The Paradyn Parallel Performance Measurement Tool

Barton P. Miller
Mark D. Callaghan
Jonathan M. Cargille
Jeffrey K. Hollingsworth
R. Bruce Irvin
Karen L. Karavanic
Krishna Kunchithapadam
Tia Newhall

University of Wisconsin, Madison

Because it inserts instrumentation during execution and only at the point where it's needed, Paradyn identifies and locates performance problems in long-running applications with little overhead.

Paradyn is a tool for measuring the performance of large-scale parallel programs. Our goal in designing a new performance tool was to provide detailed, flexible performance information without incurring the space (and time) overhead typically associated with trace-based tools. Paradyn achieves this goal by dynamically instrumenting the application and automatically controlling this instrumentation in search of performance problems. Dynamic instrumentation lets us defer insertion until the moment it is needed (and remove it when it is no longer needed); Paradyn's Performance Consultant decides when and where to insert instrumentation.

GUIDING PRINCIPLES AND CHARACTERISTICS

Seven principles guided the design of Paradyn: scalability, automated search, well-defined data abstractions, support for heterogeneous environments and high-level languages, open interfaces, and streamlined use. Below, we describe these principles and summarize the features in Paradyn that incorporate them.

Scalability

We need to measure long-running programs (hours or days) on large (about 1,000 node) parallel machines using large data sets. For correctness debugging, you can often test a program on small data sets and be confident that it will work correctly on larger ones, but for performance tuning, this is often not the case. In many real applications, as program and data-set size increase, different resources saturate and become bottlenecks.

We must also measure large programs that have hundreds of modules and thousands of procedures. The mechanisms for instrumentation, program control, and data display must gracefully handle this large number of program components.

Paradyn uses dynamic instrumentation to instrument only those parts of the program relevant to finding the current performance problem. It starts looking for high-level problems (such as too much total synchronization blocking, I/O blocking, or memory delays) for the whole program. Only a small amount of instrumentation is inserted to find these problems. Once a general problem is found, instrumentation is inserted to find more specific causes. No detailed instrumentation is inserted for classes of problems that do not exist.

Automate the search for performance problems

Our approach to performance measurement is to try to identify the parts of the program that are consuming the most resources and direct the programmer to these parts. Automating the search for performance infor-

0018-9162/95/$4.00 © 1995 IEEE

mation enables Paradyn to dynamically select which performance data to collect (and when to collect it). The goal is for the tool to identify the parts of the program that are limiting performance instead of requiring the programmer to do it.

Paradyn's Performance Consultant module has a well-defined notion, called the W^3 search model, of the types of performance problems found in programs and of the various components contained in the current program. The Performance Consultant uses W^3 information to guide dynamic instrumentation placement and modification.

Paradyn already works well in several domains and measures programs running on heterogeneous combinations.

Provide well-defined data abstractions

Simple data abstractions can unify the design of a performance tool and simplify its organization. Paradyn uses two important abstractions—metric-focus grids and time histograms—in collecting, communicating, analyzing, and presenting performance data.

A metric-focus grid is based on two lists (vectors) of information. The first vector is a list of performance metrics, such as CPU time, blocking time, message rates, I/O rates, or number of active processors. The second vector is a list of individual program components, such as a selection of procedures, processor nodes, disks, message channels, or barrier instances. The combination of these two vectors produces a matrix with each metric listed for each program component.

The matrix elements can be single values, such as current rate, average, and sum, or time histograms, which record metric behavior as it varies over time. The time histogram is an important tool in recording time-varying data for long-running programs.

Support heterogeneous environments

Parallel computing environments range from clusters of workstations to massively parallel computers. Heterogeneity arises in processor architectures, operating systems, programming models, and programming languages. Isolating each of these dimensions into abstractions within the performance tool can simplify porting. For example, adding support for the PVM programming model only requires knowing the name of the new communication and process creation operations; the underlying support for the Unix operating system, chip architecture, and programming language stays the same.

Paradyn already works well in several domains and measures programs running on heterogeneous combinations. Current hardware platforms include Thinking Machine Corporation's CM-5, SparcStation (including multiprocessors), Hewlett-Packard's PA-RISC, and IBM's RS/6000 and SP-2. Operating systems include Thinking Machine's TMC CMOST, SunOS 4.1, Solaris 2, Hewlett-Packard's HP/UX, and AIX. Programming models include PVM, CM-5 CMMD, and CM Fortran CM-RTS.

Support high-level parallel languages

Users of high-level parallel programming languages need performance information that is accurate and relevant to source code. When their programs exhibit performance problems at the lowest system levels, programmers must examine low-level problems while maintaining references to the high-level source code.

Paradyn allows high-level language programmers to view performance in terms of high-level objects (such as arrays and loops for data-parallel Fortran) and automatically maps the high-level information to low-level objects (such as nodes and messages). If users want to view the low-level information, Paradyn helps them relate performance data between levels.

Open interfaces for visualization and new data sources

Graphical and tabular displays are important mechanisms for understanding performance data. Paradyn has a set of standard visualizations (time histograms, bar graphs, and tables) and provides a simple interface to incorporate displays from other sources.

Equally important is the ability to incorporate new sources of performance data, such as cache miss data from the processor, network traffic from the network interfaces, or paging activity from the operating system. Paradyn's instrumentation is configurable to use any performance quantity that can be mapped into a program's address space. Including these new sources of data in Paradyn requires only a change to a configuration file.

Streamlined use

Ease of installation and ease of use are important features of any new tool. Installation should not require special system privileges or directory modification, and tool use should not require source code modification or special compiling techniques. In Paradyn, dynamic instrumentation avoids the need to modify, recompile, or relink an application. It also allows tools to attach to an already-running program (such as a parallel database server), monitor its performance for some interval, and then detach.

DYNAMIC PERFORMANCE MEASUREMENT

Paradyn differs from previous performance measurement tools in that program instrumentation and performance evaluation are done *during* execution of the application program. Thus, comparing Paradyn to previous tools raises questions about transient effects, brief periodic effects, and overhead for programs of shorter duration.

The answers are implicit in the fact that we are measuring long-running programs, with execution times of hours or even days. Although it might take seconds to insert new instrumentation and start evaluating the data, there is little chance that interesting behaviors will be missed. If a program runs for 10 hours, then even a 5-minute "transient" operation is less than 1 percent of the total execution time (and therefore not interesting for performance tuning). If a program repeatedly performs a brief (few seconds) operation, we will detect this behavior if the cumulative effect is large enough. Short-running programs might finish before Paradyn has had a chance

to isolate the performance problem(s). Although they are not our primary target, Paradyn may also be used to performance-tune these programs. To accommodate this class of applications, we are working on the use of multiple program runs in a single search for bottlenecks. In this case, Paradyn would save the state of its search for performance problems and rerun the program to complete the search.

Dynamic instrumentation differs from traditional data collection because it defers data selection until the program is running. Instrumentation to precisely count and time events is inserted by dynamically modifying the binary program. A more static approach is binary rewriting, a technique that reads an executable file and then generates a modified file with tracing inserted. This technique is used in tools such as Quantify[1] and qpt.[2] Quantify and qpt operate on static binaries, whereas dynamic instrumentation modifies binaries on the fly.

Trace-based tools designed for performance tuning parallel applications include Pablo,[3] AIMS,[4] and IPS-2.[5] These systems require an application to be recompiled, then run while trace data is gathered for the entire application, and finally analyzed post-mortem using the trace data. Such trace-based approaches may be limited by the amount of trace data needed to analyze a single large, long-running application.

MPP Apprentice,[6] a performance tool designed for the Cray T3D, avoids the scaling problem of trace-based tools by using static counters and timers, but it does require recompilation of the application to be tuned. Paradyn avoids both recompilation and the scaling difficulties inherent in a tracing approach.

SYSTEM OVERVIEW

Basic abstractions

Paradyn is built around two simple but powerful data abstractions—the metric-focus grid and time histogram—that unify internal system structure, giving users a consistent view of the system and the data it presents.

Metrics are time-varying functions that characterize some aspect of a parallel program's performance; examples include CPU utilization, memory usage, and counts of floating-point operations. A focus is a specification of a part of a program execution expressed in terms of program resources. Typical resource types include synchronization objects, source code objects (procedures, basic blocks), threads and processes, processors, and disks. Resources are separated into several different hierarchies, each representing a class of objects in a parallel application. For example, a resource hierarchy for CPUs contains each processor. A focus contains one or more components from each resource hierarchy. One focus might be all synchronization objects accessed by a single procedure on one processor. The combination of a list of metrics with a list of foci forms a matrix (called a grid in Paradyn) containing the value of each metric for each focus. The Performance Consultant and visualizations receive performance data by specifying one or more metric-focus grids.

Paradyn stores performance data internally in a data structure called a time histogram,[7] which is a fixed-size array whose elements (buckets) store a metric's values for successive time intervals. Two parameters determine the granularity of the data stored: initial bucket width (time interval) and number of buckets. Both parameters are supplied by higher level consumers of the performance data. If a program runs longer than the initial bucket width times the number of buckets, we double the bucket width and re-bucket the previous values. The change in bucket width can cause a corresponding change in the sampling rate for performance data, reducing instrumentation overhead. This process repeats each time we fill all the buckets. As a result, the rate of data collection decreases logarithmically, while maintaining a reasonable representation of the metric's time-varying behavior.

System components

Paradyn consists of the main Paradyn process, one or more Paradyn daemons, and zero or more external visualization processes. The central part of the tool is a multithreaded process that includes the Performance Consultant, Visualization Manager, Data Manager, and User Interface Manager. Figure 1 shows the Paradyn architecture.

The Data Manager handles requests from other threads for data collection, delivers performance data from the Paradyn daemon(s) to the requesting thread(s), and maintains and distributes information about the metrics and resource hierarchies for the currently defined application.

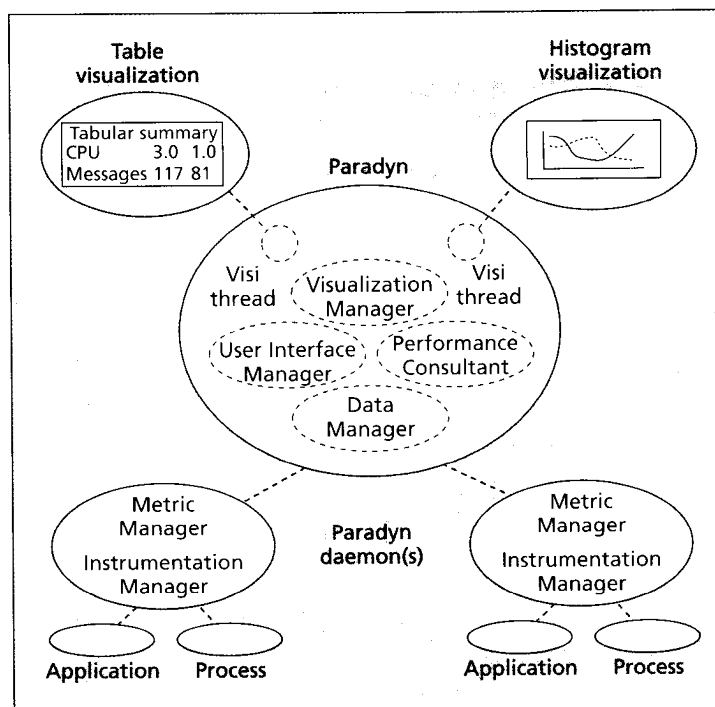

Figure 1. Overview of Paradyn structure. Dotted ovals are threads, and solid ovals are processes.

The User Interface Manager provides visual access to the system's main controls and performance data. The Performance Consultant controls the automated search for performance problems, requesting and receiving performance data from the Data Manager. The Visualization Manager starts visualization processes; one visualization thread is created in Paradyn for every external visualization process that is started. The job of a visualization thread is to handle communication between the external visualization process and the other Paradyn modules.

The daemon contains Paradyn's platform-dependent parts. It is responsible for inserting the requested instrumentation into the executing processes it monitors. The interface between the Paradyn process and daemon supports four main functions: process control, performance data delivery, performance data requests, and the delivery of high-level language mapping data. The daemon services requests from Paradyn for process control and performance data and delivers performance data from the application(s) to Paradyn. We currently support daemons for various versions of Unix, the TMC CM-5, and networks of workstations running PVM.

Configuration files

Paradyn configuration language (PCL) files describe all architecture, operating system, and environment characteristics of applications and platforms. PCL lets users create new metrics and instrumentation, incorporate new visualizations, specify alternate Paradyn daemons, set various display and analysis options, and specify command lines for starting applications.

Paradyn's default PCL file describes basic metrics, instrumentation, visualizations, and daemons. Each user can provide an additional PCL file with personalized settings and options. Users can also create an application-specific PCL file that describes details of an application and how it is run.

DYNAMIC INSTRUMENTATION

Paradyn instruments only those parts of the program that are relevant to the current performance problem[8] and defers instrumentation until the program is executing.

Interface

Requests for dynamic instrumentation are made in a metric-focus grid, which the Paradyn daemon translates. Translation is a two-step process: The Metric Manager translates the metric-focus requests into machine-independent abstractions, which the Instrumentation Manager then converts to machine instructions.

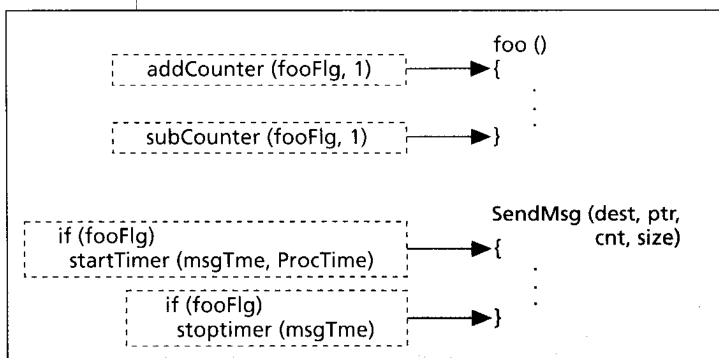

Figure 2. Sample metric computation.

Dynamic instrumentation provides two types of objects, timers and counters, to extract performance information from an application. Counters are integer counts of the frequency of some event, and timers measure the amount of time spent performing various tasks (in either wall-time or process-time units).

Points, primitives, and predicates

Points, primitives, and predicates provide a simple, machine-independent set of abstractions that are used as building blocks for dynamic instrumentation. Points are locations in the application's code where instrumentation can be inserted. Primitives are simple operations that change the value of a counter or a timer. Predicates are Boolean expressions that guard the execution of primitives (essentially, If statements). By inserting predicates and primitives at the correct points in a program, we can compute a wide variety of metrics.

The points currently available in our system are procedure entry, procedure exit, and individual call statements, but we are extending them to include basic blocks and individual statements. We provide six primitives: set counter, add to counter, subtract from counter, set timer, start timer, and stop timer. We also provide a primitive, used primarily to discover resources and to record mapping information, to call arbitrary functions. To compute the predicate, we use counters, constants, parameters to a procedure, a procedure return value, and combinations of these, using relational and numeric operators. Predicates support constraints by limiting when operations on counters and timers can occur.

Figure 2 shows how primitives and predicates can be combined to create metrics. It computes the time spent sending messages on behalf of the procedure *foo* and its descendants. The *fooFlg* counter keeps track of whether *foo* is on the stack; it is incremented on entry and decremented on exit from *foo*. The value of *fooFlg* is then used as a predicate to control whether the *msgTime* timer primitives will be called upon entry and exit from *SendMsg*. When *foo* is not active, the primitives will not be executed.

The translation from metric-focus specifications to points, primitives, and predicates is described by metric definitions contained in the configuration language's files. These definitions simplify the addition of new metrics, porting of Paradyn to new systems, and user customization. We provide a standard library of metric descriptions, and users (and other tools) can add to this library. The metric descriptions consist of definitions and constraints. Some metric definitions and resource constraints are generic and apply to all platforms; others are specific to a platform or programming model.

A metric definition is a template that describes how to compute a metric for different resource combinations. It consists of a series of code fragments that create the primitives and predicates to compute the desired metric. We need to be able to compute each metric for any combination of resource constraints. To make these metric definitions compact and modular, we divide metric definition into two parts: a base metric and a series of resource con-

straints. The base metric defines how a metric is computed for an entire application (all procedures, processes, or processors). A resource constraint defines how to restrict the base metric to an instance of a resource in one of the resource hierarchies. Resource constraints usually translate to an instrumentation predicate.

Instrumentation generation

The Instrumentation Manager encapsulates the architecture-specific knowledge; it locates the allowable instrumentation points and performs the final translation of points, primitives, and predicates into machine-level instrumentation. When Paradyn is initially connected to an application process, the Instrumentation Manager identifies all potential instrumentation points by scanning the application(s) binary image(s). Procedure entry and exit, as well as procedure call sites, are detected and noted as points.

After Paradyn is connected to the application, the Instrumentation Manager waits for requests from the daemon's Metric Manager, translates them into small code fragments, called trampolines, and inserts them into the program. Two types of trampolines, base trampolines and mini-trampolines, are used. There is one base trampoline per point with active instrumentation. A base trampoline is inserted into the program by replacing the machine instruction at the point with a branch to the trampoline, and relocating the replaced instruction to the base trampoline. A base trampoline has slots for calling mini-trampolines both before and after the relocated instruction.

Mini-trampolines contain the code to evaluate a specific predicate or invoke a single primitive. There is one mini-trampoline for each primitive or predicate at each point. Creating a mini-trampoline requires generating appropriate machine instructions for the primitives and predicates requested by the Metric Manager. The Instrumentation Manager assembles the necessary instructions, which are transferred to the application process by a variation of the Unix ptrace interface. The generated code also includes appropriate register save and restore operations.

Data collection

Once instrumentation has been inserted into the application, data begins flowing back to the higher level clients. The current value of each active timer and counter is periodically sampled and transported by the Paradyn daemon to the Data Manager. Note that the instrumentation keeps track of the precise value of each performance metric, and the sampling rate determines only how often Paradyn sees the new value.

It is easy to integrate performance data from external sources into Paradyn. Instrumentation is inserted to read the external source and store the result in a counter or a timer. For example, most operating systems keep a variety of performance data that can be read by user processes; examples include statistics about I/O, virtual memory, and CPU use. Several machines also provide hardware-based counters. For example, the IBM Power2, Cray Y-MP, and Sequent Symmetry systems provide detailed counters of processor events. Data from external sources is treated identically to Paradyn's own instrumentation. External data can be constrained in the same way as other performance metrics, to relate it back to specific parts of a program. For example, if we have a way to read the cumulative number of page faults taken by a process, we can read this counter before and after a procedure call to approximate the number of page faults taken by that procedure.

Internal uses

Resource discovery is an important use of dynamic instrumentation. It determines which resources an application uses and builds the resource hierarchies from this information. Much resource information can be determined statically when Paradyn is first connected to the application; for example, at this point we know all of the procedures that might be called and what types of synchronization libraries are linked into the application. However, some aspects of resource discovery must be deferred until the program is executing. For example, information about which files are read or written during execution can only be determined when the files are first accessed. To collect this runtime resource information, instrumentation is inserted into the application program. We insert instrumentation (using the same technique as performance data instrumentation) to record the file names on open requests.

Paradyn's goal is to assist the user in locating program performance problems—that is, parts of the program that contribute significant time to its execution.

Another important use of dynamic instrumentation is the collection of dynamic mapping information for high-level languages. Many parallel languages defer mapping data structures to processor nodes until runtime, and some languages change data mappings during execution. In these cases, we dynamically instrument runtime mapping routines, and the Paradyn daemons send the collected mapping information to the Data Manager. The Data Manager uses the information to support language-specific views of performance data.

SEARCH MODEL AND PERFORMANCE CONSULTANT

Paradyn's goal is to assist the user in locating program performance problems—that is, parts of the program that contribute significant time to its execution. To assist in finding these problems, Paradyn has a well-defined model, W^3,[9] that organizes information about a program's performance. Paradyn's Performance Consultant module uses the W^3 search module to automate the identification of performance problems by using data gathered via dynamic instrumentation.

W^3 search model

The W^3 search model abstracts aspects of a parallel program that can affect performance on the basis of three questions: Why is the application performing poorly? Where is the performance problem? When does the problem occur?

To answer the "why" question, our system includes hypotheses about potential performance problems in parallel programs. We collect performance data to test whether these problems exist in the program. To answer the "where" question, we isolate a performance problem

to a specific program component or machine resource. (We use the term "resource" to mean either a machine resource or a program component—for example, a disk system, a synchronization variable, or a procedure.) To identify "when" a problem occurs, we try to isolate a problem to a specific time interval during the program's execution. Isolating a performance problem is an iterative process of refining our answers to these three questions. Our model treats the three questions as orthogonal axes of a multidimensional search space.

"WHY" AXIS. The first performance question programmers ask is often "Why is my application running so slowly?" The "why" axis represents broad problems that can cause slow execution. Potential performance problems are represented by hypotheses and tests. Because our model decouples the type of problem ("why") from its location ("where"), hypotheses encode general types of performance problems. One hypothesis, for example, might be that a program is synchronization bound. Since hypotheses represent activities universal to all parallel computation, they are independent of the program being studied and the algorithms it uses. Hence, a small set of hypotheses (a couple dozen), provided by the tool builder, can cover most types of performance problems.

Hypotheses can be refined into more precise hypotheses. The dependence relationships between hypotheses define the search hierarchy for the "why" axis. These dependencies form a directed acyclic graph, and axis searching involves traversing this graph. Figure 3 shows a partial "why" axis hierarchy; the current hypothesis is HighSyncBlockingTime. This hypothesis was reached after first concluding that a SyncBottleneck exists.

Tests are Boolean functions that evaluate the validity of a hypothesis. Tests are expressed in terms of a threshold and one or more metrics (for example, synchronization blocking time is greater than 20 percent of the execution time).

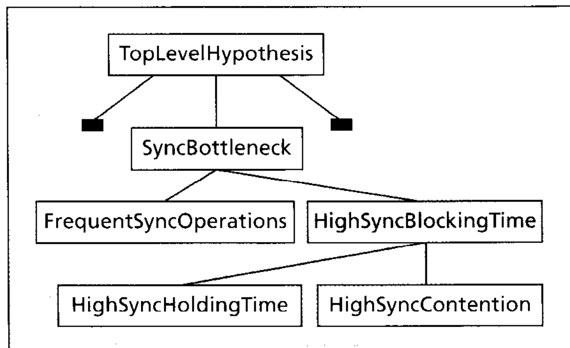

Figure 3. A portion of the "why" axis representing several types of synchronization bottlenecks. The shaded node shows the hypothesis currently being considered.

"WHERE" AXIS. The second performance question most programmers ask is "What part of my application is running slowly?" The "where" axis represents the different program components and machine resources that can be used to isolate a problem source. Searching along the "why" axis classifies the problem, while searching along the "where" axis pinpoints its location. For example, a "why" search may show that a program is synchronization bound, and a subsequent "where" search may isolate one hot synchronization object among many thousands in the program.

The "where" axis represents the different foci that can be measured. Each axis hierarchy has multiple levels, with the leaf nodes being the instances of the resources used by the application. Isolating a performance problem to a leaf node indicates that a specific resource instance is responsible for the problem. Higher level nodes represent

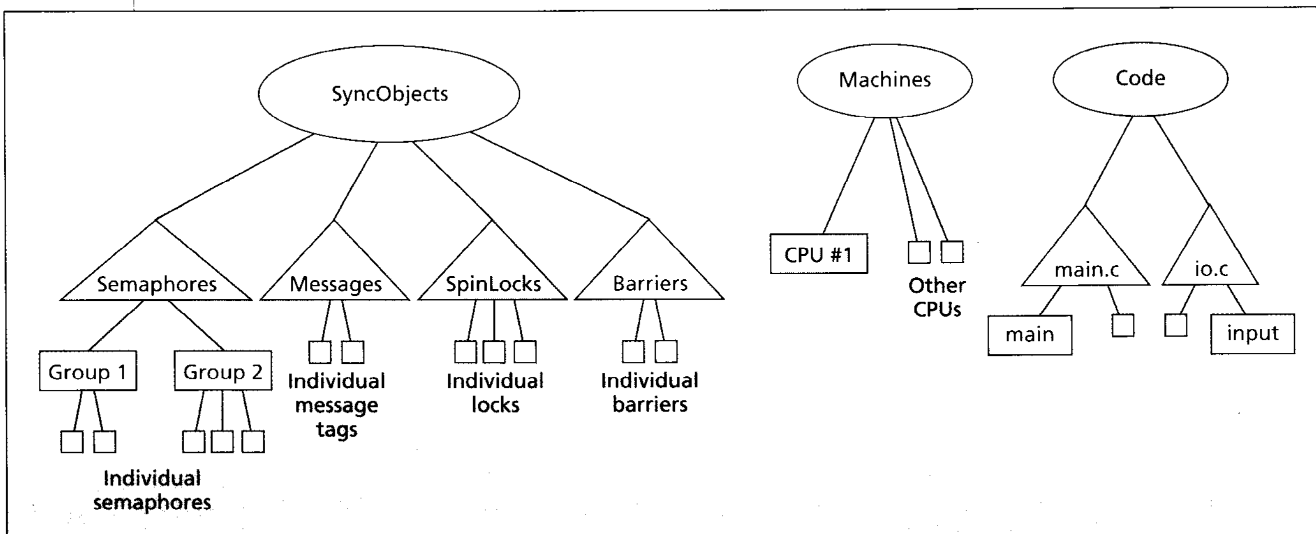

Figure 4. A sample "where" axis with three class hierarchies. The shaded nodes show the current focus. The oval objects are defined in the W³ search model. The triangles represent static entities based on the application, and the rectangles represent dynamically (runtime) identified entities. The shaded nodes indicate the current focus (all SpinLocks on CPU #1, in any procedure). The Paradyn resource hierarchies include several other classes, such as I/O, memory, and process, which are not shown.

collections of resource instances. For example, a leaf resource might be a procedure and its parent a module (that is, a source file in the C programming language). Isolating a performance problem to a module indicates that the problem is due to the collective performance of the module's procedures. Each resource hierarchy can be refined independently.

The trees in Figure 4 represent sample resource hierarchies. The root of the leftmost hierarchy is SyncObjects. The next level contains four types of synchronization: Semaphores, Messages, SpinLocks, and Barriers. Below the SpinLocks and Barriers nodes are the individual locks and barriers used by the application. The children of the Messages node are the types of messages used. The children of the Semaphores node are the semaphore groups used in the application. Below each semaphore group are the individual semaphores. Different components of the "where" axis may be created at different times. Some nodes are defined statically, some when the application starts, and others during the application's execution. The root of each resource hierarchy is statically defined.

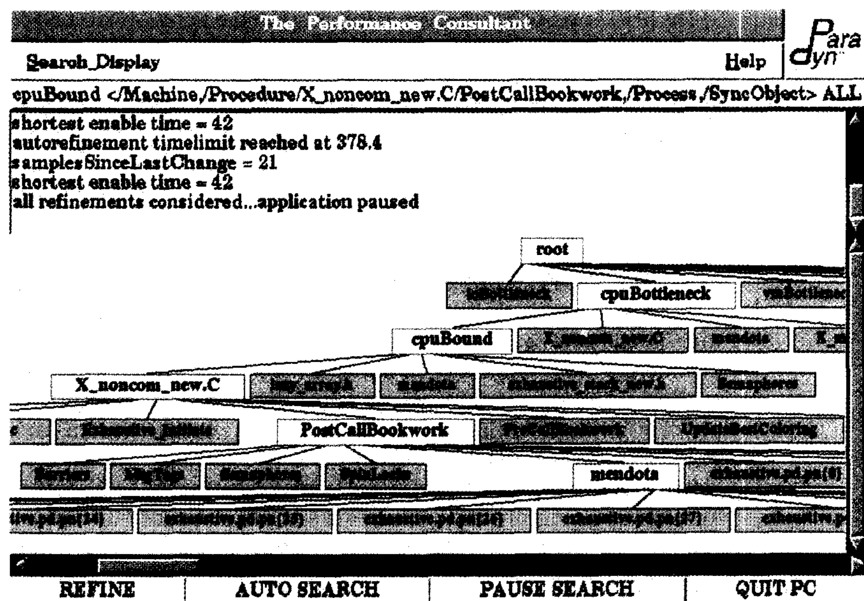

Figure 5. Performance Consultant search for the graph-coloring application.

W^3 can also represent resources specific to high-level parallel programming languages, by representing each high-level language abstraction with its own "where" axis. A language-specific axis contains only those resource hierarchies that correspond to resources found in that language. For example, a data-parallel Fortran "where" axis would include a data-parallel array hierarchy. Language-specific resource information is imported via PCL configuration files. Each language's compiler may emit a PCL file that describes the resources and metrics for each program that it compiles.

Abstract resources must ultimately map to resources and metrics that Paradyn can measure. For example, a high-level matrix reduction might map to the computations and messages used for reductions in a runtime system. Therefore, PCL files written by compilers must also include mappings that explain how high-level resources and metrics map to lower level resources and metrics. Paradyn uses the mappings to satisfy high-level focus-metric requests. If the user (or the Performance Consultant module) wants to measure time spent reducing a matrix, then Paradyn will use the compiler-supplied information to automatically map the request to measure time spent in all the routines that perform matrix reductions.

"WHEN" AXIS. The third performance question programmers may ask is "At what time did my application run slowly?" Programs have distinct execution phases. Within each phase, performance tends to be uniform, but behavior may differ radically from phase to phase. The "when" axis represents an application's execution time as a series of distinct phases, which can be independently tested for performance problems.

Performance Consultant

Paradyn's Performance Consultant module discovers performance problems by automatically searching through the space defined by the W^3 search model. Refinements are made across the "where," "when," and "why" axes without user involvement. Since there is a cost associated in evaluating refinements (due to collecting different metrics), we need to select a subset of the possible refinements to evaluate at once. We determine a list of possible refinements by considering the children of the current nodes along each axis, then order this list using internally defined hints. Finally, we select one or more refinements from the ordered list. If a selected refinement is not true, we consider the next one. Paradyn will conduct a fully automatic search, let the user make individual manual refinements to direct the search, or combine these two methods.

The search history graph (SHG) provides feedback about the search process currently underway. This graph records refinements considered along the "why," "where," and "when" axes and the result of testing the refinements. Figure 5 shows an SHG display from one of our sample application programs.

Each node in the graph represents a single step in the overall search process, which is a refinement along one of the three axes. The nodes are colored according to the current state of the particular hypothesis it represents: currently being tested (pink), tested true (blue), tested false (green), or never tested (orange). The arcs indicate refinements and are color-coded according to axis. Refinements made along the "why" axis are in blue, and refinements along the "where" axis are in purple. The node label indicates a particular node being explored; for example, refinements considered from the root node include a node from the "why" axis for cpuBottleneck. Because each step of the search is limited to a single refinement, the complete focus of any node can be determined by the path from the root node to the desired node.

The SHG is useful because it represents the refinements made, those tried and rejected, and those possible but not tried. The current exploration path can easily be determined by following the blue ("true") nodes from the root to a leaf. If you follow the blue nodes in Figure 5, you'll see that the search discovered that the program was CPU bound in procedure PostCallBookwork in module X_ noncom_new.C for machine partition mendota.

OPEN VISUALIZATION INTERFACE

Paradyn's visualization interface provides a mechanism to easily add new visualizations (visis) to the system. Many commercial data visualization packages are currently available. Rather than redevelop much of this work, we provided an interface that lets a programmer use existing visualizations. It is straightforward to build visis that provide data to commercial data visualization packages such as AVS,[10] or incorporate displays from systems such as ParaGraph[11] or Pablo.[3] The visi interface has been used for evaluating performance predicates for application steering and logging performance data for experiment management.

The Performance Consultant discovered and isolated a CPU bottleneck in the coloring application. It discovered two synchronization problems and a CPU bottleneck in the linear programming code.

Paradyn provides a simple library and remote procedure call interface to access performance data in real time. Visis in Paradyn are external processes that use this library and interface, and all performance visualizations are implemented as visis. Paradyn currently provides visis for time histograms (strip plots), bar charts, and tables.

After the user selects a visi from the menu, Paradyn provides the menus to select foci (program components) and metrics for display. The visi library isolates the visi from the details of Paradyn's internal structure and user interface. The selection of a list of performance metrics for a list of foci can most easily be pictured as a two-dimensional array. The visi library provides a C++ class, called the DataGrid, which is the visi programmer's interface to performance data. Each element of the DataGrid is a time histogram, representing the metric's time-varying behavior. The library also provides aggregation functions, such as minimum, maximum, current, average, and total, that can be invoked over each DataGrid element.

EXAMPLES OF USE

We've used Paradyn to study several parallel, distributed, and sequential applications. To demonstrate Paradyn's basic features and show how programmers use Paradyn to find performance problems, we'll look at two applications, a graph-coloring program based on a branch-and-bound search and a linear programming optimization code that uses a domain decomposition method. Both programs were written by people outside the Paradyn project and were intended to solve real application problems; both ran on a TMC CM-5 in a 32-node partition.

When we ran these applications with Paradyn, the Performance Consultant discovered and isolated a CPU bottleneck in the coloring application. It discovered multiple problems (two synchronization problems and a CPU bottleneck) in the linear programming code.

When a programmer starts an application, Paradyn displays an initial "where" axis and is ready to accept user commands to control the application, display visualizations of performance data, or invoke the Performance Consultant to find bottlenecks. The user may start or stop the application as many times as desired during execution. Stopping the application stops the flow of data to visualizations and also stops the Performance Consultant.

The "where" axis display shows each resource hierarchy. To select a particular focus, a user selects up to one node from each resource hierarchy. For example, selecting the root of the procedure hierarchy and a leaf node in the machine hierarchy requests all procedures on a particular machine. To display a visualization of a metric for a focus, a user simply selects a focus in the "where" axis display and selects a visualization from the Start Visual menu. Paradyn then prompts the user for a list of metrics and starts the visualization.

Alternate high-level language "where" axis views are displayed in separate windows. Paradyn uses static and dynamic mapping information to map each abstract focus to the base view. When the user selects a focus in an abstract view, Paradyn automatically highlights the corresponding resources in the base view.

Typically, users start the Performance Consultant on an automated search and wait for Paradyn to find a performance bottleneck. When the Performance Consultant is running, it displays a window similar to the one shown in Figure 5. The top row of the Performance Consultant window contains pull down menus for display configuration. The middle area reports the status of the search (such as a description of the current bottleneck, an indication that a previously true bottleneck is no longer present, or notice that a new set of refinements is being considered). The largest area is a display of the search history graph. Nodes in the graph are colored to indicate the state of their corresponding hypotheses, as discussed above. Nodes are added to the SHG as refinements are made, and the nodes change color to reflect the current state of the search for bottlenecks. The bottom of the window contains buttons for controlling the search process.

Graph-coloring application

Our first example demonstrates Paradyn's analysis of a graph-coloring program. Called match-maker, it is a branch-and-bound search program with a central manager that brokers work to idle processors. It uses CMMD, the CM-5 explicit message-passing library. The program is written in C++ and contains 4,600 lines of code in 37 files.

The Performance Consultant discovered an initial CPU bottleneck in match-maker after 10 seconds of execution. Figure 5 shows some of the hypotheses and foci considered by the Performance Consultant. Starting from the root node, the Performance Consultant considered several types of bottlenecks (synchronization, I/O, CPU, virtual memory, and instrumentation). At this point, it identified a CPU bottleneck. At the next step, it considered refinements to the CPU bottleneck and confirmed that the program was CPU bound. Next, the Performance Consultant refined the CPU bottleneck to a specific module in the program, X_noncom_new.C. The Performance Consultant then iso-

lated the bottleneck to the procedure PostCallBookwork in that module. Finally, it (trivially) isolated the problem to the single partition used. The partition refinement is labeled mendota, the name of the partition's manager. Since the problem was diffused across all the processes, the Performance Consultant could not further refine the bottleneck.

We then displayed a visualization of the bottleneck in the graph coloring application with a time histogram display of CPU time for procedure PostCallBookwork and for the whole program. The time histogram display in Figure 6 verifies that PostCallBookwork was responsible for a large percentage of the application's CPU time.

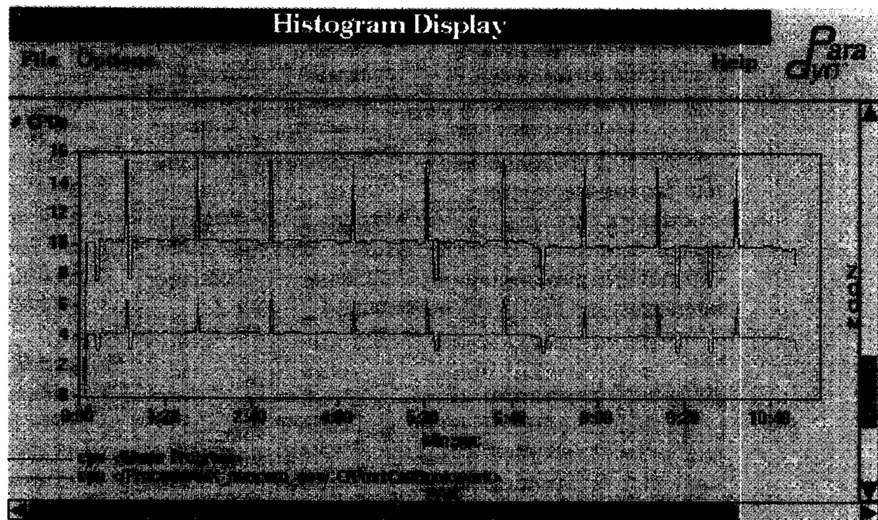

Figure 6. Visualization of bottleneck in graph-coloring application.

Message-passing optimization application

Our second application, called Msolv, uses a domain decomposition method for optimizing large-scale linear models. The application consists of 1,793 lines of code in the C programming language and uses a sequential, constrained-optimization library package called Minos. Uninstrumented, Msolv runs for 1 hour 48 seconds on a 32-node CM-5 partition. Instrumentation overhead during the Performance Consultant search was less than the program's normal, approximately 2 percent variation in runtime.

Paradyn found three bottlenecks in this program. First, it found a synchronization bottleneck as the nodes initialized. This bottleneck, which lasted less than one minute, was not further refined by the Performance Consultant. Second, it found a CPU bottleneck in the module minos_part.c during an initial computation phase. Third, the Performance Consultant located a key synchronization bottleneck that persisted for the rest of the program's execution. The search history graph of the isolation of this third bottleneck appears in Figure 7.

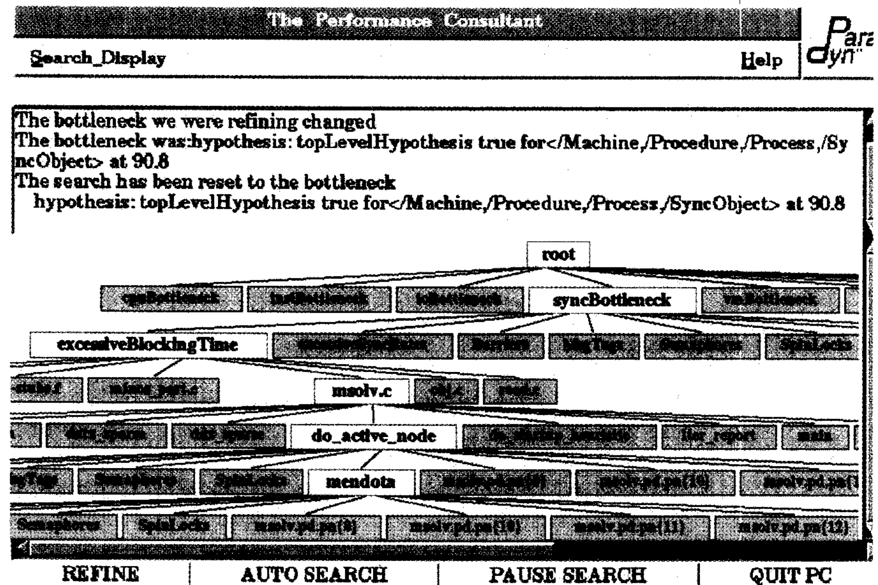

Figure 7. Performance Consultant search for the Msolv application.

The Performance Consultant made five refinements while locating the synchronization bottleneck. First, it discovered a synchronization bottleneck in the program. Second, it identified that the bottleneck was due to ExcessiveBlockingTime (as opposed to many short synchronization operations performed too frequently). Third, it isolated the synchronization bottleneck to the file msolv.c. Fourth, it refined the bottleneck to the procedure do_active_node. Again, it made the trivial refinement to the single partition used, mendota. The Performance Consultant attempted to isolate the bottleneck to a particular processor node, but the refinement failed because the bottleneck was diffused across all the processors.

To gain a better understanding of the synchronization problem, we displayed a bar chart showing the time the application spent on synchronization. One bar represented synchronization time for the entire program, and others showed times for each node. The display showed that the synchronization bottleneck was diffused across all nodes and could not be refined further.

THE PARADYN PARALLEL PERFORMANCE MEASUREMENT TOOLS incorporate several novel technologies. Dynamic instrumentation offers the chance to significantly reduce measurement overhead, and the W^3 search model, as embodied in the Performance Consultant, provides instrumentation control. The synergy between these two technologies results in a performance tool that can automatically search for performance problems in large-scale parallel programs. Paradyn's support for high-level parallel languages lets programmers study the performance of their programs using the language's native abstractions. In addition, we provide

detailed, time-varying data about a program's performance. As a result, programmers with large applications can use Paradyn as easily as someone with a small prototype application. Uniform data abstractions, such as the metric-focus grid and time histogram, allow simple interfaces within Paradyn and provide easy-to-understand interfaces to the program.

Although Paradyn is a working system, many directions remain for growth. Over the next few years, we will be expanding to new machine environments, new high-level languages, and new problem domains. ■

Acknowledgments

We thank the authors of the applications used in our study, Gary Lewandowski (graph coloring) and Spyros Kontogiorgis (Msolv).

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under Grant F33615-94-1-1525 (ARPA Order No. B550), NSF Grants CCR-9100968 and CDA-9024618, Department of Energy Grant DE-FG02-93ER25176, and Office of Naval Research Grant N00014-89-J-1222. Hollingsworth was supported in part by an ARPA graduate fellowship in high-performance computing.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the US government.

References

1. *Quantify User's Guide*, Pure Software Inc., 1993.
2. J.R. Larus and T. Ball, "Rewriting Executable Files to Measure Program Behavior," *Software—Practice & Experience*, Vol. 24, No. 2, Feb. 1994, pp. 197-218.
3. D.A. Reed et al., "Scalable Performance Analysis: The Pablo Performance Analysis Environment," *Proc. IEEE Scalable Parallel Libraries Conf.*, IEEE Service Center, Piscataway, N.J., 1993.
4. J. Yan, S. Sarukkai, and P. Mehra, "Performance Measurement Visualization and Modeling of Parallel and Distributed Programs Using the AIMS Toolkit," *Software—Practice & Experience*, Vol. 25, No. 5, Apr. 1995, pp. 429-461.
5. B.P. Miller et al., "The Second Generation of a Parallel Program Measurement System," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 2, Apr. 1990, pp. 206-217.
6. W. Williams, T. Hoel, and D. Pase, "The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D," in *Programming Environments for Massively Parallel Distributed Systems*, K.M. Decker and R.M. Rehmann, eds., Birkaeuser Verlag, Basel, Switzerland, 1994.
7. J.K. Hollingsworth, R.B. Irvin, and B.P. Miller, "Integration of Application and System Based Metrics in a Parallel Program Performance Tool," *ACM SIGPlan Notices Symp. Principles and Practice of Parallel Programming*, ACM, New York, 1991, pp. 189-200.
8. J.K. Hollingsworth, B.P. Miller, and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," *Proc. Scalable High-Performance Computing Conf.*, Knoxville, Tenn., 1994, pp. 841-850.
9. J.K. Hollingsworth and B.P. Miller, "Dynamic Control of Performance Monitoring on Large-Scale Parallel Systems," *Proc. Seventh ACM Int'l Conf. Supercomputing*, ACM, New York, 1993, pp. 185-194.
10. C. Upson et al., "The Application Visualization System: A Computational Environment for Scientific Visualization." *IEEE Computer Graphics and Applications*, Vol. 9, No. 4, July 1989, pp. 30-42.
11. M.T. Heath and J.A. Etheridge, "Visualizing Performance of Parallel Programs," *IEEE Software*, Vol. 8, No. 5, Sept. 1991, pp. 29-39.

Barton P. Miller *is a professor in the Computer Sciences Department at the University of Wisconsin, Madison. He received a BA from the University of California, San Diego, and an MS and PhD from the University of California, Berkeley; all three degrees are in computer science.*

Mark D. Callaghan *is a working toward a doctoral degree at the University of Wisconsin, Madison. He received a BS in computer science from San Diego State University in 1992. He is a student member of the ACM.*

Jonathan M. Cargille *is an associate researcher in the Computer Sciences Department at the University of Wisconsin, Madison. He received BS and MS degrees in computer science from the University of Wisconsin, Madison, and the University of Southwestern Louisiana, respectively.*

Jeffrey K. Hollingsworth *is an assistant professor in the Computer Science Department at the University of Maryland, College Park. He received a BSEE from the University of California, Berkeley, and MS and PhD degrees in computer science from the University of Wisconsin.*

R. Bruce Irvin *is a member of the technical staff at Informix. He received a BS degree in computer science from Carnegie Mellon University and MS and PhD degrees from the University of Wisconsin.*

Karen L. Karavanic *is a doctoral student at the University of Wisconsin, Madison. She received BA and MS degrees in computer science from New York University and the University of Wisconsin, respectively. She is a member of ACM.*

Krishna Kunchithapadam *is working toward a doctoral degree at the University of Wisconsin, Madison. He received a bachelor of technology in computer science from the Indian Institute of Technology, Madras, and an MS in computer science from the University of Wisconsin, Madison.*

Tia Newhall *is working toward a doctoral degree at the University of Wisconsin, Madison, where she received a BS in mathematics and an MS in computer science.*

The authors can be contacted at the Computer Sciences Dept., University of Wisconsin, Madison, 1210 W. Dayton Street, Madison, WI 53706; e-mail {bart, markc, jon, karavan, krishna, newhall}@cs.wisc.edu, hollings@cs.umd.edu, and rbi@informix.com.