interactions of compiled code with architectural features and runtime systems. This is especially difficult when data parallel source code is aggressively transformed to message passing or shared memory compiled code [1].

The complexity of new parallel architectures further exacerbates performance analysis problems — new distributed shared memory (DSM) systems have multilevel memory hierarchies managed by distributed cache coherence protocols, all accessed by superscalar processors that speculatively execute instructions. Understanding the execution behavior of application code in such environments requires access to hardware performance counters and careful mapping of the resulting data to source code constructs.

Correlating data parallel source code with dynamic performance data from both software and hardware measurements, while still providing a portable, intuitive, and easily used interface, is a challenging task [9]. However, without such tools, the use of high-performance parallel systems will remain limited to a small cadre of application developers willing to master the arcane details of processor architecture, system software, and compilation systems.

To provide a language and architecture independent mechanism for performance analysis, we developed SvPablo (source view Pablo), a graphical environment for instrumenting application source code and browsing dynamic performance data. SvPablo supports performance data capture, analysis, and presentation for applications written in a variety of languages and executing on both sequential and parallel systems. In addition, SvPablo exploits hardware support of performance counters.

During the execution of instrumented code, the SvPablo library captures data and computes performance metrics on the execution dynamics of each instrumented construct on each processor. Because only statistics, rather than detailed event traces, are maintained, the SvPablo library can capture the execution behavior of codes that execute for hours or days on hundreds of processors.

Following execution, performance data from each processor is integrated, additional statistics are computed, and the resulting metrics are correlated with application source code, creating a *performance file* that is represented via the Pablo self-describing data format (SDDF) [11, 3]. This performance file is the specification used by the SvPablo browser to display application source code and correlated performance metrics.

Use of the Pablo SDDF meta-format has enabled us to develop a user interface that is both portable and language independent. Moreover, because all performance metrics are defined in SDDF, the interface is also performance metric independent, allowing us to introduce new metrics or support new languages without change to the user interface code.

The remainder of this paper is organized as follows. In §2, we begin by describing the SvPablo interactive and automatic source code instrumentation system, followed in §3 by a discussion of support for hardware performance counters, and in §4, by a description of the SvPablo performance visualization interface. Following this, we describe use of SDDF to facilitate language and metric independence in §5. Building on this base, §6 demonstrates application of SvPablo to several large scientific applications. Finally, §7–§8 discuss related work and summarize our conclusions.

2 Performance Instrumentation

Interactive instrumentation provides detailed control, allowing users to specify precise points at which data should be captured, albeit at the possible expense of excessive perturbation and inhibition of compiler optimizations. In contrast, automatic instrumentation relies on the compiler or runtime system to insert measurement probes in compiler-synthesized code. Although this reduces the probability of excessive instrumentation perturbation, it sacrifices user control over instrumentation points. Because each is appropriate in different circumstances, SvPablo supports both interactive and automatic instrumentation.

The current version of SvPablo supports interactive instrumentation of C, Fortran 77, and Fortran 90 and automatic instrumentation of data parallel High Performance Fortran (HPF). As described below, these choices were driven by experience with earlier generations of instrumentation systems [11, 1] and analysis of common compiler transformations on current parallel systems.

2.1 Automatic Instrumentation

To support analysis of codes written in data parallel HPF, SvPablo is integrated with the commercial HPF compiler [10] from the Portland Group (PGI). The PGI HPF compiler emits message passing code with embedded calls to the SvPablo data capture library. This instrumentation captures data for each executable line in the original HPF source code. In addition, the compiler synthesizes instrumentation for every procedure entry and exit and for each message exchange among tasks.

Using these instrumentation points, the SvPablo data capture library automatically computes performance metrics for all procedures and source code lines. In addition, as described in §3, this library can also capture hardware performance events [7, 14] via the MIPS R10000 [8] performance counters[1].

The desire to capture performance data for data parallel code via compiler-synthesized instrumentation is based on the results of our collaboration with the Rice Fortran D project [1, 2]. The D system, a precursor to current HPF compilers, supports a large set of aggressive optimizations, including procedure inlining, loop distribution, software pipelining, and global code motion. Collectively, these optimizations result in executable code that differs markedly from the original source.

Instrumenting the data parallel source code can potentially inhibit any or all of these optimizations, dramatically reducing performance and, equally importantly, resulting in performance measurements that are not typical of normal execution. Hence, SvPablo relies on

[1]The hardware instrumentation interface is readily extensible to other microprocessors.

the data parallel HPF compiler to emit instrumented code.

2.2 Interactive Instrumentation

In contrast to the high-level optimizations often supported by data parallel compilers, most compilers for sequential languages focus primarily on local optimizations (e.g., register reuse, common subexpression elimination, and strength reduction). Hence, SvPablo supports the interactive instrumentation of ANSI C, Fortran 77, and Fortran 90 codes via instrumenting parsers[2], since source code instrumentation in these languages tends to have far less pernicious effects.

For each source file the user wishes to instrument, SvPablo parses the file and marks all *instrumentable constructs*. In a compromise between instrumentation detail and perturbation, SvPablo restricts these constructs to outer loops and procedure calls. This restriction draws on our earlier experience with the Pablo interactive instrumentation interface [11]. We observed that naive users tended to instrument everything, resulting in prodigious volumes of performance data, high perturbations, and little insight.

After the instrumentation is complete, the SvPablo parser generates a copy of the source code file with calls to the data capture library. During execution, the SvPablo library accumulates execution counts and durations for all instrumented constructs. As we shall see in §4, performance metrics are shown beside each instrumentable construct. This allows users to instrument an application, examine the correlation of performance metrics and

[2]Programs written in these languages can also be semi-automatically instrumented via command line options.

source code, and re-instrument the application using the knowledge obtained.

3 Hardware Performance Integration

Although software instrumentation can capture the interaction of compiler-synthesized code with runtime libraries and system software, understanding the effects of superscalar instruction execution and caching requires concurrent capture of both software and hardware performance metrics. This is especially true when such processors are combined to form parallel systems with hardware-managed memory hierarchies that interact with compiler-synthesized application data movement. Fortunately, new microprocessors commonly provide a set of performance registers for low overhead access to hardware performance data. For example, the MIPS R10000 includes registers that count cycles, level one and level two cache misses, floating point instructions, and branch mispredictions. Similar counters exist on the SUN UltraSPARC, Intel Pentium Pro, and IBM Power2.

The SvPablo instrumentation library includes a standard interface for augmenting software performance data with hardware metrics retrieved from special-purpose processor registers. We have exploited this interface to capture hardware performance data on the MIPS R10000 and SGI Origin 2000.

The MIPS R10000 microprocessor includes two hardware performance counters, each able to track one of 16 different events. To capture more than two events during program execution the operating system maintains a set of 32 virtual counters, multiplexing the physical counters across these. When N events are selected, the SGI operating system switches events every cycle, counting each event every N clock cycles. Although this multiplexing sacrifices accuracy, it increases coverage, allowing a single program execution to acquire data for all the desired hardware events. Within SvPablo, a user can select any desired set of hardware events by specifying an ASCII file that contains the virtual counters of interest. The interface with the hardware counters is done through the SvPablo data capture library, so the user does not need to re-compile the program to use a different set of hardware events.

During program execution, the SvPablo data capture library queries the virtual counters and records the data with extant application measurements. In addition to presenting the raw counter data, the SvPablo library also computes derived metrics for each source code line (e.g., MFLOPS and branch misprediction percentages).

After program execution, the SvPablo data capture library records its statistical analyses in a set of *summary files*, one for each executing process. A post-mortem utility program then merges the summary files, computing new global statistics and correlating metrics across processors. The resulting summary file is then input to the analysis graphical interface.

Taken together, the application and hardware performance measurements provide a rich set of metrics for program analysis. Moreover, the SvPablo interface allows users to identify high-level bottlenecks (e.g., procedures), then explore increasingly levels of detail (e.g., identifying the specific cause of poor performance at a source code line executed on one of many processors).

4 Performance Visualization

As noted at the outset, one of the design goals for SvPablo was to create an intuitive, cross-architecture, language independent performance analysis interface. Realizing such a design would allow users and performance analysts to learn a single set of software navigation skills and then apply those skills to application codes written in a variety of languages and executing on a diverse set of sequential and parallel architectures.

Hence, the SvPablo implementation relies on a single interface for performance instrumentation and visualization. If the program was interactively instrumented, the user can refine the performance analysis by re-instrumenting the source code while visualizing performance data from earlier executions. Regardless of the instrumentation mode, one can access and load performance data from multiple prior executions, including different numbers of processors and hardware platforms. This allows one to compare executions to understand hardware and software interactions.

As an example, Figure 1 shows the SvPablo interface, together with code and performance data from an HPF program. As the figure suggests, the SvPablo interface supports a hierarchy of performance displays, ranging from color-coded routine profiles to detailed data on the behavior of a source code line on a single processor.

In the figure, the leftmost scrollbox shows the set of files comprising the HPF program, with all previously measured executions of this code shown in the scrollbox to the right. Here, the user has loaded a performance data context (i.e., a measured execution) for an eight processor SGI Origin 2000. After selecting a performance context, the list of procedures in the application code, together with two color coded metrics, is shown below the performance contexts scrollbox in the area labeled *Routines in Performance Data*. The two colored columns summarize, over all processes, the average number of calls and average cumulative time for the routines.

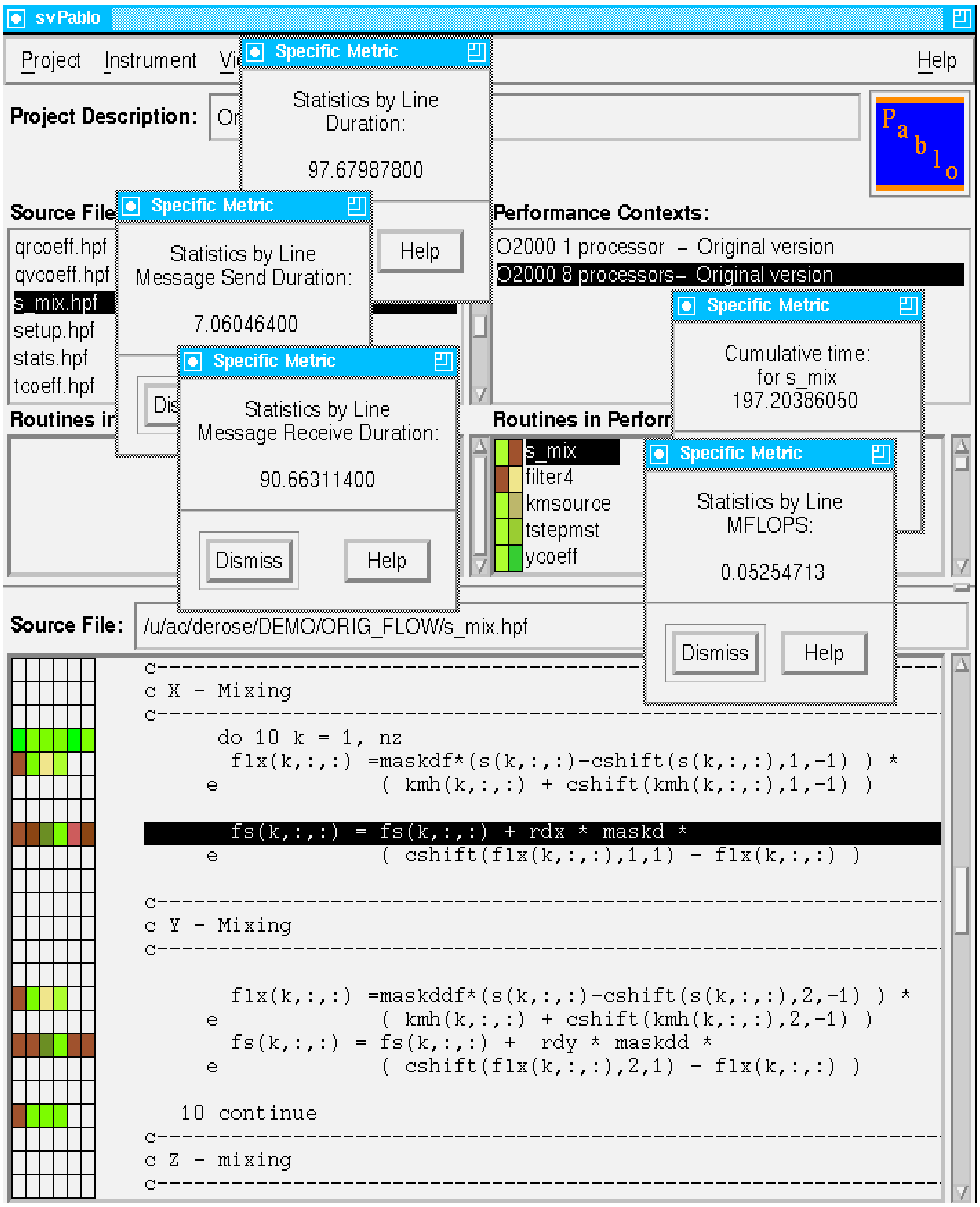
Clicking on a routine name loads the associated source code in the bottom pane of Figure 1, together with color-coded metrics beside each source code line. By default, the SvPablo interface displays one column for each metric. However, the user can select only a subset of the metrics to appear in the color coded columns. Clicking the mouse on a colored box, either in the routine list or beside a source code line, creates a dialog box displaying the maximum value associated with the selected metric.

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

5 Language and Architecture Transparency

Developing a user interface that separates performance data presentation from language and architecture idiosyncrasies requires a flexible specification mechanism for both instrumentation points and performance metrics. Only with this separation can one readily add new metrics and support new languages, compilers, and architectures without requiring extensive modifications to the user interface code.

In SvPablo, the Pablo self-defining data format (SDDF) [11, 3] provides this separation.



```
qrcoeff.hpf
qvcoeff.hpf
s_mix.hpf
setup.hpf
stats.hpf
tcoeff.hpf
```

```
O2000 1 processor  – Original version
O2000 8 processors– Original version
```

```
s_mix
filter4
kmsource
tstepmst
ycoeff
```

```
Statistics by Line
Duration:

97.67987800
```

```
Statistics by Line
Message Send Duration:

7.06046400
```

```
Statistics by Line
Message Receive Duration:

90.66311400
```

```
Cumulative time:
for s_mix
197.20386050
```

```
Statistics by Line
MFLOPS:

0.05254713
```

Source File: /u/ac/derose/DEMO/ORIG_FLOW/s_mix.hpf

```
c-----------------------------------------------------------------------
c X - Mixing
c-----------------------------------------------------------------------
      do 10 k = 1, nz
       flx(k,:,:) =maskdf*(s(k,:,:)-cshift(s(k,:,:),1,-1) ) *
     e              ( kmh(k,:,:) + cshift(kmh(k,:,:),1,-1) )

       fs(k,:,:) = fs(k,:,:) + rdx * maskd *
     e              ( cshift(flx(k,:,:),1,1) - flx(k,:,:) )

c-----------------------------------------------------------------------
c Y - Mixing
c-----------------------------------------------------------------------

       flx(k,:,:) =maskddf*(s(k,:,:)-cshift(s(k,:,:),2,-1) ) *
     e              ( kmh(k,:,:) + cshift(kmh(k,:,:),2,-1) )
       fs(k,:,:) = fs(k,:,:) +  rdy * maskdd *
     e              ( cshift(flx(k,:,:),2,1) - flx(k,:,:) )

   10 continue
c-----------------------------------------------------------------------
c Z - mixing
c-----------------------------------------------------------------------
```

Figure 1: Baseline Performance Data (MSTFLOW HPF Code)

SDDF defines data streams that consist of a group of record descriptors and record instances. Much as structure declarations in the C programming language specify templates for storage allocation, SDDF descriptors define the structure for record instances. The data following the descriptors consists of a stream of descriptor tag and data record pairs. Each tag identifies the descriptor that defines the juxtaposed data. By separating the structure of data from its semantics, the Pablo SDDF library permits the construction of tools that can extract and process SDDF records and record fields with minimal knowledge of the data's deeper semantics.

The SDDF meta-format provides the generality and extensibility necessary to represent a diverse set of performance metrics and measurement points. Because this data may vary across languages or even across executions of the same program (e.g., when instrumentation points are changed interactively or when a different set of hardware metrics is captured), the performance data files that define execution contexts rely on SDDF specifications that include both mandatory and optional data fields.

As an example, consider the C code fragment

```
for ( i = loopStart( p1 ); i < loopEnd( p2 ); i++ ) { A[i] = f( i ); }
```

containing four instrumentable constructs: the `for` loop and three function calls (`loopStart`, `loopEnd`, and `f`). Multiple performance metrics (e.g., elapsed time and hardware metrics) may be associated with each of these instrumentable constructs. Similarly, automatic instrumentation of an HPF statement may result in multiple communication time and data volume metrics.

To isolate such language differences from the user interface, the performance metrics associated with each procedure and source line are organized as a hierarchy defined by a set of SDDF records. As Figure 2 shows, this *meta-meta-format* hierarchy contains three groups of SDDF record descriptors: *mapping*, *configuration* and *statistic*.

Mapping records define the set of statistics associated with each instrumentable construct. In turn, configuration records indicate the statistic record names and allow the SvPablo interface to extract the base names of all performance metrics before reading the statistics records, which define the actual performance metrics.

This meta-configuration is the key to SvPablo's extensibility. Tool developers can add new metrics to SvPablo simply by updating the mapping and configuration records and then generating the desired set of statistic records. Below, we describe the meta-format for both event (line) and procedure statistics.

5.1 Event Statistics

The performance data associated with an instrumentable construct is represented by a single SDDF *Event Statistics Mapping* record and by a set of *Statistics* records, one for each associated performance metric. The mapping record defines the general information about the instrumentation point (e.g., its source code location) and specifies a group of pointers to statistics records for the actual performance metrics.

As an example, consider the mapping for event 70, shown in Figure 2. The event mapping record indicates two types of associated configuration and statistic records: "LOOP" and

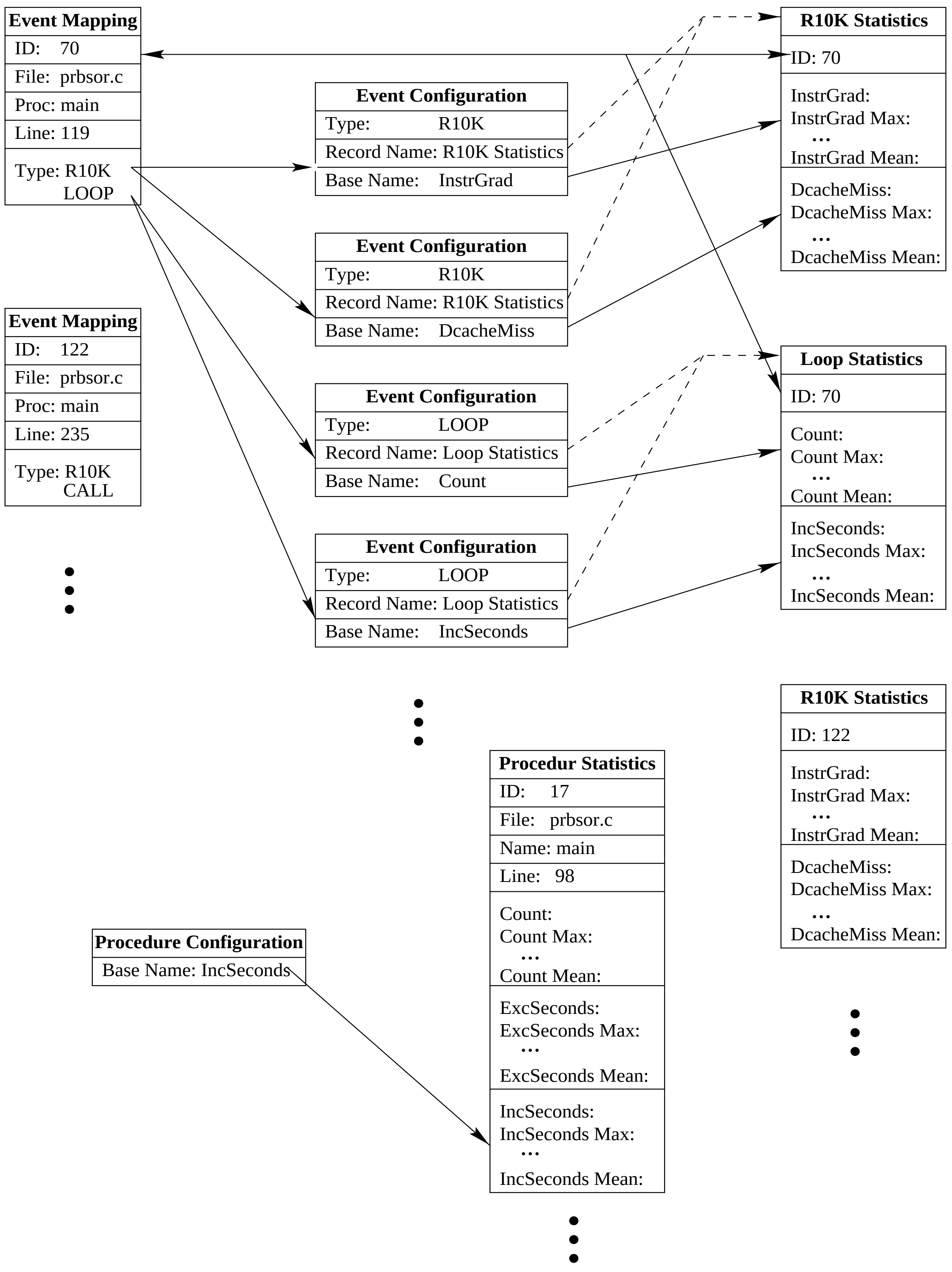


Figure 2: SvPablo SDDF Record Hierarchy

```
// "description" "Performance Statistics Based On Event ID"
"R10K stats" {
    int "Event ID";
    // "InstrGrad" "Instructions Graduated"
    double "InstrGrad"[];
    double "InstrGrad Max";
    int "InstrGrad Max Node";
    double "InstrGrad Min";
    int "InstrGrad Min Node";
    double "InstrGrad Mean";
    double "InstrGrad Std Dev";
    // "DcacheMiss" "Data Cache Misses"
    int "DcacheMiss"[];
    int "DcacheMiss Max";
    int "DcacheMiss Max Node";
    int "DcacheMiss Min";
    int "DcacheMiss Min Node";
    double "DcacheMiss Mean";
    double "DcacheMiss Std Dev";
};;
```

Figure 3: R10000 Hardware Performance Counters Record Descriptor

"R10K." The configuration records then define the base names for all statistics associated with the "LOOP" and "R10K" metrics. In turn, Figure 3 shows the SDDF statistics record descriptor for the MIPS R10000 counter data for this example. The event identifier specifies the source code location by correlating it to parent mapping record.

Each metric (e.g., data cache misses) includes data for each process as a vector, together with standard statistics (i.e., minima, maxima, mean, and standard deviation). SvPablo uses the vector to present performance data for each process and the statistics to present data for each source code line. For each metric in the statistics record descriptor, there is an event configuration record which specifies the base name of the metric. These base names are used by SvPablo to read the corresponding performance data.

5.2 Procedure Statistics

Finally, a separate set of SDDF procedure statistics records define the performance metrics associated with all procedures. These records contain both mandatory and optional fields. The mandatory fields include procedure mapping information and two metric fields — the number of calls to the procedure and the exclusive duration of the procedure (i.e., its execution time excluding all descendent procedures).

As illustrated in Figure 2, the mapping information is provided by the procedure statistics record and includes: a unique procedure identifier, the name of the file containing the procedure, the procedure name, and the procedure's first source code line. The SvPablo interface uses these fields to display the list of procedures and their metrics, shown in Figure 1, and the associated performance pop-up dialogs.

Finally, the remaining, optional fields define the data needed for the detailed performance metric displays. As illustrated in Figure 2, the base names of these additional metrics are specified in the procedure configuration records.

6 Application Tuning Example

The true test of any performance tool is its effectiveness when applied to poorly performing applications in a realistic context. To assess the utility of SvPablo and the possible usability penalties induced by our design's emphasis on language independence and portability, we worked with application developers from the National Center for Supercomputing Applica-

Statement	**Exclusive Seconds**	**MFLOPS**	**Send Message Duration**	**Receive Message Duration**
loop 10	6.1		0.25	5.71
(73)	91.3	0.04	5.61	84.00
(82)	94.8	0.04	5.94	86.95

Table 1: Average Metric Values (Array “`fs`” in Baseline Code)

tions (NCSA). Based on this experience, we present a case study on the SGI Origin 2000 that demonstrate SvPablo’s effectiveness when analyzing performance and tuning codes.

This case study explores the performance of a numerical model simulating cloud and density current dynamics [5]. The code is a three-dimensional, non-hydrostatic, finite difference, convective cloud model that utilizes a quasi-compressible version of the Navier-Stokes equations. Originally written in CM Fortran for the CM5, the code was later translated to High-Performance Fortran, yielding approximately 9000 lines of HPF code. We executed two versions of the program (baseline and modified), using 8 processors.

Figure 1 shows the SvPablo interface displaying the execution behavior of the original code on eight processors of an SGI Origin. The pane `Routines in Performance Data`, which is sorted by routine duration, indicates that the main bottleneck in the program is the routine “`s_mix`”, which had a cumulative time of 197.20 seconds. Via SvPablo, we identified the computation of “`X - Mixing`” and “`Y - Mixing`” in the routine “`s_mix`” as the primary bottleneck. The average metric values for the two most time consuming statements in the code and the enclosing loop, summarized in Table 1, indicate that the bottleneck was caused by communication inside the loop.

Three pertinent observations are revealed by the data shown in Figure 1:

- The compiler-synthesized communication required by the circular shift (“`cshift`”) on the array “`flx`” is primary cause of the poor performance observed for this HPF statement. This is denoted by the dark color bars for the communication metric.

- The previous assignment also contains circular shifts and would expect compiler-synthesized communication as well. However, SvPablo shows that no communication occurred (indicated by the absence of communication metric data in the associated columns).

- SvPablo assigns communication costs to a loop, rather than to its component statements, if some loop-related communication occurs prior to execution of the loop body. For this loop, no loop-related communication should be necessary. However, SvPablo shows that the HPF compiler did synthesize some communication prior to the loop body.

Based on these observations, one can deduce the cause for the performance difference between the two circular shifts. Succinctly, the arrays “`kmh`” and “`s`”, which must be shifted for the computation of “`flx`,” are loop independent. Hence, the HPF compiler can prefetch the necessary data before loop execution begins. However, the array “`flx`” is computed in the first statement of the loop and used in the second. Hence, the data must be circularly shifted during each loop iteration.

To improve the performance of this routine, we split the original loop, forming four loops, one for each statement inside the original loop, as shown in Figure 4. By splitting the loop, we expected the HPF compiler to begin migrating data as soon as an iteration of the previous

Statement	**Exclusive Seconds**	**MFLOPS**	**Send Message Duration**	**Receive Message Duration**
$\sum$ four loops	10.54		0.50	9.38
(73)	0.23	16.3		
(82)	0.28	14.0		

Table 2: Average Metric Values (Array "`fs`" in Modified Code)

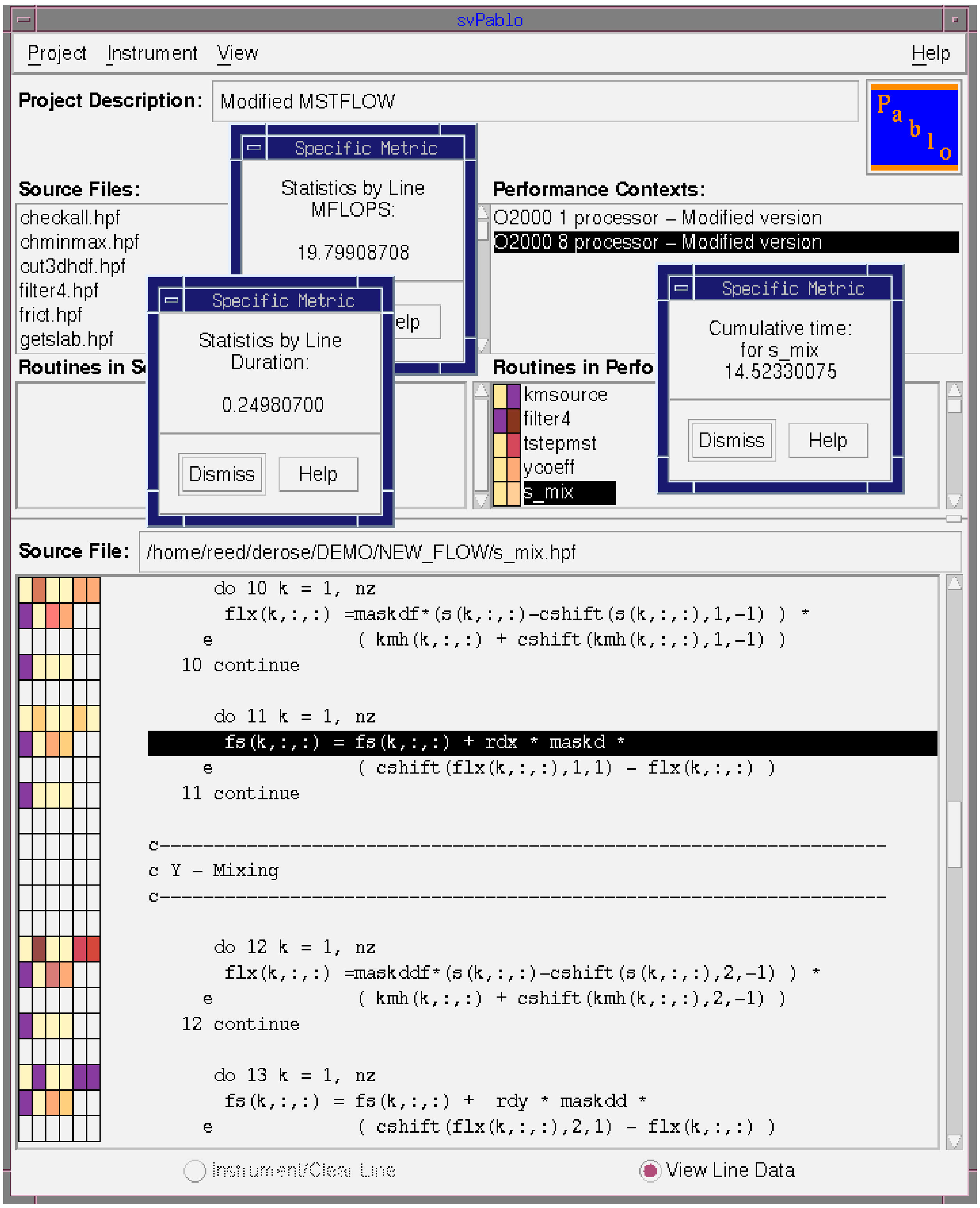
loop is completed, overlapping computation and communication.

After splitting the original loop, we again executed the code and measured its performance with SvPablo. As expected, communication occurred between loops, reducing the subroutine's execution time by an order of magnitude, from 197.2 seconds to 14.5 seconds.

Table 2 shows the average metric values for the same two statements and for the sum of the execution times of the four loops. Clearly, the additional overhead of the loops is very small compared to the performance gained by overlapped communication. Moreover, the mean MFLOPS rate for each processor increased by four orders of magnitude.

7 Related Work

Performance Evaluation tools can normally be classified into two major categories, the ones that try to predict performance (see for example P^3T [4]) and the ones that measure the performance of a program during run-time. SvPablo falls into the latter category, which has been employed by several efforts described in the literature, targeting both sequential and parallel systems — far more than can be discussed here. Notable examples include Paradyn [6], developed at the University of Wisconsin, the Pablo Performance Analysis Environment [11], developed at the University of Illinois, and the Automated Instrumentation



Project Description: Modified MSTFLOW

Source Files:
checkall.hpf
chminmax.hpf
cut3dhdf.hpf
filter4.hpf
frict.hpf
getslab.hpf

Routines in S

Specific Metric
Statistics by Line
MFLOPS:
19.79908708

Specific Metric
Statistics by Line
Duration:
0.24980700

Performance Contexts:
O2000 1 processor – Modified version
O2000 8 processor – Modified version

Specific Metric
Cumulative time:
for s_mix
14.52330075

Routines in Perfo
kmsource
filter4
tstepmst
ycoeff
s_mix

Source File: /home/reed/derose/DEMO/NEW_FLOW/s_mix.hpf

```
      do 10 k = 1, nz
       flx(k,:,:) =maskdf*(s(k,:,:)-cshift(s(k,:,:),1,-1) ) *
     e             ( kmh(k,:,:) + cshift(kmh(k,:,:),1,-1) )
   10 continue

      do 11 k = 1, nz
       fs(k,:,:) = fs(k,:,:) + rdx * maskd *
     e             ( cshift(flx(k,:,:),1,1) - flx(k,:,:) )
   11 continue

c-----------------------------------------------------------------
c Y - Mixing
c-----------------------------------------------------------------

      do 12 k = 1, nz
       flx(k,:,:) =maskddf*(s(k,:,:)-cshift(s(k,:,:),2,-1) ) *
     e             ( kmh(k,:,:) + cshift(kmh(k,:,:),2,-1) )
   12 continue

      do 13 k = 1, nz
       fs(k,:,:) = fs(k,:,:) +  rdy * maskdd *
     e             ( cshift(flx(k,:,:),2,1) - flx(k,:,:) )
```

Figure 4: Performance Data (MSTFLOW HPF Modified Code)

and Monitoring System (AIMS) [13], developed at NASA Ames Research Center.

Paradyn is a tool for measuring the performance of large-scale parallel programs. It performs dynamic instrumentation on long-running programs, in search of performance problems. Its main difference from SvPablo and other performance measurements tools is that instrumentation and visualization is performed during run-time.

The Pablo Performance Analysis Environment consists of several components for instrumenting and tracing parallel programs and for analyzing the trace files produced by the instrumented programs. SvPablo used and extended some of these components, such as the SDDF library [3] and the C parser and GUI from iPablo [12]. However, it differs from the Pablo Environment in that it performs run-time summarization instead of collecting trace data, and the data capture and presentation components are integrated into the same graphical performance browser.

A work closely related to the Pablo Environment, is AIMS, a software toolkit for performance evaluation of parallel applications on multiprocessors. AIMS accepts Fortran and C parallel programs written using two message passing libraries: MPI and PVM. It has three major software components: a source code instrumentor, a run-time performance-monitoring library, and a suite of tools that process and display execution data. SvPablo differs from AIMS in that SvPablo performs run-time summarization and has the instrumentation and visualization components integrated.

In general, the majority of the performance measurement tools focus on particular programming models (e.g., message passing or data parallel) or specific hardware/software platforms. In contrast, SvPablo is designed to be independent of both programming model and architecture, allowing developers to re-target the performance analysis infrastructure simply by changing SvPablo's meta-data specification of performance metrics.

8 Conclusions

In this paper, we described the design of SvPablo, a toolkit for instrumentation, data capture and analysis of sequential and parallel codes. SvPablo supports both software performance measurement and low overhead access to hardware performance counters, summarizing the performance data during program execution. Working with a group of large-scale application developers, we observed that SvPablo enabled us to rapidly identify and correct performance bottlenecks.

However, the key feature of SvPablo is its language and architecture transparency, achieved by representing performance data via a meta-meta-format presentation of events from different languages using the same graphical interface. The flexible structure of the performance file also permits the ready introduction of new metrics and support of new languages without requiring modifications to the graphical user interface.

References

[1] Adve, V., Mellor-Crummey, J., Wang, J.-C., and Reed, D. Integrating Compilation and Performance Analysis for Data-Parallel Programs. In *Proceedings of Supercomputing'95* (November 1995).

[2] Adve, V. S., Mellor-Crummey, J., Anderson, M., Kennedy, K., Wang, J., and Reed, D. A. Integrating Compilation and Performance Analysis for Data-Parallel Programs. In *Proceedings of the Workshop on Debugging and Performance Tuning for*

Parallel Computing Systems, M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, Eds. IEEE Computer Society Press, 1994.

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

[4] Fahringer, T. Estimating and Optimizing Performance for Parallel programs. *IEEE Computer 28*, 11 (November 1995), 47–56.

[5] Lee, B. D., and Wilhelmson, R. B. The Numerical Simulation of Non-Supercell Tornadogenesis. In *18th Conference on Severe Local Storms* (February 1996).

[6] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer 28*, 11 (November 1995), 37–46.

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

[8] MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual*, version 2.0 ed. 2011 N Shoreline Blvd; PO Box 7311; Mountain View, CA 94039-7311, October 1996. http://www.sgi.com/MIPS/products/r10k/UMan_V2.0/R10K_UM.cv.html.

[9] Pancake, C. M., Simmons, M. L., and Yan, J. C. Performance Evaluation Tools for Parallel and Distributed Systems. *IEEE Computer 28*, 11 (November 1995), 16–19.

[10] The Portland Group, Inc. *PGHPF User's Guide*, 1994.

[11] Reed, D. A., Aydt, R. A., Noe, R. J., Roth, P. C., Shields, K. A., Schwartz, B., and Tavera, L. F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference* (1993), A. Skjellum, Ed., IEEE Computer Society.

[12] Shields, K. A. iPablo User's Guide. Tech. rep., Department of Computer Science at the University of Illinois at Urbana-Champaign, December 1994.

[13] Yan, J. C., Sarukkai, S. R., and Mehra, P. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice & Experience 25*, 4 (April 1995), 429–461.

[14] Zagha, M., Larson, B., Turner, S., and Itzkowitz, M. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of Supercomputing'96* (November 1996).