An Overview of the Pablo Performance Analysis Environment

Daniel A. Reed*
Ruth A. Aydt
Tara M. Madhyastha
Roger J. Noe
Keith A. Shields
Bradley W. Schwartz

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

November 7, 1992

Copyright © 1992 The University of Illinois Board of Trustees.
All Rights Reserved.

*Supported in part by the Defense Advanced Research Projects Agency under DARPA Contract Number DABT63-91-K-0004, by the National Science Foundation under grants NSF CCR87-06653 and NSF CDA87-22836, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, and by an equipment grant from the Digital Equipment Corporation External Research Program.

Contents

1 Introduction **1**

2 Pablo Design Philosophy **2**
2.1 Portability 2
2.2 Scalability 3
2.3 Extensibility 3
2.4 Pablo Software Overview 4

3 Pablo Instrumentation Software **6**
3.1 Instrumentation Software Components 6
3.2 Instrumentation Options 8
3.3 Adaptive Instrumentation Control 9

4 A Pablo Performance Instrumentation Environment Example **10**

5 Pablo Trace Data Format **11**
5.1 Trace Format Requirements 11
5.2 A Self-Describing Trace Data Format 13

6 Pablo Data Analysis Environment **14**
6.1 Graphical Programming Models 16
6.2 Data Analysis Graph Components 16
6.3 Data Binding and Module Execution Rules 19
6.4 Analysis Module Palette 20
6.5 Parallel Execution and Display Control 21

7 Pablo Data Sonification **22**
7.1 Sonification Software Structure 23
7.2 Sonification Experiences 24

8 A Pablo Performance Analysis Environment Example **25**

9 Potential Pablo Applications **27**

10 Performance Data Immersion **28**

11 Data Parallel Programming and Performance Analysis **29**

12 Current Status and Software Plans **30**

List of Figures

1 Pablo Instrumentation Software 7
2 *iPablo* Graphical Instrumentation Interface 34
3 *iPablo* Instrumentation of One Procedure 35
4 Sample Pablo trace file (ASCII format) 36
5 Pablo Data Analysis Module Components 37
6 Pablo Graph Construction 38
7 Pablo Analysis Graph Configuration 39
8 Pablo Analysis Graph Execution 40
9 Performance Tool/Compiler Integration 41

Abstract

As massively parallel, distributed memory systems replace traditional vector supercomputers, effective application program optimization and system resource management become more than research curiosities — they are crucial to achieving substantial fractions of peak performance for scientific application codes. By recording dynamic activity, either at the application or system software level, one can identify and remove performance bottlenecks. Pablo is a performance analysis environment designed to provide performance data capture, analysis, and presentation across a wide variety of scalable parallel systems. The Pablo environment includes software performance instrumentation, graphical performance data reduction and analysis, and support for mapping performance data to both graphics and sound. Current research directions include complete performance data immersion via head-mounted displays and the integration of Pablo with data parallel Fortran compilers based on the emerging High Performance Fortran (HPF) standard.

1 Introduction

As computational science becomes an equal partner to theory and experiment, there is growing consensus that massively parallel systems are the only technically and economically viable approach to achieving the computing power needed to effectively study the "Grand Challenge" problems (e.g., ocean and climate modeling, computational fluid and combustion dynamics, or computational drug design) [10]. Several vendors have announced or begun delivery of massively parallel systems whose potential performance exceeds three hundred gigaflops, all having clear performance growth paths to multiple teraflops within three years. All these systems are constructed by coupling large numbers of standard, high-performance microprocessors via a high-speed communication network. For example, the Paragon XP/S parallel system [17], the commercial realization of the Intel/DARPA Touchstone project [19], is based on the high-performance i860/XP microprocessor and provides 200 megabyte/second, full-duplex communication links with software message passing latency measured in microseconds; the peak performance of the largest hardware configuration approaches 300 gigaflops.

As massively parallel, distributed memory systems replace traditional vector supercomputers, effective application program optimization and system resource management become more than research curiosities — they are crucial to achieving substantial fractions of peak performance for scientific application codes. Unfortunately, effectively managing a massively parallel system is fraught with the same difficulties one faces when managing a large, human organization — information acquisition problems and personnel interactions make decision procedures more error prone. Moreover, in a parallel system, the interactions occur on a microsecond time scale, and the potential information volume is much greater.

Not only do we currently have no general methods for predicting application or system behavior on massively parallel systems, there are few extant tools for post-execution analysis. However, by recording dynamic activity, either at the application or system software level, one can identify and remove performance bottlenecks. To gain insight from this data and to tune both application and system software, the data must be processed and presented in ways that not only show trends but also allow detailed exploration of small scale behavior.

The remainder of this paper describes the design philosophy and software components of Pablo, a performance analysis environment designed to provide performance data capture, analysis, and presentation across a wide variety of scalable parallel systems. In §2, we summarize the software's design philosophy, describe the requirements for a portable performance data analysis system, and outline Pablo's software components. Following the discussion of the Pablo design, we describe the software instrumentation, its graphical user interface, and an example of its use in §3–§4. In §5, we describe the performance trace data format, followed in §6–§8 by a description of the data analysis environment and an example of its use. In §7, we describe Pablo's integration of sonification (i.e., the mapping of time varying data to sound) with data analysis, followed in §8 by an illustration of Pablo's use. Given Pablo's data reduction, display and sonification features, §9 describes our plans to study system software performance and resource management issues, followed in §10-§11 by a description of our current work to extend Pablo's functionality to include both data immersive presentation via head-mounted displays and integration with the Rice Fortran D compiler [15] Finally, §12 summarizes the current environment state and plans for future software development.

2 Pablo Design Philosophy

The performance variability of parallel computer systems with even modest numbers of processors is well documented [3]. Larger numbers of processors exacerbate the already difficult problems of bottleneck identification and performance tuning. The primary reason for the increased difficulty is not the number of processors, but rather the greater complexity and frequency of component interactions as well as the number of potential optimization gradients. Typically, the space of potential performance optimizations is poorly understood, rarely convex, and of very high dimension. In addition, the sensitivity of performance to the values of certain system parameters (e.g., the balance of communication latency and bandwidth, the data distribution across secondary storage devices, or task assignment to processors) can be determined only experimentally.

The precursor to performance optimization is insight — one must understand not only the effects of potential optimizations, but also *how* and *why* they affect the system. Historically, computer systems have been studied experimentally by subjecting the test system to a known load and capturing static, aggregate performance measures such as throughput, response time, and utilization [7]. Similarly, the performance of application programs often has been assessed via code profiles that reveal the distribution of execution time across the code. Although these application and system performance measures have been effective for systems with only a small number of processors, they do not provide sufficient information to easily understand massively parallel systems; the complexity of the performance optimization space is too great.

Like an archeologist who must infer physical structure, habitat, and activity from a few physical artifacts, static performance measures often force users and system software designers to infer system or application behavior from summary data. Given the complexity of component interactions in a massively parallel system, the inference process is tedious and error prone. Simply put, traditional, static performance measures specify what happened (e.g., processor utilization was low), but they reveal frustratingly little about the more important issues: why and how it happened. To understand these issues, one must capture and analyze the dynamics of application, system software, and hardware interactions. Hardware designers use logic analyzers to systematically capture and simultaneously display transient signals in different portions of a hardware design; designers of parallel application and system software need comparable tools.

Drawing on the logic analyzer analogy, an ideal performance analysis environment should support interactive insertion of instrumentation points, as well as data analysis, reduction, and display. Moreover, the environment should be *portable* across a range of parallel architectures, its performance and data analysis capabilities should be *scalable* with the size of the system being studied, and it should be *extensible*, allowing users to add environment functionality as needed. Below, we elaborate on these triune goals, and then describe our approach to these issues in the Pablo performance environment.

2.1 Portability

Repeated experience has shown that scientific application software developers users will eschew powerful, but complex tools in favor of inferior, but easily understood tools. Unless compelled by circumstances, most such users are unwilling to invest great time and effort to learn the syntax and semantics of new performance tools; they often view performance optimization as an unavoidable evil. Hence, portability and ease of use are critical to the acceptance of new performance tools. Indeed, to ensure widespread use, one's design goal must be the creation of a *de facto* standard.

Not only does portability permit use of a familiar set of tools in otherwise disparate parallel software environments, it admits the possibility of cross-architecture performance comparisons. By capturing and analyzing equivalent application performance data on multiple parallel machines, one can study the effects of system software and architecture on application performance. For example, changes in task scheduling policies, input/output performance, memory management algorithms, the ratio of floating point and integer performance, and interprocessor communication latency and bandwidth can shift application program bottlenecks or change the application's critical path. By analyzing this data for a sufficiently large suite of application programs, one can identify the strengths and weakness of each system, providing feedback for the design of next generation systems.

2.2 Scalability

Scalability is a key characteristic of the new generation of massively parallel systems; by adding processors one can incrementally increase performance without replacing existing hardware or changing the underlying software. As an example, the Intel Paragon XP/S system is designed to scale from 64 to over 500 processors, with planned expansion to 2048 processors [17]; the Thinking Machines CM-5 scales over a similar range [39]. Thus, portability is a necessary, but not sufficient, condition for a performance environment; scalability is also required. As the parallel system scales in size and performance, the performance environment must scale commensurately.

Scalability of a performance environment not only implies that the environment must be capable of capturing and analyzing data from very large numbers of processors, it must also be capable of presenting that data in ways that are intuitive and instructive. Near real-time reduction and analysis of time varying performance data from hundreds or thousands of processors is itself a computationally intensive task, and at least some portions of the performance data analysis environment must be resident on a parallel computer system. Parallelizing the analysis of performance data, though non-trivial, pales by comparison to the daunting task of effectively presenting the data.

Current graphical techniques often represent the states of individual processors (e.g., by a colored square for each processor). These techniques clearly do not scale to thousands of processors; graphics screens lack the requisite pixel count. Moreover, this approach is intellectually dubious. For a massively parallel system, processor behaviors often form a small number of equivalence classes; it frequently suffices to see aggregate behavior with detail for equivalence class representatives and outliers. Hence, new display idioms are needed if performance environments are to scale with massively parallel systems.

2.3 Extensibility

Finally, a performance environment must be extensible, allowing its users to interact with the data, to change the types of data analyses and to add new analyses as needed. The desire for extensibility must be carefully balanced against ease of use; all too often, software tools flounder on the twin shoals of specificity and generality. If a tool's functionality is too limited, it will not be used. Conversely, if a tool is too general and does not support common cases in obvious ways, users will abandon it in frustration.

Based on our experience designing, implementing and using two previous generations of performance analysis software [23, 26], we believe there are three classes of potential performance

environment users: novice, intermediate, and expert. Novice users know relatively little about parallel machine software or hardware, nor do they wish to learn more than the minimum necessary to optimize the performance of their application codes. They want a performance tool that is simple and easy to use and that will quickly identify performance bottlenecks; they are unlikely to modify or expand the performance environment's functionality.

In contrast, intermediate users often wish to conduct performance experiments, asking such questions as "What caused this behavior?" or "How do these performance metrics interrelate?" Although they are unlikely to be willing to extend the performance environment by writing new software, these users do want a modicum of control over the environment's behavior. For example, they may wish to compute new performance metrics from the measured data and to compare them to other metrics. Simply put, this user class needs an environment toolkit whose components can be assembled in a wide variety of ways.

Finally, expert users are intimately acquainted with the parallel architecture and system software. Indeed, they may wish to use the performance environment to study the effects of system software modifications. These users need the broadest latitude, subsuming the needs of both the novice and intermediate users. Not only will they wish to reassemble the existing components of the environment toolkit, they will want to add new toolkit components. Moreover, they expect the added components to integrate seamlessly with extant elements.

2.4 Pablo Software Overview

The design of the Pablo performance analysis environment draws on the lessons learned from the design, implementation and use of two previous generations of performance analysis software [23, 26]. Chief among these were the importance of portability, scalability, and extensibility.

One of our earlier performance environments, developed for the Intel iPSC/2 hypercube, included instrumentation of Intel's NX/2 operating system [34, 32] to record operating system calls, context switches, message passing activity, and application-specified data. We also modified the Free Software Foundation's gcc compiler to emit application program code with embedded calls to the operating system instrumentation library [25]. In collaboration with Intel, we also developed prototype hardware to unobtrusively capture the operating system and application software instrumentation data [24]. Using performance data, we created a graphical data analysis and display environment that supported a fixed set of data reductions and a variety of graphical displays [26]. Users could choose from the cross product of data analyses and data displays.

Although this environment proved useful in studying the performance of both parallel applications [38, 18] and the Intel iPSC/2's parallel file system [1], the dependence of performance data capture on modifications to a proprietary operating system precluded both wide distribution and porting to new architectures. The performance data file format was designed to support only distributed memory parallel systems, and the performance data analyses were intimately tied to the data source. Finally, although we supported arbitrary, pairwise combinations of the extant data analyses and displays, it was impossible to extend either the possible combinations of analyses and displays or to add new components without intimate knowledge of the environment's internal software structure.

Based on this experience, the bulk of the Pablo environment design effort has centered on supporting portability, scalability, and extensibility. Pablo is best viewed as a toolkit for the construction of performance analysis environments. As such, it consists of two primary components:

(1) portable software instrumentation, and (2) portable performance data analysis, with a trace data meta-format coupling the instrumentation with the data analysis. The instrumentation, data format, analysis environment, and display options are briefly described below; details can be found in §3–§8.

Pablo's portable software instrumentation has been designed to support interactive specification of source code instrumentation points. The software instrumentation can be used to gather performance data about either system or application codes, though our initial target is the latter. As part of the instrumentation, we have developed and are continuing to develop three pieces of software: a graphical interface for instrumentation specification, modified C and Fortran parsers that emit instrumented source code, and a library of trace capture templates whose members can capture performance data generated by the instrumented source code when it is executed on distributed memory parallel systems. Our initial architectural targets are the Thinking Machines CM-5 [39] and the Intel Paragon XP/S systems [17]. As exemplars of common, massively parallel architectures, availability of software performance instrumentation for these machines will aid a large and growing user community.

The performance analysis component of Pablo consists of a set of data transformation modules that can be graphically interconnected, in the style of aPE or AVS [37], to form an acyclic, directed data analysis graph. Performance data flows through the graph nodes and is transformed to yield the desired performance metrics. Each performance data transformation module consists of a data transformation core (i.e., the actual data transform) and a system-provided data access "wrapper" that can read and write a self-documenting data stream meta-format that includes internal definitions of data types, sizes, and names, but does not include embedded semantics. The wrapper's data access methods permit retrieval and generation of data field information without *a priori* knowledge of the data stream structure, and associated software supports data replication and distribution to other modules.

Although performance analysis occasionally requires knowledge of architecture-specific data semantics, the Pablo design philosophy presumes that embedding this information in either the trace data format or the analysis software modules will preclude module reuse on differing architectures or application software environments. For this reason, the performance data format has *no* embedded semantics (i.e., there are no predefined event types or data sizes). Similarly, most data transformation modules provide architecture-neutral data reductions (e.g., averages or histograms) on user-specified data stream components. Thus, one can write a data transformation module that maintains a running average without knowing the range of data values, their semantic interpretation, or their storage scheme.

The performance data analysis software is based on an object-oriented design, written in C++, and by virtue of the performance data meta-format and the architecture-neutral data reductions, designed to be easily ported to new machine architectures. Also, because the data analysis modules communicate solely via message passing, individual modules can potentially execute in parallel, allowing the construction of scalable data analysis graphs.

In addition to dynamic X window graphics for the display of performance data, Pablo supports the use of sonic data presentation via the replay of sampled sounds on a Sun SparcStation audio port or the use of a sound synthesizer via the Musical Instrument Digital Interface (MIDI).

3 Pablo Instrumentation Software

Historically, performance measurement and instrumentation techniques have included profiling, sampling, and event tracing. Execution profiles that show the distribution of execution time across application program routines are the most common form of performance data. Although extremely useful — they show the source code location of performance bottlenecks, they do not reflect performance variations over time (e.g., time varying parallelism), nor can they be used to identify the causes for poor performance. From a measurement perspective, the primary advantage of profiles is that they can be calculated at execution time and require only a small amount of storage. Conversely, sampling the system state periodically can provide time varying detail, but the samples are not correlated with particular logical states of program execution. In short, sampling the system state shows *when* the system is operating effectively, but it provides no information about logical causes (i.e., *where* in code). Tracing is the most general, and based on our earlier argument that understanding parallel system dynamics was crucial, provides the best means to capture and analyze the time varying behavior of complex parallel systems.

If one views the execution of a parallel computer system as a sequence of events, each representing some significant physical or logical activity (e.g., entry to a procedure or the beginning of a loop's execution), an event trace is a recorded sequence of the events. Data associated with each trace event typically include the identity of the event (i.e., what happened), the time the action occurred (i.e., when it happened), information about where the event occurred (e.g., the processor and task), and any additional data that further define the computer system state. Event tracing subsumes both profiling and sampling; each trace event includes information on where, what, and when; from this trace, one can construct both execution profiles and descriptions of time varying behavior.

Ideally, a portable implementation of event tracing should support both user specification of instrumentation points and subsequent capture of the specified data during execution. And, as we discussed in §2, the performance analysis system should be scalable and extensible. Finally, a trace library should balance event data rates against undue perturbation of the instrumented code.[1] Below, we describe the design of the Pablo instrumentation library and our approach to satisfying these goals.

3.1 Instrumentation Software Components

As Figure 1 suggests, the three components of the Pablo instrumentation software are a graphical interface for interactively specifying source code instrumentation points, C and Fortran parsers that emit source code with embedded calls to a trace capture library, and a trace capture library that records the performance data. This approach is necessarily a compromise to maximize portability. Compiler support for instrumentation is clearly preferable[2] (e.g., to avoid precluding certain optimizations), as is hardware and operating system support for data capture (e.g., to reduce the instrumentation perturbation). However, these approaches require access to vendor hardware and system software, precluding portability. Our intent is that the three independent, though cooperating, instrumentation software components can be quickly ported to a new parallel system or the

[1] The only thing more disheartening than inadequate data is a large volume of data whose accuracy is suspect.

[2] We will return to the issue of compiler support for instrumentation synthesis in §11, when we discuss our current plans for integration of the Pablo environment with the Rice experimental compiler for the data parallel Fortran D language.

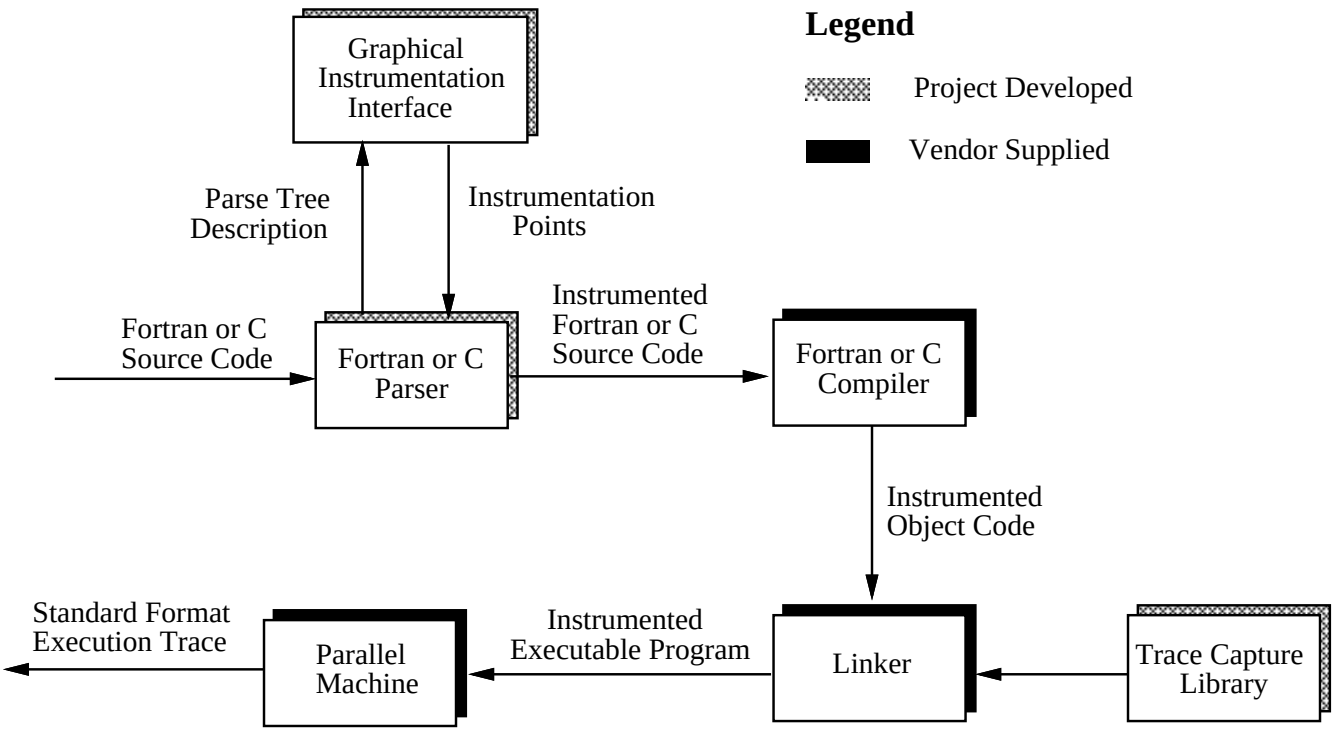

Figure 1: Pablo Instrumentation Software

individual components can be replaced by machine specific versions.

The parsers accept source code to be instrumented and produce the parse tree information needed by the graphical instrumentation interface.[3] The graphical interface, based on X and Motif, interprets the parse tree data information and allows the user to graphically specify source code instrumentation points.

Although the trace capture library supports the instrumentation and capture of data from the manual instrumentation of arbitrary source code points, at present, the graphical instrumentation interface supports only the interactive instrumentation of entry/exit points for procedure calls and outer loops. This limitation is intentional; based on our earlier experience developing instrumentation for both shared and distributed memory parallel systems, we believe loop and procedure entry/exit instrumentation is the most valuable when tuning application performance, and it can be used to compute a wide variety of dynamic performance metrics.[4] Moreover, it is the most detailed level of instrumentation possible without precluding major optimizations by the compiler; as interprocedural analysis becomes more common, additional compiler support may be required even at this level. Finally, it strikes the right balance of detail and flexibility — additional instrumentation excessively perturbs the computation, but less may not provide sufficient information. Given

[3] At this writing, the C language parser, the trace capture library and the graphical instrumentation interface are complete. We are currently designing the Fortran parser and extending the trace capture library to include new parallel architectures, notably the Thinking Machines CM-5.

[4] Because most distributed memory parallel systems implement message passing via calls to a message passing library, procedure entry/exit tracing also captures task interactions.

the specification of instrumentation points, the parsers emit modified source code with embedded instrumentation. At execution time, the inserted instrumentation code invokes tracing routines supplied by the trace capture library, producing performance data in a standard trace format, described in §5.

Because Pablo's only modification to source code is the insertion of calls to the trace capture library, it is possible to move an instrumented program to another parallel system. If the trace capture library has been ported to the new parallel system, the same application data can be captured there, permitting cross-architecture performance comparisons. All the idiosyncrasies of generating event timestamps, buffering data, and extracting that data from a particular parallel machine are isolated in the Pablo trace capture library.

Obviously, the design of an effective trace capture library will differ markedly based on the underlying architecture. However, by implementing a documented trace capture library for the Thinking Machines CM-5 and Intel Paragon XP/S, we believe we can provide "reference" implementations that can be quickly ported to other systems by either vendors or users. In short, the Pablo instrumentation software provides a wide range of potential instrumentation options and instrumentation perturbation controls, allowing it to be easily customized for a particular machine architecture.

3.2 Instrumentation Options

To balance the countervailing needs of detailed data and minimal perturbation, the Pablo instrumentation software supports three class of instrumentation events: trace, count, and time interval. Instances of each event class are recorded in the trace file using a self-documenting data format that includes internal definitions of data types, sizes, and names; see §5 for complete details. Minimally, a timestamp, event identifier, and processor number are recorded with instances of events in each class.

Each of the three event classes lies at a different point in the spectrum of detail and perturbation. Trace events represent the occurrence of a specific event (e.g., a particular procedure was called at this time by this processor), and a trace file entry is produced for each instance.[5] Moreover, instances of these events can be accompanyed by an arbitrary amount of user-specified data.

In contrast, count events contain no user data; only the count of the number of occurrences is incremented, and the time of each event occurrence often is not significant. The Pablo trace capture library allows users to specify when count event records should placed in the trace file (e.g., when the count value has increased by a particular quantity). In the limiting case, for procedure and loop entry/exit instrumentation, this produces an execution summary of procedure and loop invocations.

Finally, time interval events associate an event with a pair of source code points (i.e., a logical code fragment). Each occurrence produces an event containing the time that elapsed during execution of the source code fragment. Because the association is with logical, rather than physical, code fragments, one can delimit logically significant code sections and measure their time varying contribution to total execution time.

For events of all three classes, the Pablo trace capture library supports optional, user-written extension functions that can process event data before it is written to the trace file. Via this mechanism, users can create higher-level events (e.g., by combining multiple events), selectively discard events, or modify event record fields using knowledge of the application software's behavior.

[5] Subject to the perturbation monitoring described in §3.3.

As an example, one might create a routine that dynamically computes procedure profiles from procedure entry/exit data. This mechanism, though deceptively simple, empowers users to extend the Pablo instrumentation in a variety of ways. In particular, we have used it capture the parameters of message passing calls on distributed memory parallel systems and to compute inclusive and exclusive lifetimes of procedure calls (i.e., the procedure duration inclusive and exclusive of time spend in descendent procedures).

3.3 Adaptive Instrumentation Control

Excessive perturbation of program behavior remains the bane of event tracing and has limited its widespread use. Potential perturbations range from the benign (e.g., simple slowdown) to malignant (e.g., changing the code's critical path or reordering events). Moreover, in the hands of a novice, event tracing can produce prodigious volumes of inaccurate data. Users instrument procedures without *a priori* knowledge of procedure invocation frequency, which can lead unintentionally to large data volumes. Thus, it is imperative that the Pablo trace capture library provide some safeguards for data volume and accuracy.

There are two possible techniques that can reduce event tracing overheads to acceptable levels. The first is to develop hardware that supports capture and recording of software-generated events [24, 12, 13, 29, 28]. Although ideal, this approach is necessarily machine-specific and inherently non-portable; developing such hardware must remain the responsibility of parallel system vendors.[6]

The second approach is portable and relies on software event recording, but limits the number of events — this is one of the reasons the Pablo graphical instrumentation interface limits users to the interactive specification of only procedure and outer loop entry/exit events. The thoughtful reader will realize that restricting the class of possible events is not always sufficient to bound the maximum possible event rate, nor is restricting the class of possible events always desirable. Thus, in addition to allowing the user to specify instrumentation points via the graphical instrumentation interface, the Pablo trace library has been designed to reduce the likelihood of malignant perturbations by monitoring and dynamically altering the volume, frequency, and types of event data recorded. If events with relatively large amounts of associated trace data occur at rates higher than the user desires, the trace library will automatically substitute less invasive data recording (e.g., periodic event counts rather than complete event traces). If the event rate returns to more modest levels, the trace library will resume recording of all events.

Pablo's implementation of adaptive instrumentation control associates a user-specified maximum trace level with each event. Higher trace levels allow greater amounts of event data to be recorded in the trace file; lower trace levels bound the maximum volume of event data to smaller amounts. Each of the three event classes (trace, count, and time interval) has an associated, internal *event threshold*. If the user-specified trace level for an event of a particular class does not fall below the threshold for the class, the event data are recorded in the trace file. Otherwise, the class of the event is reduced (e.g., trace and time interval events are converted to count events). If the event's user-specified trace level is too low even for count events, the Pablo trace capture library will silently record the occurrence of the event, but will not produce a trace file record. By choosing appropriate trace levels, it is possible to completely exclude certain events from the trace file.

[6] The recently announced Paragon XP/S system contains just such support — dedicated performance data recording hardware on each node.

In addition to instrumentation control via user-specified trace levels, the Pablo instrumentation software will dynamically adjust event trace levels within the user-specified range (i.e., the instrumentation software may reduce the trace level below the user-specified point, but it will not increase it beyond that point). Event trace levels vary dynamically based on each event's rate of occurrence. The adaptive instrumentation control software associates user-specified *low water* and *high water* marks with each event. These represent, respectively, the shortest and longest intervals of time between successive occurrences of the same event that will leave the current trace level unchanged. If successive occurrences of a particular event occur in a time interval less than the low water mark, the instrumentation software responds to this high event rate by reducing this event's current trace level. Should this high rate persist, the trace level will eventually drop below one or more of the event trace level thresholds, reducing the volume of recorded event data. Conversely, if successive occurrences of a particular event occur in a time interval greater than the high water mark, the instrumentation software increases the event's current trace level, perhaps increasing the volume of recorded data. It is important to note that the low water mark is a shorter time interval, therefore associated with high event rates and potential reduction of event trace level and recorded data volume. Likewise, the high water mark is a longer time interval, associated with lower event rates and potential increase of event trace level and recorded data volume.

The Pablo trace capture library also monitors the aggregate event rate using a similar algorithm. Together, the event thresholds and the global threshold allow the Pablo instrumentation software to balance event data volume against application perturbation, maximizing the amount of useful trace data.

4 A Pablo Performance Instrumentation Environment Example

Although one can instrument an application source code by manually inserting calls to the Pablo performance data capture library, this process is laborious and error prone if the program is of substantial size. The Pablo graphical instrumentation interface reduces the intellectual cost of source code instrumentation by allowing one to interactively specify the desired instrumentation [36]. These instrumentation directives can be global (e.g., trace calls to all procedures) or selective (e.g., count the calls to this procedure or trace only this specific procedure call).

As noted in §3, the graphical instrumentation interface passes instrumentation specifications to a instrumentation parser that synthesizes instrumented application code. This code can then be compiled, linked with the Pablo trace capture library, and executed to produce a trace file in the Pablo self-documenting data format, described in §5.

To illustrate the operation of the graphical instrumentation interface, called *iPablo*, we describe the steps a user might follow when instrumenting a simple source code.[7] For pedagogic simplicity, consider a simple matrix multiplication code that consists of a main program and two procedures, one that multiplies the two arrays and another that prints the result of the multiplication. Figure 2 shows the *iPablo* instrumentation interface after the user has loaded the file that contains the matrix multiplication code. From top to bottom in Figure 2, there are four components in the *iPablo* interface: the Motif menu bar, the list of instrumentable procedures contained in or called from the currently loaded source code file, the scrollable source code window that highlights the

[7] Space restrictions preclude a complete description of all the instrumentation options supported by the *iPablo* instrumentation interface; for complete details see [36].

instrumentable source code constructs, and the source code line instrumentation buttons at the bottom of the window.

The Motif menu items at the top of Figure 2 allow the user to load a source code file, save an instrumented file, globally apply an instrumentation option to all procedures and/or outer loops found in the code, or apply an instrumentation option to a specific function or procedure. As an example, Figure 3 shows the result of selecting the `printArray` procedure in the scrollable list of routines, and then using the pulldown *Routine* menu to trace call procedure calls in that procedure and count the number of times each outer loop in the procedure is executed. The small icons at the left of the scrollable text window show the effects of this instrumentation; the ++ symbol denotes counting; the arrow symbol denotes tracing.

Finally, one can select and instrument specific lines of code using the buttons at the bottom of the *iPablo* interface. For example, in Figure 3, one of the calls to the `printf()` routine is nested inside two loops; it is doubtful that the volume of data produced by this instrumentation is useful. To disable this instrumentation, one can click the mouse on the `printf()` construct to mark the source code line, then click the *Clear Line* button to remove the instrumentation point.

After selecting the desired instrumentation points, one uses the *File* menu to save the instrumented code. The instrumentation parser then uses the instrumentation specification and its parse tree to create a new file that contains the requisite calls to the Pablo trace capture library. In addition, the instrumentation parser and the *iPablo* interface record the specified instrumentation points, allowing one to later change the instrumentation points or apply the same instrumentation to other files.

5 Pablo Trace Data Format

Many scientific communities have well-established data formats and international databases for storing and distributing information critical to future research. For example, NASA regularly publishes CD-ROM catalogs of guide stars, X-ray sources, and images from its interplanetary probes, together with standard software to manipulate the data. Similarly, the biological community maintains GenBank,[8] a standard, public repository for genome data. Moreover, many scientific journals will not accept papers for publication without concomitant deposit of the raw data (e.g., genetic sequences or bond angles from a crystallography study) that formed the basis for the paper.

Quite clearly, a consistent data format simplifies the dissemination of research results and promotes the development and sharing of tools to manipulate the data. In addition, the adoption of a standard "data language" reduces the time required to exchange and analyze raw data, allowing researchers to devote more time to research.

The Pablo portable trace data format links the Pablo instrumentation software, which captures dynamic performance data, and the Pablo data analysis environment, which provides the tools to reduce and analyze the data. As we shall see, the Pablo goals of portability and extensibility are reflected in the requirements imposed on the Pablo trace record format.

5.1 Trace Format Requirements

Unfortunately, there are no standard data formats, either physical or logical, for processing trace data from sequential or parallel computer systems. Not only does this force each group to develop

[8] GenBank is a registered trademark of the U.S. Department of Health and Human Services.

new trace management tools, it also limits data sharing. The latter is a particularly debilitating because some trace data sets are extraordinarily difficult to obtain (e.g., memory reference traces from parallel systems).

The difficulty in defining a standard medium of data exchange is exacerbated by the diversity of interesting performance data and the plethora of extant and proposed parallel hardware and software architectures. It seems doubtful that one could define a trace format sufficiently general to encompass, for multiple parallel systems, the semantics of virtual memory translation buffer misses, directory-based cache coherence activity, interconnection network contention, memory and instruction traces, operating system activity, and application program control and data flow. As a consequence, one often finds special purpose data formats that exploit the semantics of the data they encode (e.g., by omitting page numbers in virtual memory traces except on page boundary crossings or by using records whose fields require additional, machine specific data to interpret).[9] However, special purpose data formats are not extensible to new trace data types, nor can they be easily ported to new architectures.

Given the problem of defining a general purpose trace format that could represent data from diverse sources, we considered adopting a fixed trace record format that could represent just those events generated by the current implementation of the Pablo instrumentation software. However, a fixed format was not consistent with Pablo's extensibility design goal. It would have made it impossible to add new event types without major software redesign, and it would have limited Pablo's potential applicability to future generations of massively parallel systems. For these reasons, we sought an alternative to the fixed file format, subject to the following constraints.

- Trace files are often quite large, potentially reaching many megabytes or even gigabytes for massively parallel systems — compactness is critical, and data compression should not be precluded.
- Traces may be gathered and then analyzed on machines with different byte order, floating point formats, or word lengths — trace file portability is vital.
- The diversity of software and hardware architectures and the need to correlated performance data across architecture, system software, and application instrumentation levels dictates generality — a diverse and user-extensible set of trace record types must be supported.
- The nature of experimental performance data analysis and the rapid evolution of massively parallel systems precludes predicting what data will be interesting in the future — trace format extensibility is essential.

Recognizing the paramount importance of portability and extensibility, we opted to remove all notion of data semantics from the trace data format, and instead, embed only a description of the structure of the data records in the trace file.

Previously developed self-describing data formats such as the Hierarchical Data Format (HDF) [30] from NCSA and netCDF [33] from UCAR have been very successful. However, the scientific and graphical data sets supported by HDF and netCDF usually consist of a small number of large records, often containing arrays of floating point data. In contrast, a performance trace data file typically contains many event types and thousands of data records of each type. The event records

[9] As an example, the trace format for our earlier instrumentation system [26], could only be interpreted using software-embedded knowledge of the Intel iPSC/2 message passing hardware and operating system software.

are often quite small, only a few tens or hundreds of bytes, with a mixture of integers, floating point values, and character strings; scalars are more common than vectors or arrays.

To accommodate the distinctive nature of performance trace data, we drew on the tested ideas of HDF and netCDF to develop a new, self-describing trace data format. Self-describing data files include a group of record definitions and a subsequent sequence of tagged data records. The tag identifies the type of the record, allowing the data record byte stream to be interpreted using a particular record definition.

5.2 A Self-Describing Trace Data Format

The Pablo Self-Describing Data Format (SDDF) is a trace description language that specifies both the structure of data records and data record instances. Because the format can describe general data records, not just a predefined record set, it is best viewed as a data meta-format. Intuitively, the format supports the definition of records containing scalars and arrays of the base types found in most programming languages (i.e., byte/character, integer, and single and double precision floating point). After evaluating several possible data formats, we concluded that the difficulty in implementing record pointers, self-referential data types, or nested records outweighed the potential benefits. Despite these restrictions, SDDF supports the definition of multi-dimensional arrays whose sizes, but not number of dimensions, can differ in each record instance. At present, we are exploring the possible inclusion of sparse matrix specifications.

A trace file meta-format does not define a fixed set of supported event types, nor does it specify the size, data types or semantics of information associated with a particular event. Instead, when creating a trace file, one is free to decide what is appropriate for the situation and describe event records accordingly. Correspondingly, when analyzing the trace file data one uses the embedded descriptions to interpret the event records.

The desires for compactness and efficient processing are best satisfied by a binary trace format, but byte order and the diversity of number representations make binary files non-portable. ASCII files are portable and human-readable, but are neither compact nor efficiently processed. Because both ASCII and binary versions of a trace format have important benefits, we have developed both binary and ASCII representations of the Pablo self-describing trace data format. The binary format can be used when compactness and speed are of the utmost concern, and the ASCII format provides portability when needed. Simple tools can quickly convert from one representation to the other.

In their most general form, the ASCII and binary versions of the SDDF meta-format describe four classes of records.[10]

- stream attribute — information about the entire trace file
- record descriptor — a template for a data record
- data — a record instance
- command — action to be taken

Stream attribute records contain information pertinent to the entire trace file — for example, the source of the data and the date the trace was generated. Each stream attribute consists of a key and an attribute; both can be any string of characters.

[10] See [2] for complete details.

Descriptor records describe record layouts, similar to a structure definition in the C programming language. Each descriptor record associates a record name with a description of the fields that will appear in all data records having that name. In addition, descriptor records can contain both record and field attributes; their function is similar to that for stream attributes (i.e., to provide descriptive information).

Data records contain the actual event trace information. In the ASCII version of SDDF, a data record is interpreted by matching the record name in the data record with the name of a previously defined descriptor record. In the binary version of SDDF, records are matched to definitions via integer tags. Finally, command records are used internally by the Pablo data analysis software to control the environment's execution, but rarely appear in performance trace files.

Figure 4 shows a sample SDDF file in the ASCII format. This file contains a stream attribute (the trace file generation date), two record descriptors (`message send` and `context switch`), and four data records. The integers "1" and "2" near the `message send` and `context switch` record descriptors are the record tags used to match data records to definitions in the binary version of SDDF. The `message send` field "SourcePE" is a one-dimensional array whose actual size will be specified in each instance of the `message send` data records. Using the record descriptors, the first data record shows that processor 2 context switched to process 23 at time 100.15.

6 Pablo Data Analysis Environment

The Pablo data analysis environment, the final component of the Pablo performance software triumvirate, provides the framework for reducing, analyzing, and presenting performance data. As we discussed in §2, developing a portable, scalable, and extensible performance data analysis environment is non-trivial. Portability mandates not only that the data analysis environment software execute on a variety of hardware platforms, but also that the data reductions be appropriate to the data source (i.e., the parallel system under study). Similarly, scalability dictates that the environment's performance increase with additional processors and larger event data volumes. Finally, extensibility implies that new data analyses can be easily constructed, ensuring applicability to new parallel systems.

To date, most performance analysis environments have been designed to process trace data that contains only a fixed set of trace events captured on a particular parallel system. Even within such restricted domains, most environments are limited in scope and provide only a small set of simple data reductions. In consequence, the initial excitement of users often gives way to frustration as interpreting the performance data rapidly leads to questions that cannot be answered within the scope of the environment's capabilities.

Given the rapid evolution of massively parallel systems, it is increasingly unrealistic to believe that robust performance analysis tools can be cost-effectively developed for each new software and hardware environment. Moreover, the diffusion of intellectual effort across multiple, one-of-a-kind performance tools perpetuates the problem, namely the lack of standard, portable, and extensible performance analysis software.

The twin keys to the Pablo data analysis environment's extensibility and portability are the reliance on a toolkit of data transformation modules capable of processing the self-describing data format and the recognition that software embedded knowledge of parallel system architectures precludes portability. The Pablo data analysis environment supports the graphical interconnection of performance data transformation modules, in the style of aPE [9] or AVS [37], to form a directed,

acyclic data analysis graph. Performance data flows through the configured graph nodes and is transformed to yield the desired performance metrics. The details of file input/output, storage allocation, and module execution scheduling are isolated in the environment infrastructure, making it straightforward to extend the environment by adding new data transformation modules.

This design and its emphasis on the absence of software embedded data semantics reflects our earlier experience [26] that most performance data analyses are simple transformations that do not require knowledge of the data source (e.g., a particular parallel programming model) or its higher level meaning (e.g., this data value is a context switch). For example, an addition module can compute sums without regard to the meaning of the data; the values need only be mathematically compatible. Selecting what to sum and deciding if it is semantically meaningful is a user *configuration* choice, not a software constraint. For this reason, the majority of the Pablo analysis analysis environment's data transformation and presentation modules are designed to process data values without knowledge of the underlying data semantics.

By graphically connecting analysis and data display modules to form a directed acyclic graph and then interactively selecting which trace data records should be processed by each data analysis module, the user specifies the desired data transformations and presentations. All data passed among modules in the configured data analysis graph is encapsulated in self-describing data format records. The environment infrastructure uses the record and field names from the record descriptors, described in §5.2, to prompt the user as individual modules are configured. When the analysis environment begins execution, the environment infrastructure automatically extracts the necessary fields from data records and passes those values to the modules for processing.

Simply put, the semantics of the data analysis are embedded in the user's graphical configuration, not the data analysis software.[11] The importance of the self-describing data format should now be clear. By isolating all higher level semantics in the graphical configuration, both the data format and the data analysis software are independent of specific hardware or software architectures. This allows the Pablo environment to analyze performance data from diverse sources, ensuring portability. Similarly, the graphical programming model allows users to develop new data analyses by constructing data analysis graphs whose semantics reflect the desired transformation. Finally, the ability to augment the suite of data analysis modules ensures extensibility to new parallel systems.

We expect the Pablo performance data analysis environment to serve three user classes: novice, intermediate, and expert. Although each has differing needs, we believe the ability to save and load preconfigured data analysis graphs, the ability to graphically create new graphs, and the ability to develop and add new data analysis modules to the Pablo performance environment's palette, provide the ease of use and extensibility to satisfy the needs of all three user classes.

Novice users are most likely to view Pablo's performance analysis environment as supporting a fixed set of data transformations and presentations. This functionality is defined by a small set of data analysis graphs that have been developed by either the parallel system vendor or by other software support staff at the user's institution. This class of users will load a previously configured data analysis graph that realizes the desired data transformation, specify a trace data file, and study the results of the data analysis. They need not know that Pablo supports graphical module configuration.

In contrast, intermediate users often wish to conduct performance experiments, asking and exploring the answers to "what if" questions. Simply put, this user class needs an environment toolkit whose components can be assembled in a wide variety of ways. The graphical programming

[11] See §8 for a detailed example that illustrates the construction and configuration of a Pablo data analysis graph.

model allows users in this class to quickly experiment with different data analysis and presentation modules — no software development is required. Instead, a suite of data analysis graphs can be refined over time as the user decides that particular data analysis modules are no longer necessary or that other modules would be more beneficial. These users can tailor the environment to their individual needs with no knowledge of the system internals and no coding effort.

Clearly, Pablo's standard set of data analysis modules will not satisfy the requirements of all users. Expert users, who are intimately acquainted with the parallel system architecture and system software, will need the broadest latitude. Not only will they wish to reassemble the existing components of the environment toolkit, they will want to add new components to the toolkit. As an example, there are some data transformations that *do* require knowledge of trace data semantics; these transformations are meaningful only when analyzing data drawn from a particular parallel architecture. Expert users will develop software to realize these transformations, and they will expect these added components to integrate seamlessly with extant data analysis modules. The Pablo data flow metaphor makes this integration possible.

6.1 Graphical Programming Models

The graphical programming paradigm has been widely used in both academic and commercial settings to create coarse-grained data flow graphs. Couch's Seecube performance environment [6], the aPE visualization system [9], originally developed at the Ohio State Supercomputer Center, and the University of New Mexico Khoros image processing system [40] are well-known research implementations of such systems. The AVS visualization system [37] and Silicon Graphics' Explorer system are successful examples in the commercial arena. The wide acceptance of these systems suggests that users find the graphical construction of coarse-grained data flow graphs intuitive; the graphical interface provides the desired flexibility without introducing overwhelming complexity.

The Pablo data analysis environment is written in C++and uses the X window system and Motif toolkit as its user interface. This ensures a standard look-and-feel, and the widespread acceptance of X and Motif guarantees the environment will be portable to a wide variety of machines. The primary difference between Pablo's data analysis environment and other academic and commercial implementations of the graphical programming model is a consequence of the problem domain. Typically, a scientific visualization involves a relatively small number of large data structures (e.g., large, two or three-dimensional arrays) consisting of either measured or computed data. In contrast, performance data often involves many millions of trace data records of many types, each only a few tens or hundreds of bytes in size. Reflecting this, the AVS, Explorer, and aPE data flow graphs are very coarse-grained with a modest number of graph module input data types, whereas Pablo's data flow graphs are very fine-grained with modules capable of accepting diverse data types. This seemingly small difference substantially changes the underlying implementation of inter-module communication, data buffering, and scheduling.

6.2 Data Analysis Graph Components

A Pablo data analysis graph consists of two primary components: a collection of graph nodes and their associated data transfer conduits. For descriptive simplicity, these are often referred to as modules and pipes, respectively. There are four primary module types, reflecting both function and typical location in the graph: data analysis, data presentation, trace file input, and trace file output.

Because a Pablo data analysis graph is directed and acyclic, file input and output modules most often lie, respectively, at the root(s) and leaves(s) of the graph; these modules read and write files in the Pablo self-describing data format. Moreover, all information that flows through the pipes connecting modules is encapsulated in self-describing data format records. This allows one to tap the data stream between any pair of modules and write the data to a file via a file output module. The resulting file, in the self-describing data format, can be subsequently processed by another data analysis graph.

As the name suggests, data analysis modules reduce or transform performance data as it flows through the graph; modules of this type typically lie in the interior of the graph. Finally, data presentation modules, which include mappings of data to both dynamic graphics and sound, most often form the leaves of a data analysis graph, though they can also be used to display data produced by interior graph nodes.

In a typical Pablo data analysis graph, one or more trace file input modules read files written in the self-describing data format and send a sequence of records through their output pipes to descendent modules. A given module may have more than one output pipe, but the data on all of those output pipes will be identical (i.e., the module's output is replicated on all of its output pipes). Interior graph nodes have both input and output pipes, but only data analysis modules can produce *new* data records on their output pipes. Both presentation and analysis modules can forward *copies* of records that appear on their input pipes. Obviously, nodes at the leaves of the graph have only input pipes. These can be either trace file output modules that write their input to a data file in the self-describing data format, or data presentation modules that update graphic or sonic displays.

Figure 5 illustrates the internal features of a data analysis module and its interactions with the data transfer conduits (i.e., pipes) and the data configuration interface.[12] The heart of a data analysis module is a *functional unit* — it implements a particular data transformation (e.g., addition or scaling) without embedded software knowledge of the input data's semantic interpretation. Each functional unit must provide a standard set of interfaces to the analysis module's *data wrapper*. The latter encapsulates each functional unit and interacts with the data configuration interface during user configuration of the data analysis graph, and with the data conduits during module execution. Via these standard interfaces, the data wrapper and configuration editor can determine, for example, that an addition functional unit accepts two values as inputs and produces one result. The addition functional unit may further advertize the valid combinations of input data types and, based on those input types, the resulting output data type.

This isolation of system infrastructure from the functional unit's data transformation simplifies the creation of new data analysis modules. The developer need only write the code to perform the desired data transformation and advertize the types of data the functional unit will accept as input and produce as output. During execution, the Pablo environment infrastructure manages the scheduling of module execution, extracts input data from the input pipe(s), passes this data to the functional unit, and writes the functional unit's output data to the module's output pipe(s). With this background, we can now describe the steps required to create a new data analysis graph and the underlying interactions of the environment's software components.

Users construct new data analysis graphs in three phases: module selection, module intercon-

[12] This discussion focuses on the operation of a data analysis module. File input/output and presentation modules are special cases.

nection, and module configuration.[13] The first two of these, module selection and connection, are trivial; users graphically select a collection of modules from a palette that includes data analysis, data presentation, and file input/output modules. As each module is selected, an instance of that module becomes visible on the graph canvas. Given the desired module set, the user then interconnects the modules by graphically specifying a series of edges in the directed graph.

The final phase, module configuration, is the most complex and provides Pablo's independence from specific data semantics and trace data record formats. First, the Pablo environment software infrastructure topologically sorts the graph to identify the file input modules at the graph roots. The descriptor records from the self-describing trace data input files are then propagated to the modules at the next level in the graph. Using these descriptors, the data wrapper for each module now knows what types of data records can be expected on each of the module's input pipes. By querying the functional unit, the data wrapper also knows the number and types of inputs expected by the encapsulated functional unit. Using this information, the data configuration editor presents the user with a list of record types that may appear on each of the module's input pipes. For each record type, the user may elect to discard all instances of the record, copy all record instances to the module's output pipe unchanged, or extract one or more fields from record instances for input to the module's functional unit. Using the type information advertized by the functional unit, the data wrapper verifies that the fields the user has elected to process are of acceptable types.

The data wrapper also allows the user to construct a new descriptor record containing one or more fields for the functional unit's output(s); during execution, the analysis module will produce data records of this type. The new output record also can contain data from the records whose fields formed the input to the functional unit; selected fields from the records used as module input are copied into the new module output records. Finally, before configuration continues for modules at the next level in the graph, the module's data wrapper writes to the module's output pipes the descriptor records for both the newly created output record type and each record type that will be copied through unchanged.

The importance of both topologically sorting the graph before configuration and the interactive configuration process should now be clear. Each module can accept as input only those records propagated by modules at the level above. These records include both records that were copied through unchanged *and* those new records created by the modules above. Because modules are configured based on the record types available on their input pipes, configuration must begin at the graph roots.

Interactive configuration and the data wrapper's isolation of record structure from functional unit inputs allows the Pablo performance environment's data analysis and presentation modules to process any trace file that was written using the self-describing data format. With interactive selection and binding of record fields to module inputs, the scope of possible data reductions is limited only by the palette of available analysis modules and the user's facility in combining the modules.

Once all modules have been configured, the environment is ready to execute the data analysis graph. Data records from the file input modules flow through the configured graph and are processed by the data wrappers in accordance with the user's previous requests. At each module, some data records may be discarded, others may be copied through the module unchanged, and some will be selected for input to the module's functional unit.

[13] These phases are illustrated in §8.

Normally, the data wrapper requires new data values for all functional unit inputs before invoking the functional unit. However, the user can configure a module to execute asynchronously (i.e., whenever a new data value is available for any of the functional unit's inputs). When the functional unit executes, the data wrapper uses the functional unit's output to synthesize an instance of the previously specified output record and transmits the record to the output pipe.

6.3 Data Binding and Module Execution Rules

In contrast to simple, static data flow graphs whose nodes have fixed size input data buffers, strongly typed inputs, and simple firing rules [8], Pablo supports data buffering on module input and output pipes, polymorphic data analysis modules, complex mappings of data records to module inputs and outputs, and both synchronous and asynchronous module execution. Although we believe the Pablo environment's greater power more than offsets its correspondingly greater configuration complexity, one must carefully balance generality, complexity, and ease of use.

The Pablo data analysis environment has a well-defined set of rules governing data binding and module execution that have evolved as we have tested successive versions of the software. Some of these rules have been implied in §6.2, but it is instructive to examine each rule and the reasons for its existence. Most rules are derived from the fundamental assumption that each module's data wrapper must be capable of acting on each data record as it arrives. Although the rules may seem excessively complex, our experience has shown that their implementation creates an intuitive interface for graph configuration.

- Each module input port is associated with one and only one input data pipe. Although multiple record types from a single input pipe may satisfy a particular input port, each time a new data record arrives, the module's data wrapper can immediately determine if the record satisfies any of the functional unit's input ports.

- If an analysis module produces an output record, each field in the output record that is not filled by a result of the functional unit must be bound to one and only one input pipe taken from the set of pipes bound to the input ports of the module. This restriction is necessary to limit the number of module output record types, and it prevents deadlock.

- There is at most one output record type per analysis module. Although not strictly necessary, this greatly simplifies interactive configuration. Without this restriction, it is possible to create many output record types by mapping fields from input records to output records.

- When fields from multiple input records are bound to the same input port or output record field, the data type of the first field bound will determine the valid data types for subsequent bindings. Intuitively, if one chooses a vector from one input record as an input to an addition module, any fields chosen from other input records must also be vectors.

- Any record selected from an input pipe must satisfy all the input ports and output record fields that are associated with that pipe. This ensures that either all ports associated with a particular input pipe are satisfied by a particular record or none are.

- A module executes synchronously by default. Asynchronous mode must be selected explicitly.

- In synchronous execution mode, a module is eligible for execution only when there is new data on all its input ports.

- In asynchronous mode, a module executes when there is new data on at least one input port, and there are valid data on all other input ports. Data on an input port is valid until a record appears on the associated input pipe that is discarded or copied through. The module's functional unit can query an input port to determine if the data is new or simply valid. This rule ensures that modules whose input pipes fill at different rates can continue to execute and process the incoming data.

6.4 Analysis Module Palette

The Pablo data analysis environment's flexibility rests not only on the self-describing data format and the environment infrastructure, but also on the palette of data analysis and presentation modules. Clearly, a basic set of data analysis and presentation modules should include those that will be universally useful and that can be combined in the largest number of ways. However, the basic set should not be so primitive that users must constantly assemble multiple modules to realize simple, frequently needed data transformations or presentations. Hence, when choosing an appropriate set of data analysis and presentation modules one must balance generality, specificity, and convenience. The current module set reflects our experience to date as users of the Pablo environment.

Because portability is one of the guiding design principles of the Pablo performance analysis environment, the basic set of data analysis modules process performance data without software embedded knowledge of higher level data semantics. All data analysis modules in the basic set are polymorphic — they process all mathematically valid combinations of data inputs. For example, the addition module accepts scalars, vectors, and higher-dimensional arrays whose elements can be integers, single, or double precision floating values. If the data inputs are mathematically conformable, the addition module will produce the appropriate output, promoting data types where necessary.

The basic module set includes simple mathematical transforms such as counts, sums, products, differences, ratios, maxima, minima, averages, absolute values, powers, logarithms, trigonometric functions, and simple statistics. In addition, a set of synthesis modules allow one to construct vectors and arrays from scalar input data. We plan to add simple data filtering transforms (e.g., sorting, ranking, linear interpolation, extraction of sub-vectors or sub-arrays, scaling, histogramming, and curve fitting) and other statistical transforms (.e.g., skew, kurtosis, median, and mode). We also plan to develop a richer set of modules that embody common performance analyses (e.g., utilizations, critical paths, and profiles. Finally, we expect, and hope, that additional, more sophisticated data analysis modules will be developed and shared by the Pablo data analysis environment's user community.

As noted earlier, there are some data transformations that are most simply realized using knowledge of trace data semantics or knowledge of the data source (e.g., the characteristics of a specific parallel architecture). These modules can be integrated seamlessly with the basic module set, albeit with consequent loss of environment portability and generality.

Finally, the Pablo environment supports two types of data presentation modules: graphical and sonic. Given the relative novelty of sonic presentation modules, these are described separately in §7. As with the basic set of data analysis modules, the design emphasis is simplicity and portability. The graphical displays are polymorphic and present data without regard to its source. Thus, for example, one can create individual bar graphs as well as vectors and two-dimensional arrays of bar

graphs. In addition, each display module allows the user to scale incoming data as needed and to specify display options such as labels and tick marks.

The graphical displays are based on the X window system and require only standard, 8-bit color graphics. Currently supported graphical displays include, bar graphs, bubble charts, strip charts, contour plots, dials (with and without history), interval plots, kiviat diagrams, LEDs (discrete bar graphs), X-Y line/scatter plots, matrix displays, pie charts, polar plots, and a 3-dimensional scatter plot. As with the data analysis modules, we expect that the Pablo user community will augment the basic display set with new displays.

By restricting the basic display set to those graphics that are realizable on entry-level color workstations, we have sought to maximize the potential user base. Though the value of high-performance color graphics is undeniable. the expected user base of the Pablo environment includes working scientists, engineers, and system software developers who may not have access to high-performance graphics displays.[14]

6.5 Parallel Execution and Display Control

At present, the data processing and presentation rate of the Pablo data analysis environment is limited only by the speed of the underlying processor. Because the graphical displays are updated each time new data arrives on their input ports, the display update rate is a function of data analysis graph complexity. Simple graphs execute quickly, and the displays are updated rapidly. In contrast, complex graphs execute more slowly, and the display rate slows commensurately. For simple data analysis graphs, users often wish to slow the display update rate — the rate exceeds their perceptual acuity. We plan to add display update controls, allowing users to throttle the graph execution rate to perceptually manageable levels.

In addition to the actual display update rate, Heath's experience with the ParaGraph performance visualization system [14] and our own experience [26] suggest that users also wish to view periodic performance "snapshots" (i.e., they wish to see state of the graphical displays only at discrete, often widely separated, points in the trace data). Here, the display update is logical, rather than physical, and is simply realized by modifying the display module data wrappers to process every N-th data record.

Because a display snapshot represents a single point in the trace data, users frequently want to re-examine earlier display states. Hence, we plan to implement analysis graph checkpoints; the checkpoint frequency would be a user configuration option. By selecting a particular checkpoint, the user could replay the data analysis from that point, allowing him or her to compare display states.

Parallel module execution is another, planned environment infrastructure extension. Because all data analysis graph modules communicate solely via message passing, individual modules can potentially execute in parallel, allowing the construction of *scalable* data analysis graphs. The goal of supporting parallel graph execution, via both lightweight tasks and explicit message passing, has influenced all phases of the environment's infrastructure design, and it is a metric against which all design decisions have been measured. The use of data conduits to propagate record descriptors and record data between modules disperses information to each module rather than relying on a global name space. Each module receives all the data needed to execute from its antecedent modules. Moreover, the binding and configuration rules ensure that each module can immediately process

[14] We will return to issue of high-performance, graphical data presentation in §10.

data records as they are received; this limits data buffering and enforces regular data flow. The small size and large number of typical trace data records makes pipelining effective and speedup likely. Finally, the object-oriented implementation in C++ admits replacement of the data conduits' current implementation using FIFO buffers with a message-based implementation on a distributed memory parallel system.

7 Pablo Data Sonification

The use of scientific data visualization to provide new insights is well established. However, visualization is but one way of presenting data — psychological studies show that individuals respond more quickly to audio cues in certain instances. Just as visual elements (e.g., color, form, and line) are combined to present and analyze data visually, the elements of sound (e.g., duration, pitch, volume, timbre, and spatial location) can be combined to present data aurally. In short, sound represents a rich communication channel that has heretofore received little attention as a medium for scientific data presentation.

If used correctly, sound can emphasize data characteristics not easily seen, much in the same way a movie sound track conveys information complementary to the imagery. For example, we immediately recognize that the sound of footsteps growing louder means someone is approaching, even if the individual remains outside our field of view. A careful listener might even be able to deduce an individual's appearance or identity from secondary cues (e.g., the tapping of shoes or the rustling of clothing).

Experiments with sonic data presentation, often called sonification, have been underway since the early 1980's, when inexpensive technology made applied research in this area economical [5].[15] However, the sonifications have often been limited to simple mappings between data values and primitive sound characteristics, such as volume, pitch, and wave shape. Although such simple mappings can represent discrete data values as a sequence of isolated sounds, there is no sonic equivalent of a pie chart, bar graph, or line plot.

Just as scientific visualization initially drew on the experience of the graphics community before developing it's own language and representations, sonification has drawn on the experience of the computer music community. However, like visualization and graphics, sonification and music differ in important ways; sonification idioms are only beginning to emerge. As examples, Scaletti [35], Rabenhorst *et al* [31], and Francioni [11] have begun exploring the use of new idioms and abstractions for mapping data to sound. Francioni's work [11] is most relevant to our research goals, as it concerns the mapping of parallel computer system performance data to synthesized sound.

Much of sonification's embryonic state is attributable to the lack of software standards and common hardware interfaces (e.g., there is no sonic equivalent of the X display protocol, nor do all workstations include similar sound hardware). Ideally, we would like to accompany visualizations with real-time output of high quality digital sound. However, general purpose digital signal processing hardware, while extremely flexible, is also expensive. At the other extreme, the audio chip in the Sun SparcStation is a ubiquitous hardware platform; unfortunately, its sound fidelity is very low.

A better alternative is instruments that conform to the Musical Interface Digital Interface

[15] The computer music community has been exploring the mapping of data to sounds for over thirty years.

(MIDI) [27]. Using a variety of built-in synthesis techniques, most MIDI instruments can play notes in many voices (timbres), ranging from traditional voices like a piano or trumpet to novel sound effects. MIDI commands are just a few bytes long, so it is easy to control output in real-time. A wide range of sound mappings can be efficiently produced by connecting one or more of these instruments to a workstation. However, MIDI is a very general protocol, and most manufacturers use "system exclusive," or device-dependent messages to control the unique features of their instruments (e.g., the ability to create new voices). Furthermore, different instruments can interpret MIDI messages in different ways. This makes it almost impossible to play a composition written for one MIDI instrument on another.

In light of these software and hardware constraints, we have identified the following criteria that we believe a portable sonification system should satisfy [20, 21, 22]:

- Sound hardware should be manipulated by a network server, so that the interface to local and remote sound devices is the same.
- The interface to a particular sound device cannot be too abstract — it must reflect hardware functions. Sound devices vary greatly in functionality, and attempts to create a single set of commands for all types of sonic hardware are bound to be restrictive or vague.
- The core of a sonification should be both configurable and able to utilize all available capabilities of a sound device.
- It should be easy to create a wide range of sonifications that operate in real-time and that can be synchronized with visualizations.

7.1 Sonification Software Structure

The Pablo sound toolkit [20] is an attempt to ensure portability across multiple sound devices by separating the sound hardware interface from the sonification (i.e., the algorithm that creates sound from the data on wishes to sonify). The design of the sound toolkit reflects the success of the X window system's client/server protocol. Each sound device is managed by its own network server, which accepts *sonic messages* that cause sounds to be played. An application program opens a connection to a sound server, which may be remote, receives a list of supported sonic messages (e.g., play note, vary pitch, or select synthesized instrument), and uses parameterized variants of those messages thereafter to interact with the sound server.

Mapping data to sound involves sending a stream of sonic messages to one or more sound servers for each scalar or vector data value one wishes to sonify. For example, a simple sonification for a distributed memory parallel system might map a stream of X-Y pairs representing processor number and message transmission, either send or receive, to musical notes as follows: for each input X-Y pair, the processor number is mapped to pitch and message type is mapped to locations, with message sends in the left speaker and message receives in the right speaker.

Our current implementation supports both the sampled sound capability of the Sun SparcStation audio port and a variety of MIDI synthesizers. Within the constraints of common functionality, it is possible to switch between these sound devices solely by connecting to the appropriate sound server; no software modifications are necessary.

Creating sonifications is simplified by four major abstractions: *sound control files*, *transformation functions*, *sonic widgets* and *widget control files*. Sound control files are scripts written for a

particular sound device that list sonic messages to be sent to that device's server. Transformation functions, which provide simple functional modifications of input data values, are used to choose between sound control files or to vary the parameters of individual sonic messages. Sonic widgets manipulate one or more sound control files and an optional transformation function to create a specific sonification. For example, a sonic widget might use a data range transformation function to scale its input data to specified range and use a boolean transformation function to send its sound control file (e.g., to play a note) to the sound server only if the transformed datum exceeded a particular threshold. Finally, sonic widgets can write complete configuration information in widget control files that can be saved, edited, and later reloaded.

7.2 Sonification Experiences

We have integrated the sound toolkit with the Pablo performance analysis environment and have developed several mappings of performance data from distributed memory parallel systems to sound. Some of these mappings complement existing graphic displays by offering redundant information, others present information not easily displayed graphically. Simple examples for sonifying message transmissions include:

- mapping the node number of a message transmitter to a note of fixed duration (i.e., playing a note of a particular pitch when each processor sends a message),
- mapping the node number of a message transmitter to a note of fixed duration in the left speaker and, when the message is received, the node number of the message receiver to another note in the right speaker, and
- mapping the node numbers of message transmitters to pitch, where a note begins when a message is sent and continues until the message is received.

Although these examples are seemingly minor variations on a theme, they convey surprisingly different perspectives on the time varying data. On hearing the simplest example, mapping message transmitters to notes, one can detect not only the frequency of transmissions but also the number of communicating processors and, with knowledge of the mapping function, their physical separation in a distributed memory parallel system. In addition, performance data traces taken from different application programs reveal strikingly different communication details when sonified. The second example reveals details about temporal communication activity that could not be easily deduced from concurrent visualization, namely that messages often occur in tightly clustered groups; staccato bursts of sound make this immediately obvious. The final example reveals not only the expected delay between message transmission and receipt but also the distribution of latencies.[16]

We have also developed a set of "earcons," [4] or audio warnings, that presently include sampled voice warnings, enumerations, and alarms. As an example, we mapped aggregate processor utilization to a sonic alarm that sounded only when utilization fell below a specified threshold. We have found this particularly useful when processing large volumes of data. Although users are unable to maintain attention focus on graphic displays for long periods of time, a sonic alarm, based on either a tone or a spoken warning, can quickly alert a user to important behaviors.

[16] One can also quickly detect message transmissions without corresponding receipt. We found a data recording error in a trace file that multiple years of visualization had not revealed.

Based on our early experience, we believe data sonification has great promise, particularly when used in concert with graphical displays. However, many open questions remain concerning the choice of appropriate data to sound mappings, the importance of spatial audio cues (e.g., stereo balance), and appropriate combinations of graphics and sound. The flexibility of the Pablo sonification toolkit provides users with a testbed for experiments with sonification. Over time, we hope that these experiments will identify the most common and useful sonifications.

8 A Pablo Performance Analysis Environment Example

When using the Pablo performance analysis environment to process the trace data contained in a self-describing trace file, one must either load a previously configured data analysis graph that realizes the needed data transformations and presentation or interactively construct a new graph with the desired features. To illustrate the use of a data analysis graph and to explicate its construction, we describe a subset of the steps a user might follow when creating and configuring a simple graph.

Figure 6 shows an intermediate stage in the construction of a graph to process performance data taken from the execution of an iterative partial differential equations solver on a 32 node Intel iPSC/860 hypercube.[17] The pulldown menus at the top of the main Pablo environment window allow one to load a previously configured graph, to initiate or halt execution of a graph, to configure a newly connected graph, to add modules to a graph, and to obtain help on the environment's functionality.

By selecting the *Module* menu, one can instantiate the *Pablo Module Creation* dialogue shown in Figure 6. Each scroll box in the dialogue contains a group of related modules: file input/output, mathematical transforms, vector and array synthesis, graphical displays, and sonifications.

Module instances are created by clicking the mouse on a module's name, perhaps specifying a new name for the module in the *Module name* text box, and clicking the mouse on the *Add* button at the bottom of the *Pablo Module Creation* dialogue. Each module instance appears as an oval icon on the primary Pablo window.

Typically, file input modules form the roots of the directed analysis graph, display and sound modules form the graph leaves, and both mathematical and synthesis modules form the interior graph nodes. The topology of the graph can be specified incrementally during module selection, or after all the needed modules have been instantiated. In either case, the *Module* pulldown menu allows one to enter graph connection mode. In this mode, clicking the mouse on a source and then a destination module will create a graph arc between the pair. In Figure 6, several modules have already been instantiated and connected.

After all the nodes and arcs of the data analysis graph have been specified, the final configuration phase can begin. The *Configure* pulldown menu allows users to interactively specify which trace file records and record fields should be processed by each module of the graph. The environment infrastructure first topologically sorts the graph and prompts the user to configure modules at successive graph levels, beginning at the graph roots.

In the example graph of Figure 6, the graph root is the *FileInput-1* module, a file input module that will read data from a file whose records written in the self-describing data format. To

[17] The data file whose processing is illustrated in Figures 6–8 was captured by specifying instrumentation points via the Pablo graphical instrumentation interface, and then compiling and linking the instrumented code with the Pablo trace capture library.

configure this module, the user specifies, via a popup file selector dialogue, which trace file should be read as input. In this example, the descriptor records are then read by the *FileInput-1* input module and propagated to the graph nodes at the level below (i.e., the modules *SynthesizeVector-1*, *SynthesizeArrayElement-2*, *SynthesizeArrayElement-1*, *SynthesizeArray-1*, *ProcedureDuration*, and *SendSound*). For each of these modules, the data configuration editor presents the user with a popup *Bind Records* dialogue similar to that shown in Figure 7. Via this dialogue, the user must specify which records are to be discarded at this module, copied through to modules below, or processed here. In addition, if the module type is data analysis (i.e., not a file input/output, display or sound module), the user must also specify the components of the module's output record.

In this example, the *ProcedureDuration* module is a direct descendent of the file input module. The *Status Board* dialogue of Figure 7 shows that the inputs to the *ProcedureDuration* module can be either a user-defined value, or data from the *FileInput-1* module. A color code in the dialogue, shown by the dark perimeter of two of the icons, indicates that the user had earlier specified that the inputs of the *ProcedureDuration* module are to be satisfied by the *FileInput-1* module.

The *Bind Records* dialogue in Figure 7 contains three boxes, labeled *Input Pipes*, *Records*, and *Fields*. The center box shows the data record types contained in the input stream;[18] here, the user has selected the *Exclusive Duration*[19] field from the *Procedure Exit Trace* record and bound it to the input port of the strip chart module named *ProcedureDuration* by clicking the mouse on the record field name and then clicking again on a module input port. Later, during graph execution, this field will be automatically extracted from each instance of a *Procedure Exit Trace* record and passed to the input of the *ProcedureDuration* strip chart module.

The *Configuration Board Detail* dialogue in Figure 7, which can be created by clicking the mouse on any icon in the *Status Board* dialogue, shows the status of selected the input pipe, input port, or output port. In the figure, the detail dialogue indicates that the *ProcedureDuration* module's input has been satisfied by the *ExclusiveDuration* field, and the input port's type has been constrained to be an integer scalar (i.e., an integer with dimension zero).

Because the *ProcedureDuration* module is a data display module (i.e., it produces no new data as output), it has no output ports. However, if it were a data analysis module, one would also need to construct a descriptor for the records that would that contain the module's output during execution. This output record specification process, not shown Figure 7, would create a new record type that could be processed by the descendents of the module.

When the configuration of a module is complete, the configuration process continues first with other modules at this graph level and then with other modules at the next level. In each case, the user must specify the disposition of incoming data records, extract record fields for processing by the current module, and construct a new record if the current module produces output. The final result of the configuration process is an executable data analysis graph. Lest this process seem excessively tedious, recall that configured graphs can be used to process many trace files, making graph configuration much less frequent that graph execution.

Now let us consider the semantics of the configured graph in Figure 8. The leftmost path in the graph uses a sequence of procedure exit records to synthesize a vector of processor busy times. Each element of the array corresponds to a particular processor, and the element's value is the sum of all exclusive procedure durations on this processor (i.e., is the total computation time on each

[18] If the record types in this trace file were identical to those shown in Figure 4, the displayed record names would be `message send` and `context switch`.

[19] The exclusive duration is the invocation lifetime of a procedure, exclusive of any procedures that it might call.

node). The binary math module then divides each element of the vector by the current time, and the result is displayed in the processor utilization kiviat display at the lower left of Figure 8. Notice that there is substantial disparity in the utilizations of the thirty-two processors.

Proceeding from the left, the next two paths in the graph of Figure 8 compute and display the sequence of procedure lifetimes and the number of times each procedure was called on each processor, respectively. To realize these two displays, the synthesize array element modules construct arrays of procedure lifetimes and counts of procedure calls that are then passed to the two matrix displays. The color code, with blue representing small values and red large values, shows that although the procedures are called an equal number of times by each processor, the procedure execution times are quite different on each processor.

The next path, terminating in the *MessageVolume* module, computes the total number of bytes sent between each pair of processors. The communication pattern shown by the contour plot reflects two types of communication in this trace file: nearest neighbor communication, shown by the clusters near the diagonal, and a logarithmic condensation algorithm used to compute a global minimum, shown by the contours off the diagonal.

The *SendSound* module maps the processor number of each message sender to a musical note; configuration options allow one to specific the audio device type, the musical instrument, and the range of notes.

Finally, the rightmost path through the data analysis graph is the *ProcedureDuration* module whose configuration was described above; it simply shows the exclusive duration of each procedure call. The repeating pattern is caused by the iterative nature of the application program.

Polymorphism is a key feature of all data analysis and presentation modules. For example, the single binary mathematics module can compute the four basic binary operations (i.e., addition, subtraction, multiplication, and division) on all mathematically valid combinations of data inputs. In this example, the two inputs to the *BinaryMath-1* module are a vector of cumulative procedure lifetimes and a scalar, the current time. Though not illustrated by this example, most display modules also accept data of multiple types (e.g., the matrix module can display vectors or or two-dimensional arrays).

9 Potential Pablo Applications

In §8, we sketched how one uses the Pablo data analysis environment to construct, configure, and execute a data analysis graph. Because the environment is capable of processing any data specified in the self-describing data format,[20] the Pablo software instrumentation need not be the source of the data nor must data analyses be limited to application program behavior. Any data that can be translated to the self-describing data format can be analyzed. Thus, one could use the Pablo performance analysis environment to explore the trace data drawn from a simulated computer architecture or to study the performance of operating system resource management policies on massively parallel systems.

As an example of the latter, consider the recently announced Intel Paragon XP/S multicomputer [17]. Each node will execute a Mach microkernel with an OSF/1 application interface on individual multicomputer nodes. As announced, the Paragon XP/S will support virtual memory, parallel disk arrays, and high-speed communication links; a decidedly more complex system soft-

[20] Subject, of course, to the availability of an appropriate set of data analysis and presentation modules.

ware environment than has been typical of distributed memory parallel systems. Unfortunately, little is known about effective resource management for massively parallel systems, particularly in the context of a general purpose operating system.

As a result of our earlier work [25] and that by NIST [28], the Paragon XP/S system contains hardware and software support for the real-time capture and extraction of performance data. The marriage of the Paragon XP/S hardware support for unobtrusive performance monitoring with the data analysis capabilities of the Pablo performance analysis environment would permit a direct analysis of individual operating system modules and their interactions.

By inserting appropriate software instrumentation in portions of the operating system, it will be possible to monitor the dynamics of multicomputer resource management and to study the effects of specific resource management algorithms. Ideally, one would approach the problem by first capturing and analyzing performance data that describes the operating system's basic behavior, then incrementally modify the software to implement variants of promising resource management policies and "close the loop" by studying the effects of those changes on the system software using the Pablo analysis environment.

10 Performance Data Immersion

Although Pablo provides extensive support for data analysis and both graphical and sonic data presentation, the human-computer interface remains bandwidth limited. On a distributed memory parallel system with hundreds or thousands of processors, the size of an event trace file can quickly reach many gigabytes [32]. To gain insight from this data and to tune both system and application software, the data must be presented in ways that not only show trends but also allow detailed exploration of small scale behavior.

New transducer technologies now make it possible both to sonically and visually *immerse* a user in data. From a sensory perspective, the computer need not be simply the medium of interaction between user and data; instead, users can interact *directly* with the data (e.g., moving their head to change their visual field of view, using their hand to rotate or transform the data, or changing their perspective by assuming the role of a system component). Data immersion provides a reality perceived by the senses, albeit a simulated or "virtual reality." The user is no longer an external observer of data, but rather an active participant with the data.[21]

One possible option is to immerse users in a virtual world based on dynamic data that describes the behavior of massively parallel computer systems. Through a head-mounted display, three-dimensional sound cues, and direct data manipulation, the user would be able to explore, viscerally experience, and modify the dynamic behavior of a high-performance multicomputer.

We are now extending the Pablo data analysis environment infrastructure to include a set of virtual reality presentation modules (VRPMs) that realize multiple virtual worlds based on the time varying performance data from parallel computer systems. After connecting a VRPM to a Pablo data analysis graph, the user will be able to don a head-mounted display and data glove and interact with the transformed performance data, rather than simply seeing and hearing graphical and sonic displays.

If the virtual environment were causally tied to the data source, one could explore the feasibility of real-time, adaptive system control. For example, one might observe access patterns to secondary

[21] By analogy, watching a bicycle rider is fundamentally different from riding a bicycle.

storage devices, detect a performance bottleneck, stretch out one's hand to a storage device, and move a file to another device. With appropriate interfaces, the operating system would then move the file, and the effect of this change would be visible in the virtual world.

Clearly, virtual reality's greatest strength is its support for a qualitative perspective — insight; the Pablo performance analysis environment provides quantitative information via its detailed displays. Joining the two would open unexplored research territory and potentially provide the best of both.

11 Data Parallel Programming and Performance Analysis

As described earlier, the Pablo performance environment's design provides the mechanism to capture, analyze, and graphically present dynamic performance data from the execution of programs on scalable parallel computer systems. The Pablo instrumentation software generates performance data by modifying application source code to include calls to a trace capture library. The Pablo environment most successfully relates dynamic performance data to the application program's control flow when the program structure is not substantially altered by the compiler (i.e., when the parallelism is explicit in the application code).

Although this explicit parallelism makes correlation of dynamic performance data with program behavior straightforward, it does nothing to lessen the already difficult problem of programming parallel systems. In contrast, data parallel languages like Fortran 90 or its proposed extensions, Fortran D [15] and High Performance Fortran [16], not only provide many of the programming abstractions needed to elide the details of data placement and movement on distributed memory parallel systems, they also provide sufficient control to tailor data placement to match application behavior.

By hiding many of the details of both parallelism and data movement, data parallel languages simultaneously make the inextricably coupled tasks of parallel programming easier and performance analysis more difficult. Because neither the interprocessor communication nor the mapping of loop iterations to processors are not visible in the application program, the application programmer cannot easily identify the causes of poor program performance — the underlying data movement and control thread asynchrony are hidden. Moreover, were one to capture the dynamics of data movement using the current Pablo software instrumentation, there would be no way to relate this data movement to the application developer's programming model.

Simply put, data parallel languages provide the mechanism to parallelize applications for massively parallel, distributed memory parallel systems, and Pablo provides the mechanism to analyze the dynamics of low-level data sharing and dynamic control flow. However, at present we have no mechanism for relating the low-level performance data to the high-level program description.

In a newly launched collaborative effort with Rice, we are working to integrate the Rice Fortran D editor and debugger with the Pablo performance analysis software to allow seamless development, debugging, and performance optimization of data parallel, Fortran D programs. In this integrated system, illustrated in 9, a performance query interface will allow application developers to ask high-level performance questions about their Fortran D programs (e.g., the overhead attributable to a particular data distribution directive).

Some performance questions can be answered at compile time using the compiler's data analysis, others require execution-time performance measurements. For those questions whose answers require measurement of the program's execution behavior, the Rice Fortran D compiler will gen-

erate instrumented code that, when linked with the Pablo performance data capture library, will generate the requisite dynamic performance data. The Pablo performance analysis software will provide the data reduction and visualization software to process the resulting dynamic data. The results of this dynamic analysis, together with the Fortran D dependence analysis data, form the response to the programmer's original, high-level query. The Fortran D software must then relate the response to the application source code.

12 Current Status and Software Plans

The design of the Pablo performance analysis system began in January of 1989. Much of the early design and implementation effort focused on the self-describing data format and the environment infrastructure for processing performance trace data in this form. The first demonstrable prototype became operational in the spring of 1991, with subsequent improvements throughout the summer and fall of that year. Clearly, much work remains, and both the performance instrumentation software and the data analysis environment are far from their final states.

As the development of the performance instrumentation software continues, we plan to complete the Fortran instrumentation parser, and integrate performance data presentation with display of source code. As we argued in 11, this is particularly important; many tools compute performance metrics without relating them to source code fragments. For example, it is not enough to know that an application program's performance on a massively parallel, distributed memory system is limited by the number of messages exchanged between parallel tasks. Instead, to correct performance problems and remove bottlenecks, users must know *where* in the application source code most messages where generated, *when* different code fragments interact by message passing (i.e., the temporal pattern of interactions), and *which* tasks are primarily responsible for the communication bottleneck. Given this information, a user can then use other performance metrics, that need not be tied to the source code, to identify *why* the bottleneck exists. The key is integrating performance metrics with the user's knowledge of the instrumented code.

For the performance analysis environment, we plan to expand the palette of data analysis and graphical display modules and broaden the configuration options for existing displays. Among the new modules we intend to develop are those supporting the identification and display of processor behavioral equivalence classes.

Identification of equivalence classes is particularly important for massively parallel systems. Fundamentally new techniques are needed to analyze and present performance data from massively parallel systems — extrapolating per processor display techniques to thousands of processors is not feasible; display screens lack the requisite pixel count. Although parallel programs contain many tasks, typically, there are only a few equivalence classes of task behavior. It often suffices to see aggregate behavior, with detail for equivalence class representatives and outliers. We hope to develop new data clustering and display modules that can scale with massively parallel systems.

Finally, as described in §6.5, four other, planned additions concern the environment's perceived and actual execution speed: control of the absolute execution rate, control of the logical display update rate (i.e., the logical separation of display snapshots), checkpointing the environment state to allow users to replay points of the data analysis, and parallel execution of data analysis graphs.

In summary, we believe the emphasis on *portability*, via source code instrumentation, a data meta-format, and data analysis modules without software embedded semantics, *scalability*, via parallel execution of data analysis graphs, and *extensibility*, via addition of new data analysis and

presentation modules, provides a powerful environment for studying the performance of application programs and system software for massively parallel systems.

References

[1] Arendt, J. W. Parallel Genome Sequence Comparison Using an iPSC/2 with a Concurrent File System. Master's thesis, University of Illinois at Urbana–Champaign, Department of Computer Science, Jan. 1991.

[2] Aydt, R. A. SDDF: The Pablo Self-Describing Data Format. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Mar. 1992.

[3] Berry, M. *et al.* The Perfect Club Benchmarks: Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications 3*, 3 (1989), 5–40. Fall.

[4] Blattner, M. M., Sumikawa, D. A., and Greenberg, R. M. Earcons and Icons: Their Structure and Common Design Principles. *Human-Computer Interaction 4* (1989), 11–44.

[5] Bly, S. *Sound and Computer Information Presentation.* PhD thesis, University of California, Davis, 1982.

[6] Couch, A. L. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors.* PhD thesis, Tufts University, Department of Computer Science, Apr. 1988.

[7] Deitel, H. M. *An Introduction to Operating Systems.* Addison-Wesley, 1984.

[8] Dennis, J. B. First Version of a Data Flow Procedure Language. In *Lecture Notes in Computer Science, 19.* Springer-Verlag, 1974, pp. 362–376.

[9] Dyer, D. S. A Dataflow Toolkit for Visualization. *IEEE Computer* (July 1990), 60–69.

[10] FCCSET. Grand Challenges 1993: High Performance Computing and Communications, the FY 1993 U.S. Research and Development Program. Federal Coordinating Council for Science, Engineering and Technology, Office of Science and Technology Policy, 1992.

[11] Francioni, J. M., Albright, L., and Jackson, J. A. The Sounds of Parallel Programs. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE Computer Society.

[12] Fromm, H., Hercksen, U., Herzog, U., John, K., Klar, R., and Kleinoder, W. Experiences with Performance Measurement and Modeling of a Processor Array. *IEEE Transactions on Computers 32*, 1 (Jan. 1983).

[13] Haban, D., and Wybranietz, D. A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems. *IEEE Transactions on Software Engineering 16*, 2 (Feb. 1990), 197–211.

[14] Heath, M. T., and Etheridge, J. A. Visualizing the Performance of Parallel Programs. *IEEE Software* (Sept. 1991), 29–39.

[15] Hiranandani, S., Kennedy, K., and Tseng, C.-W. Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *Supercomputing '91* (Nov. 1991), pp. 86–100.

[16] HPFF. DRAFT High-Performance Fortran Language Specfication. Tech. rep., High Performance Fortran Forum, Nov. 1992.

[17] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview.* Beaverton, Oregon, Nov. 1991.

[18] Jensen, D. W., and Reed, D. A. A Performance Analysis Exemplar: Parallel Ray Tracing. *Concurrency: Practice and Experience, to appear* (1992).

[19] Lillevik, S. L. Touchstone Program Overview. In *Proceedings of the Fifth Distributed Memory Computing Conference* (1990), IEEE Computer Society, pp. 647–657.

[20] Madhyastha, T. M. Porsonify: A Portable System for Data Sonification. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Mar. 1992.

[21] Madhyastha, T. M. A Portable System for Data Sonification. Master's thesis, University of Illinois at Urbana–Champaign, Department of Computer Science, May 1992.

[22] Madhyastha, T. M., and Reed, D. A. A Framework for Sonification Design. In *Data Sonification*, G. Kramer, Ed. Addison-Wesley Publishing Company, 1993.

[23] Malony, A. D., and Reed, D. A. Visualizing Parallel Computer System Performance. In *Instrumentation for Future Parallel Computing Systems*, M. Simmons, R. Koskela, and I. Bucher, Eds. Addison-Wesley Publishing Company, 1989, pp. 59–90.

[24] Malony, A. D., and Reed, D. A. A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube. In *1990 ACM International Conference on Supercomputing* (June 1990), Association for Computing Machinery.

[25] Malony, A. D., Reed, D. A., Arendt, J. W., Aydt, R. A., Grabas, D., and Totty, B. K. An Integrated Performance Data Collection Analysis, and Visualization System. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications* (Monterey, CA, Mar. 1989), Association for Computing Machinery.

[26] Malony, A. D., Reed, D. A., and Rudolph, D. C. Integrating Performance Data Collection, Analysis, and Visualization. In *Parallel Computer Systems: Performance Instrumentation and Visualization*, M. Simmons, R. Koskela, and I. Bucher, Eds. Addison-Wesley Publishing Company, 1990.

[27] MIDI Manufacturers Association. *MIDI – Musical Instrument Digital Interface, Specification 1.0.* International MIDI Association, Los Angeles, CA, 1988.

[28] Mink, A., and Carpenter, R. A VLSI Chip Set for a Mmultiprocessor Performance Measurement System. In *Parallel Computing Systems: Performance Instrumentation and Visualization*, M. Simmons, R. Koskela, and I. Bucher, Eds. Addison-Wesley Publishing Company, 1990.

[29] Mink, A., Draper, J., Roberts, J., and Carpenter, R. Hardware Assisted Multiprocessord Performance Measurements. In *Proceedings of the 12th IFIP Working Group 7.3 International Symposium on Performance: Performance 87* (Brussels, Belgium, Dec. 1987), pp. 151–168.

[30] NCSA. *NCSA HDF, Version 2.0.* University of Illinois at Urbana-Champaign, National Center for Supercomputing Applications, Feb. 1989.

[31] Rabenhorst, D. A., Farrell, E. J., Jameson, D. H., Linton, T. D., and Mandelman, J. A. Complementary Visualization and Sonification of Multi–Dimensional Data. In *Proceedings of the SPIE, Conference 1259, Extracting Meaning from Complex Data: Processing, Display, Interaction* (1990).

[32] Reed, D. A., and Rudolph, D. C. Experiences with Hypercube Operating System Instrumentation. *International Journal of High-Speed Computing* (Dec. 1989), 517–542.

[33] Rew, R. K. *netCDF User's Guide, Version 1.0.* Unidata Program Center, University Corporation for Atmospheric Research, Apr. 1989.

[34] Rudolph, D. C., and Reed, D. A. CRYSTAL: Operating System Instrumentation for the Intel iPSC/2. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications* (Monterey, CA, Mar. 1989).

[35] Scaletti, C., and Craig, A. B. Using Sound to Extract Meaning from Complex Data. In *Proceedings of the SPIE, Conference 1459, Extracting Meaning from Complex Data: Processing, Display, Interaction II* (San Jose, CA, 1991).

[36] Shields, K. A. iPablo User's Guide. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Nov. 1992.

[37] Stardent Computer, Inc. *Application Visualization System, User's Guide*, 1989.

[38] Stunkel, C. B., Fuchs, W. K., Rudolph, D. C., and Reed, D. A. Linear Optimization: A Case Study in Performance Analysis. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications* (Monterey, CA, Mar. 1989), Golden Gate Enterprises, pp. 265–268.

[39] Thinking Machines Corporation. *CM5 Technical Summary.* Cambridge, Massachusetts, Oct. 1991.

[40] Williams, C., and Rasure, J. A Visual Language for Image Processing. In *IEEE Workshop on Visual Languages* (Oct. 1990).

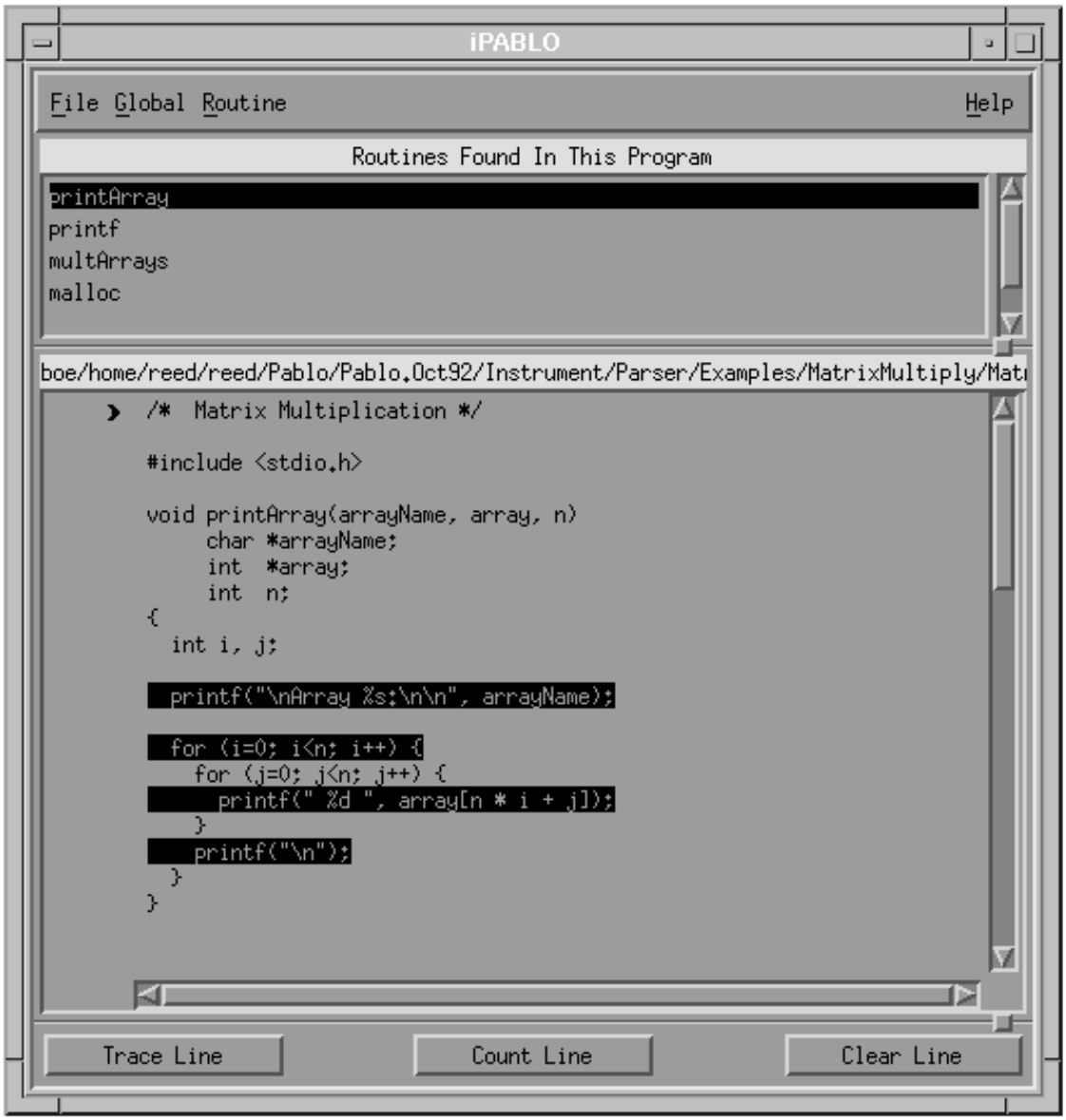

Figure 2: *iPablo* Graphical Instrumentation Interface

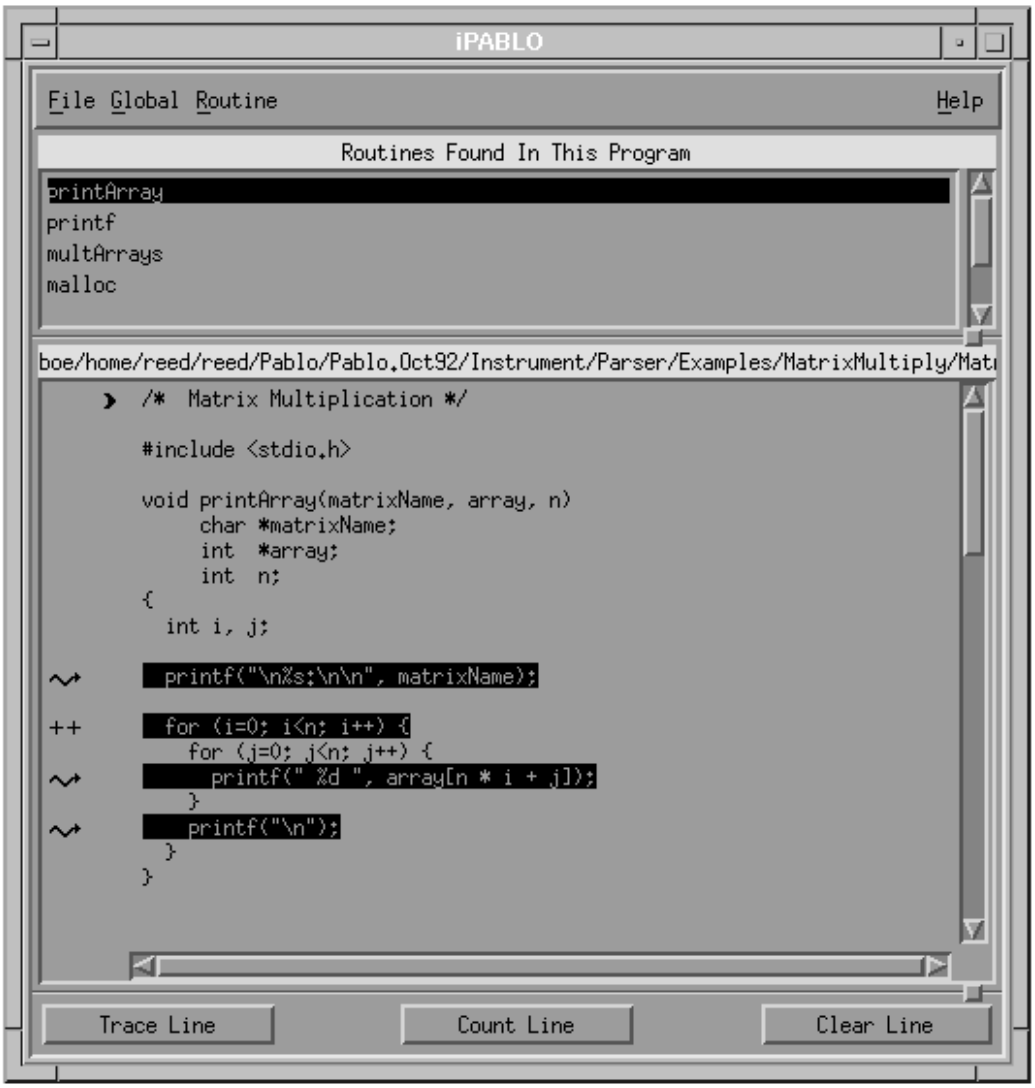

Figure 3: *iPablo* Instrumentation of One Procedure

```
/*
 * "Trace generation date" "November 1, 1991"
 */ ;;

#1:
// "event" "message sent to one or more processors"
"message send" {
        double  "timestamp";
        // "From" "Processor sending message"
        int     "SourcePE";
        // "To" "Processor(s) receiving message"
        int     "DestinationPE"[];
        // "Size" "Message length in bytes"
        int     "MessageLength";
};;

#2:
"context switch" {
        double  "timestamp";
        int     "processor_number";
        char    "process_name"[];
};;

"context switch" { 100.150000, 2, [30] { "Process 23" } };;
"message send"   { 100.100000, 0, [4] { 1, 3, 5, 7 }, 512 };;
"message send"   { 102.150000, 7, [1] { 1 }, 1012 };;
"context switch" { 108.000000, 4, [30] { "File I/O" } };;
```

Figure 4: Sample Pablo trace file (ASCII format)

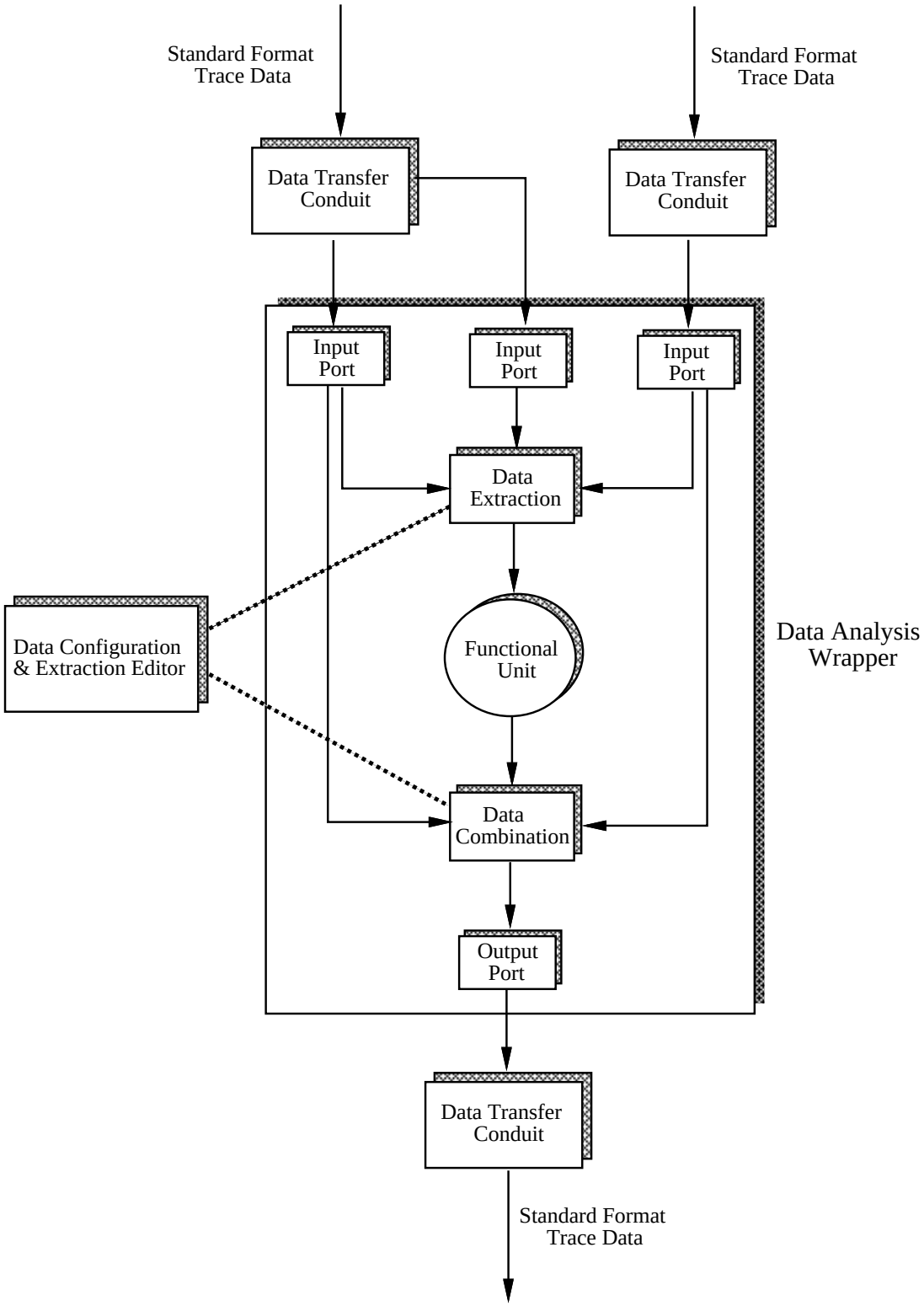

Figure 5: Pablo Data Analysis Module Components

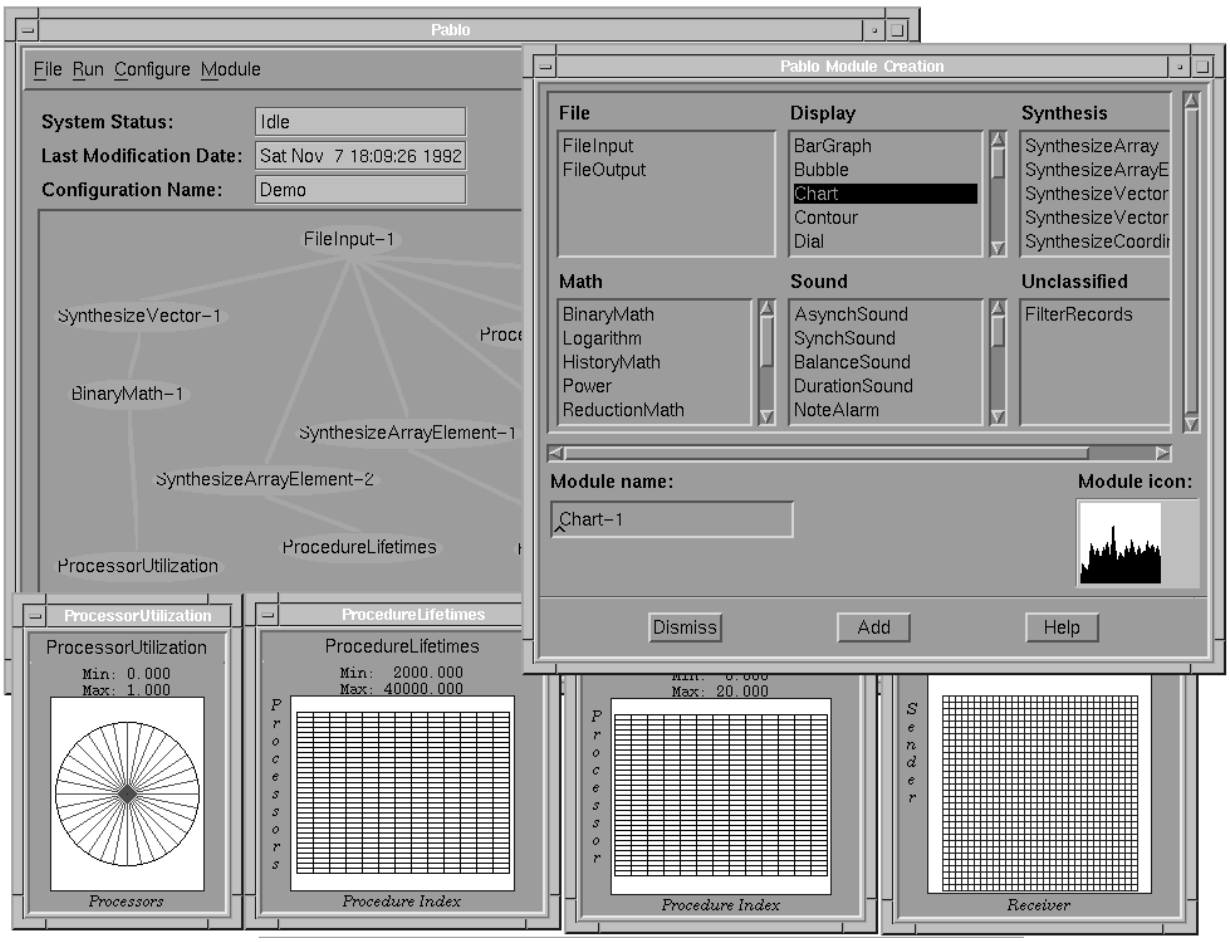

Figure 6: Pablo Graph Construction

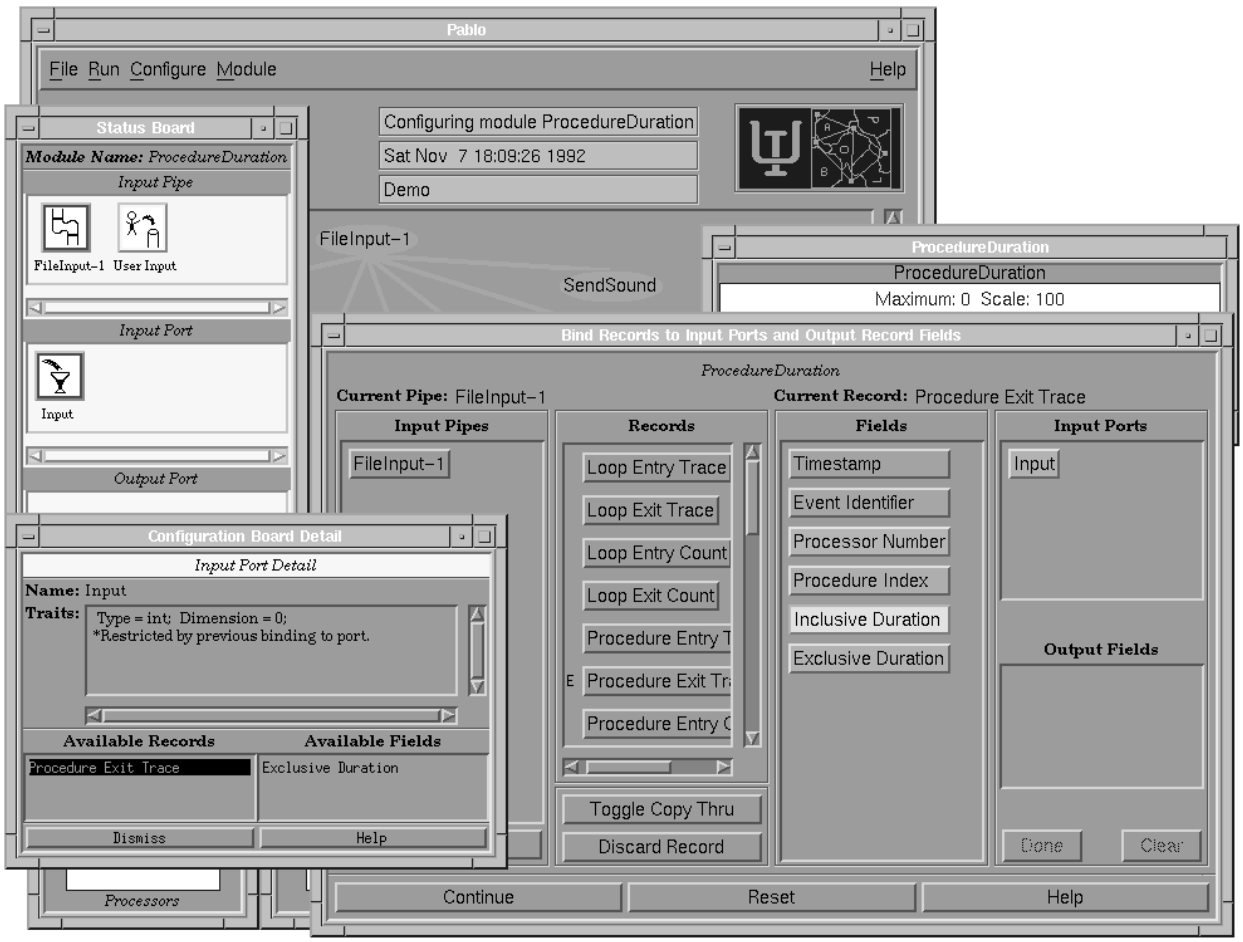

Figure 7: Pablo Analysis Graph Configuration

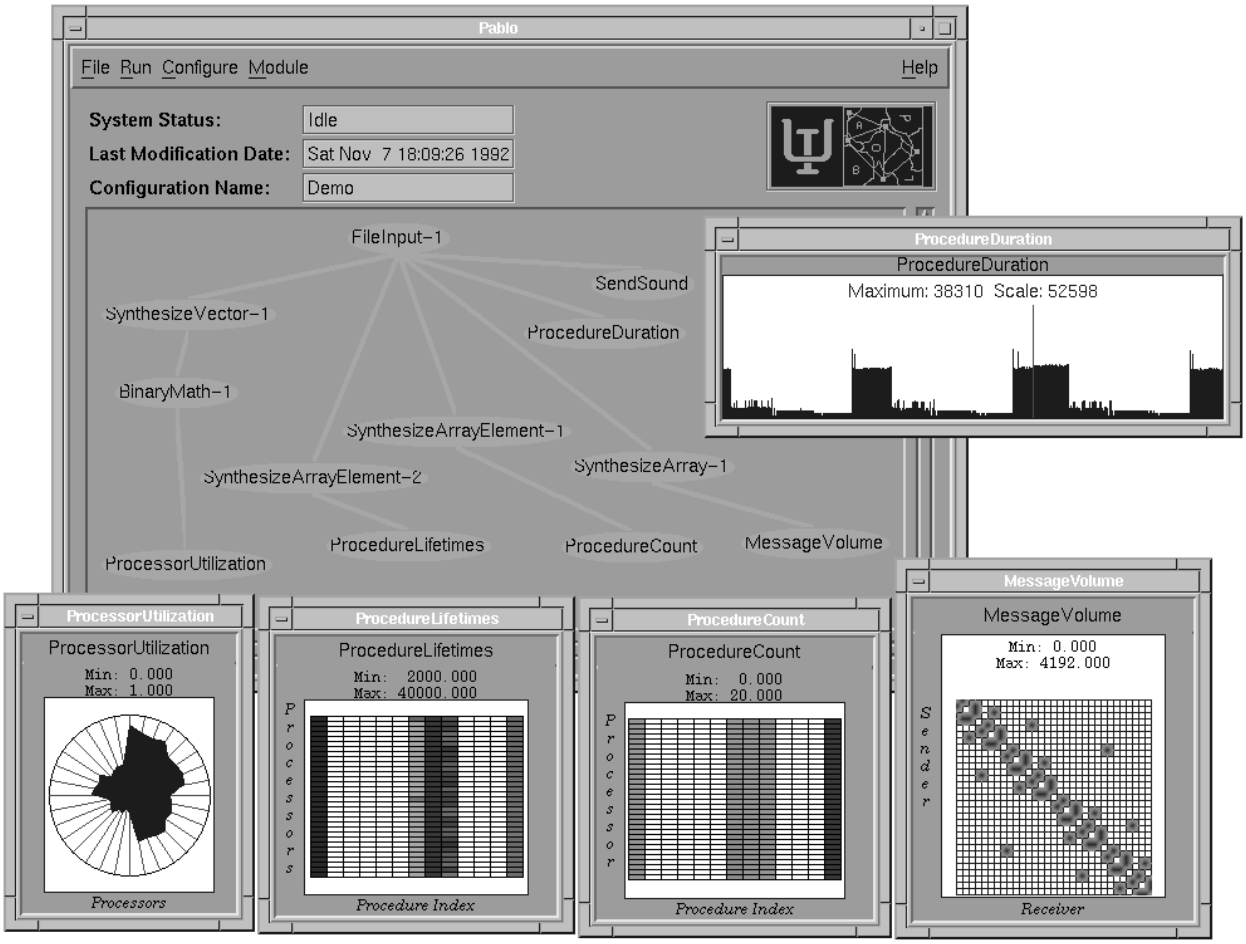

Figure 8: Pablo Analysis Graph Execution

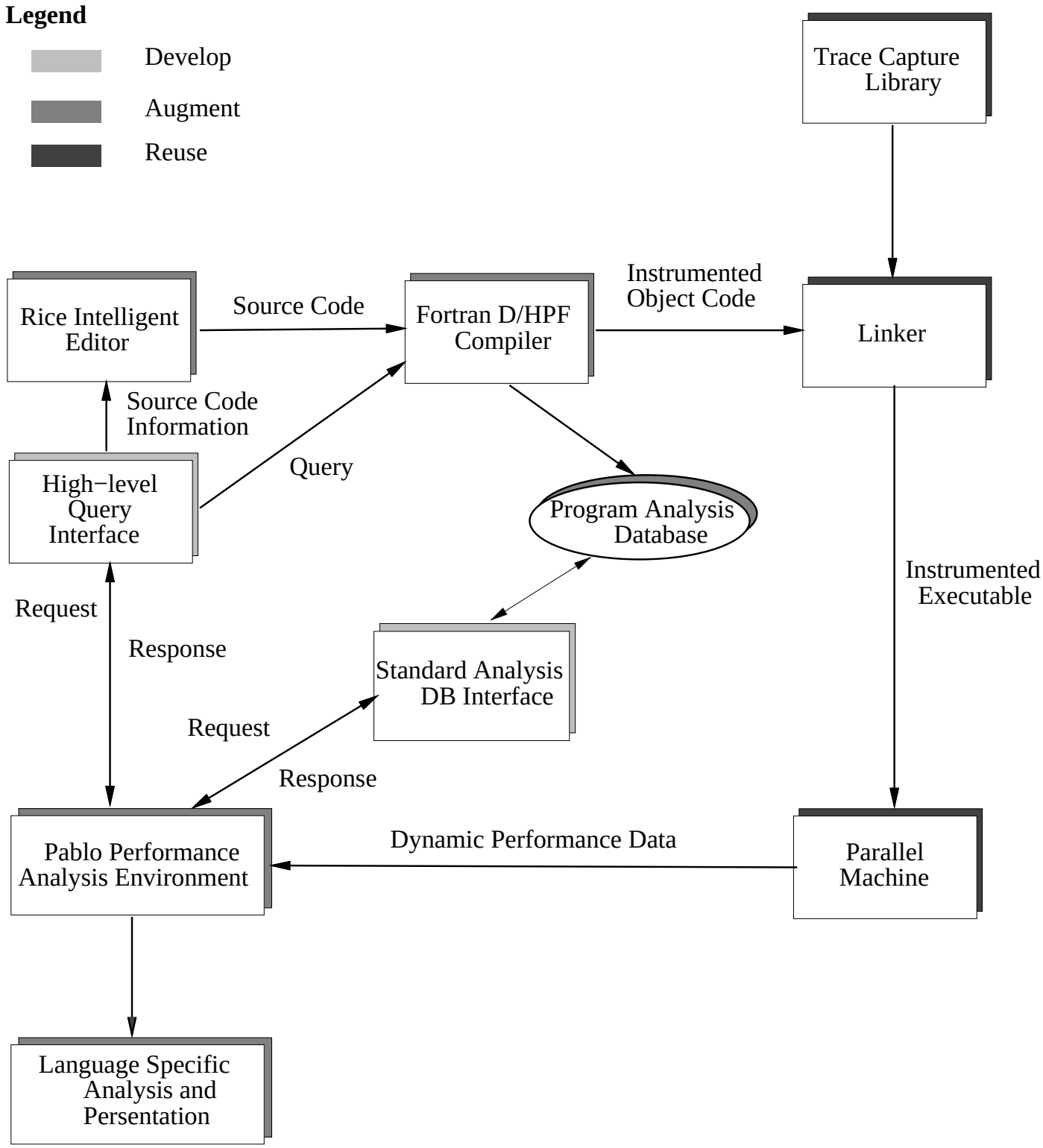

Figure 9: Performance Tool/Compiler Integration