Integrating Database Technology with Comparison-based Parallel Performance Diagnosis: The PerfTrack Performance Experiment Management Tool[1]

Karen L. Karavanic[2], John May[3], Kathryn Mohror[2,4], Brian Miller[3], Kevin Huck [4,5], Rashawn Knapp[2], Brian Pugh[2]

[2] Portland State University
Dept. of Computer Science
Portland, OR 97207-0751
{karavan, kathryn}@ cs.pdx.edu
{knappr, bpugh} @ cs.pdx.edu

[3]Lawrence Livermore National Laboratory
Livermore, CA 94551
{may10, bmiller} @ llnl.gov

[5]University of Oregon
Computer and Information Science
Eugene, OR 97403-1202
khuck@cs.uoregon.edu

Abstract

PerfTrack is a data store and interface for managing performance data from large-scale parallel applications. Data collected in different locations and formats can be compared and viewed in a single performance analysis session. The underlying data store used in PerfTrack is implemented with a database management system (DBMS). PerfTrack includes interfaces to the data store and scripts for automatically collecting data describing each experiment, such as build and platform details. We have implemented a prototype of PerfTrack that can use Oracle or PostgreSQL for the data store. We demonstrate the prototype's functionality with three case studies: one is a comparative study of an ASC purple benchmark on high-end Linux and AIX platforms; the second is a parameter study conducted at Lawrence Livermore National Laboratory (LLNL) on two high end platforms, a 128 node cluster of IBM Power 4 processors and BlueGene/L; the third demonstrates incorporating performance data from the Paradyn Parallel Performance Tool into an existing PerfTrack data store.

1. Introduction

Effective performance tuning of large scale parallel and distributed applications on current and future clusters, supercomputers, and grids requires the ability to integrate performance data gathered with a variety of monitoring tools, stored in different formats, and possibly residing in geographically separate data stores. Performance data sharing between different performance studies or scientists is currently done manually or not done at all. Manual transfer of disorganized files with unique and varying storage formats is a time consuming and error prone approach. The granularity of exchange is often entire data sets, even if only a small subset of the transferred data is actually needed. All of these problems worsen as high end systems continue to scale up, greatly increasing the size of the data sets generated by performance studies. The overhead for sharing data discourages collaboration and data reuse. There are several key challenges associated with performance data sharing. First, performance tool output is not uniform, requiring the ability to translate between different metrics and tool outputs to compare the datasets. Second, the data sets resulting from performance studies of tera- and peta-scale level applications are potentially large and raise challenging scalability issues. For some types of summary data, database storage is a well-understood task; for others, such as trace data and data generated with a dynamic instrumentation based tool, further research is necessary to develop an efficient and flexible representation. In addition to increased ease of collaboration between scientists using or studying a common application, the potential benefits of a tool to collect

[1] This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract number W-7405-Eng-48, document number UCRL-CONF-211654.

[4] This work was completed in part while the author was a member of the staff of UC/LLNL.

and integrate performance data include the ability to analyze performance of an application across architectures, and the ability to automatically compare different application versions.

To address this problem we have designed and implemented an experiment management tool for collecting and analyzing parallel performance data called PerfTrack. PerfTrack comprises a data store and interface for the storage and retrieval of performance data describing the runtime behavior of large-scale parallel applications. PerfTrack uses an extensible resource type system that allows performance data stored in different formats to be integrated, stored, and used in a single performance analysis session. To address scalability, robustness, and fault-tolerance, PerfTrack uses a database management system for the underlying data store. It includes interfaces to the data store plus a set of modules for automatically collecting descriptive data for each performance experiment, including the build and runtime platforms. We present a model for representing parallel performance data, and an API for loading it.

This work is part of the DOE's APOMP (ASC Performance Optimization and Modeling Project) effort. The initial target audience for the system is LLNL scientists developing production codes for ASC platforms. This target environment presents a number of challenges, including: the use of multiple platforms; a fairly rapid cycle of phasing in new machines; the use of a variety of languages and compilers; the use of a variety of concurrency models, including MPI, OpenMP, and mixed MPI/OpenMP; security concerns; and a level of code complexity that requires code development and tuning to be performed by a team of scientists, as opposed to a single individual. Even a single application may be studied by a dozen scientists with independent subgoals and methods; gathering the data to be used together necessitates solving key challenges related to heterogeneous data. There are also scalability challenges: it is anticipated that a production use data store will be quite large. In order to have a complete picture of the performance of an application and the factors that contribute to its performance, it is necessary to collect a large variety of runtime information. This information includes details about system software, runtime libraries, and the hardware environment. While all of these details are important to program performance, which of these details will play a significant role in diagnosing the performance of an application remains an open question. Therefore, our initial approach is to collect as much information as possible.

In the following section we present the PerfTrack design, detailing a model for representing parallel performance data. In Section 3 we describe our PerfTrack implementation. In Section 4 we detail results from using PerfTrack in three case studies, involving different applications, hardware and software platforms, and performance tools. A discussion of related work is presented in Section 5. Finally we detail our plans for future work and conclude.

2. The PerfTrack Model

In this section we detail the key design decisions and semantic concepts of PerfTrack. Our goal is a flexible and extensible tool for storing and navigating a large amount of performance data generated by a wide range of tools for a variety of purposes. This goal has driven our design choices to maximize flexibility and extensibility.

2.1 Resources

A central concept to PerfTrack is the *resource*. A resource is any named element of an application or its compile-time or runtime environment. Examples include machine nodes, processes, functions, and compilers. A *context* is a collection of resources that describes everything known about a performance measurement.

A *resource type* describes what kind of information the resource represents. For example, "Linux" and "AIX" are resource names of type "operating system." Some resource types fall into natural hierarchies: A "processor" is part of a "node," which may be in a "partition" of a "machine," which may itself be part of a "grid." A hierarchy of resources forms a tree, with the most general resource type at the root (e.g., "grid"). Hierarchical resource types are written in the style of Unix path names, such as "grid/machine/partition/node/processor." Likewise, a full resource name is written: "/SingleMachineFrost/Frost/batch/frost121/p0." A name written in this form specifies a resource and all its ancestors, and PerfTrack requires that full resource names be unique. It is sometimes convenient to refer to a collection of resources by a base name, such as "batch." This shorthand refers to the batch partition of any machine. Types that do not fall into hierarchies are considered to be single-level hierarchies, for example "application" and "/Linpack." A set of base resource types is defined as part of system initialization. PerfTrack's resource type system is extensible: users may add new hierarchies or new types within the base hierarchies.

In addition to names and types, resources can also have *attributes*, which are characteristics of resources. A resource of type "processor" might have the attributes "vendor," "processor type," and "clock MHz," with corresponding attribute values "IBM," "Power3," and "375." Attributes may be resource names, that is, one resource can be an attribute of another. For example, a compiler may be an attribute of a particular build.

2.2 Performance Results

A performance result is a measured or calculated value, plus descriptive metadata. The value itself may be a scalar or more complex type. The metadata comprises a *metric* and one or more *contexts*. A metric is a measurable characteristic of a program. Examples of metrics include CPU time, I/O wait time, or number of messages sent. A context is the set of resources that defines the specific part(s) of the code or environment included in the measurement. For example, a measurement might include one process out of 64, or one particular function, or one particular timestep. One performance result can have more than one context, and a single context can apply to multiple performance results. Associating multiple contexts with a single performance result is useful when a measurement spans processors or other resources of the same type, such as the transit time of a message between two processes. Multiple performance results with the same context make sense when several values are measured at the same time in the same part of a program. For example, execution time and floating-point instruction count might both be measured for the same entire execution. The two performance results would have identical contexts but different metrics. This example motivates our design decision to separate metrics from contexts; if metrics were allowed in contexts, these two performance results would have different contexts.

A *pr- filter* is used to find one or more performance results of interest. The result of applying a pr-filter to a set of performance results is a subset of those performance results. A pr-filter is a set of *sets* of resources. We call each set a *resource family,* because all the resources in the set belong to the same type hierarchy.

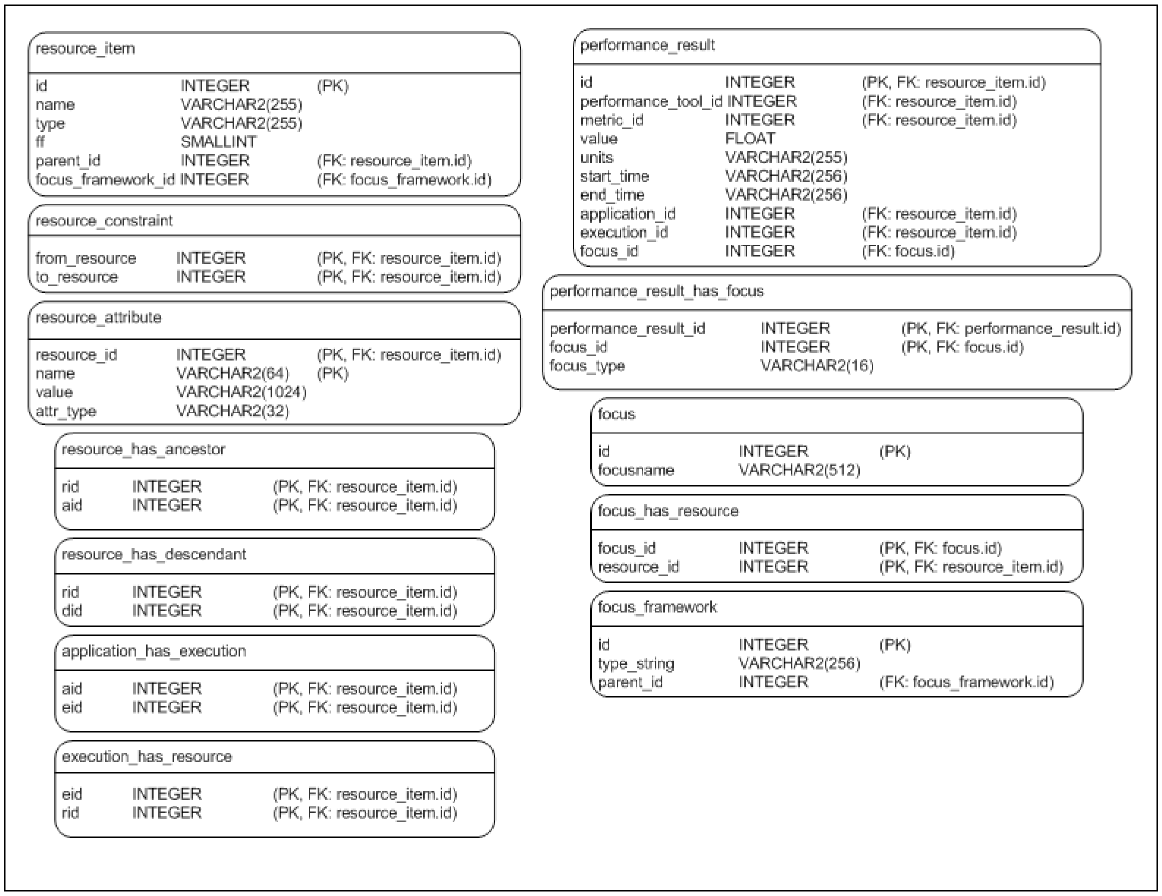
Applying a pr-filter $PRF = \{ R_1, R_2, \ldots R_n \}$ to an initial set of performance results $PR = \{pr_1, pr_2, \ldots, pr_m \}$ yields a subset of performance results PR'. Each performance result in *PR'* has a context that matches the pr-filter. A pr-filter *PRF* matches a context *C* if every resource family R_i in *PRF* contains a resource in *C*, or:

$$PRF \text{ matches } C \Leftrightarrow \forall\, R \in PRF:\ \exists\, r \in C \ni r \in R$$

Each resource family in a pr-filter is formed by applying a *resource filter* to a set of resources. A resource filter contains a resource type, a resource name, or a list of attribute-value-comparator tuples. In the first case, applying the filter simply selects resources with the specified type (for example, all results measured at the level of the whole machine); in the second case, it selects resources with the specified name, and in the third case, it selects resources that contain all of the listed attributes, and for which applying the given comparator to the attribute's value results in a true value. Each resource filter also contains an ancestor/descendant flag, which extends the resulting resource family to include the ancestors, descendants, or both, of each member resource. For example, a job on a parallel machine might record data for measurements on individual processors named "/SingleMachineFrost/Frost/batch/node1/p0," "/SingleMachineFrost/Frost/batch/node1/p1," and so on. A resource filter that includes the resource name "/SingleMachineFrost/Frost/node1" and a descendant flag would yield the set of all processor resources for node 1.

3. Prototype Implementation

We have completed a prototype implementation of PerfTrack that closely follows the model described in Section 2. Our current implementation follows the model but does not yet implement the full range of options described in the model. For example, we currently store only scalar values for performance results and not complex types. Also, resource attributes that are themselves resources are stored as "resource constraints" in a separate table. The underlying data store uses a relational database management system. PerfTrack includes both a GUI and a script-based interface to the data store. In the remainder of this section we discuss the database internals and each of the two interfaces.



resource_item

id	INTEGER	(PK)
name	VARCHAR2(255)	
type	VARCHAR2(255)	
ff	SMALLINT	
parent_id	INTEGER	(FK: resource_item.id)
focus_framework_id	INTEGER	(FK: focus_framework.id)

resource_constraint

from_resource	INTEGER	(PK, FK: resource_item.id)
to_resource	INTEGER	(PK, FK: resource_item.id)

resource_attribute

resource_id	INTEGER	(PK, FK: resource_item.id)
name	VARCHAR2(64)	(PK)
value	VARCHAR2(1024)	
attr_type	VARCHAR2(32)	

resource_has_ancestor

rid	INTEGER	(PK, FK: resource_item.id)
aid	INTEGER	(PK, FK: resource_item.id)

resource_has_descendant

rid	INTEGER	(PK, FK: resource_item.id)
did	INTEGER	(PK, FK: resource_item.id)

application_has_execution

aid	INTEGER	(PK, FK: resource_item.id)
eid	INTEGER	(PK, FK: resource_item.id)

execution_has_resource

eid	INTEGER	(PK, FK: resource_item.id)
rid	INTEGER	(PK, FK: resource_item.id)

performance_result

id	INTEGER	(PK, FK: resource_item.id)
performance_tool_id	INTEGER	(FK: resource_item.id)
metric_id	INTEGER	(FK: resource_item.id)
value	FLOAT	
units	VARCHAR2(255)	
start_time	VARCHAR2(256)	
end_time	VARCHAR2(256)	
application_id	INTEGER	(FK: resource_item.id)
execution_id	INTEGER	(FK: resource_item.id)
focus_id	INTEGER	(FK: focus.id)

performance_result_has_focus

performance_result_id	INTEGER	(PK, FK: performance_result.id)
focus_id	INTEGER	(PK, FK: focus.id)
focus_type	VARCHAR2(16)	

focus

id	INTEGER	(PK)
focusname	VARCHAR2(512)	

focus_has_resource

focus_id	INTEGER	(PK, FK: focus.id)
resource_id	INTEGER	(PK, FK: resource_item.id)

focus_framework

id	INTEGER	(PK)
type_string	VARCHAR2(256)	
parent_id	INTEGER	(FK: focus_framework.id)

Figure 1: PerfTrack database schema. *Each text box details one table in the database, with the table name listed at the top. Below the name, the table fields are listed. Field names labeled "PK" are primary keys for the table (the field is used as an index, and no duplicates are allowed), and field names labeled FK are foreign keys for the table indicated.*

3.1 Database Internals

The complete database schema for PerfTrack's underlying data store is shown in Figure 1. We can see the relationship of the schema to the model by examining the resource_item and performance_result tables. Information for each resource is stored in the resource_item table: the name, type, and parent for the resource, plus an internal identifier called the focus_framework_id that is used to represent the resource type. Each resource can have some number of resource attributes, and for each attribute an entry is stored into the resource_attribute table. Each entry in the resource_constraint table has two resource_item references, and defines a resource attribute that is itself a resource. For example, if process 8 runs on node 16, we would add an entry to resource_constraint containing the resources for process 8 and node 16. Each performance result is stored in a record of the performance_result table: the metric measured, the performance tool used, the value, etc. Each performance result context is called the "focus" in the internal database schema. Therefore, there is a one to many relationship between the focus_has_resource and resource_item tables; the latter contains one record for each resource in the context. The focus or context type can be one of primary, parent, child, sender or receiver. Several of the tables shown in the schema have been added for performance reasons, for example, the resource_has_ancestor and resource_has_descendant tables are used to avoid needing to traverse the resource hierarchy and follow the chain of "parent_id"'s to compute the answer.

PerfTrack uses a set of base resource types, shown in Figure 2. This set includes five resource hierarchies: `build`, `grid`, `environment`, `execution`, and `time`; plus eight non-hierarchical resource types : `application`, `compiler`, `preprocessor`, `inputDeck`, `submission`, `operatingSystem`, and `metric`. The Build hierarchy (`build/module/function/codeBlock`) specifies the location in the code where the measurement took place, to the level of detail that is known. The Grid hierarchy (`grid/machine/partition/node/processor`) specifies the hardware used in the measured application run. The Environment hierarchy (`environment/module/function/codeBlock`) specifies the runtime environment for the application, including dynamic libraries. The Execution hierarchy (`execution/process/thread`) includes the specific application process information, down to the thread information, if available. The Time hierarchy (`time/interval`) indicates the phase of execution in which the measurement took place. For example, a given application might have three main phases of execution: initialization, main calculation and cleanup/termination. The resource type hierarchy is designed to be flexible and extensible. For example, a researcher might gather data with timesteps subdivided into specific calculation phases. The resource type hierarchy could easily be extended by adding another level to the Time hierarchy, with the Interval type as its parent. Similarly, an analyst wishes to add a brand new resource hierarchy can do so by adding a new top-level (with the root as its parent) entry, and additional child types. In fact, PerfTrack uses the type extension interface to load the initial set of base types when a new database is initialized.

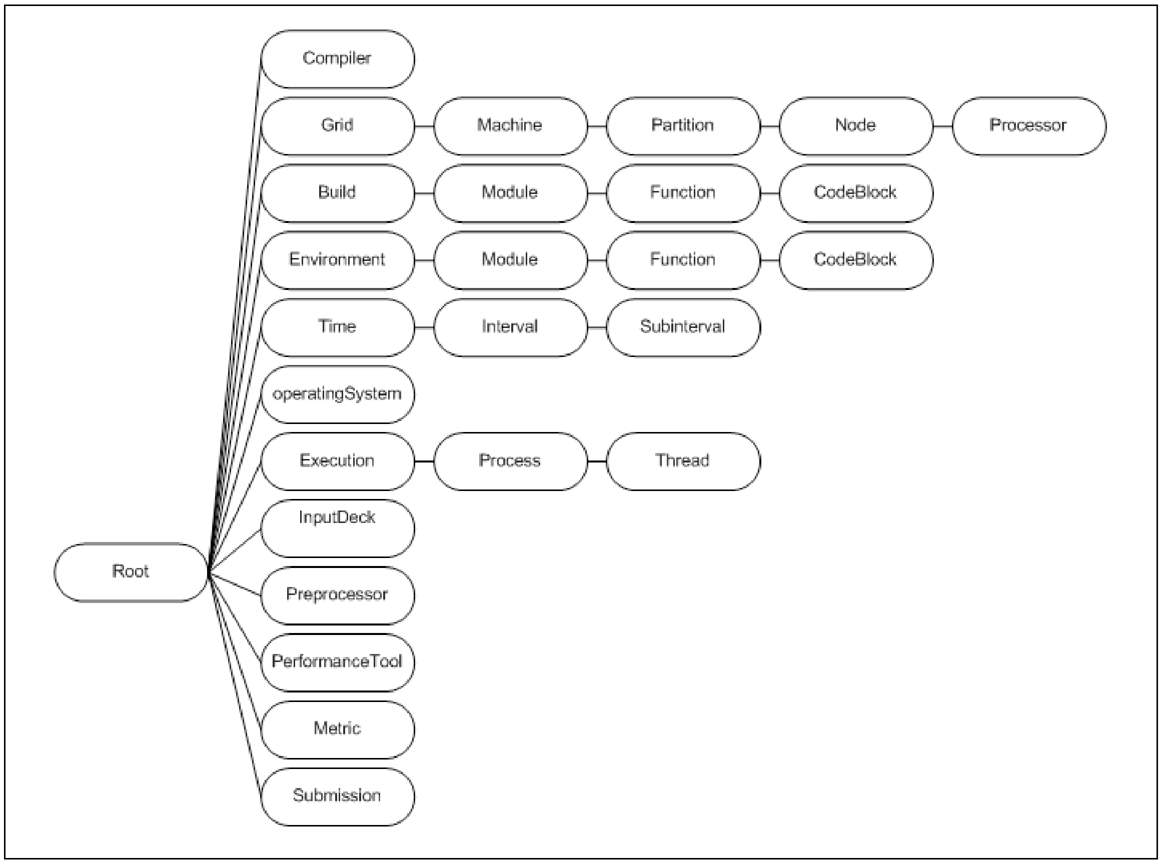


Figure 2: PerfTrack base resource types. *The figure shows hierarchical resource types, such as build and execution, and non-hierarchical resource types, such as Performance Tool.*

3.2 The PerfTrack GUI

PerfTrack users can find and view performance data using a comprehensive graphical user interface (GUI). The GUI can issue queries based on a combination of resources and attributes. It presents data in tabular form, and it can also draw bar charts and store data in a format suitable for spreadsheet programs to import.

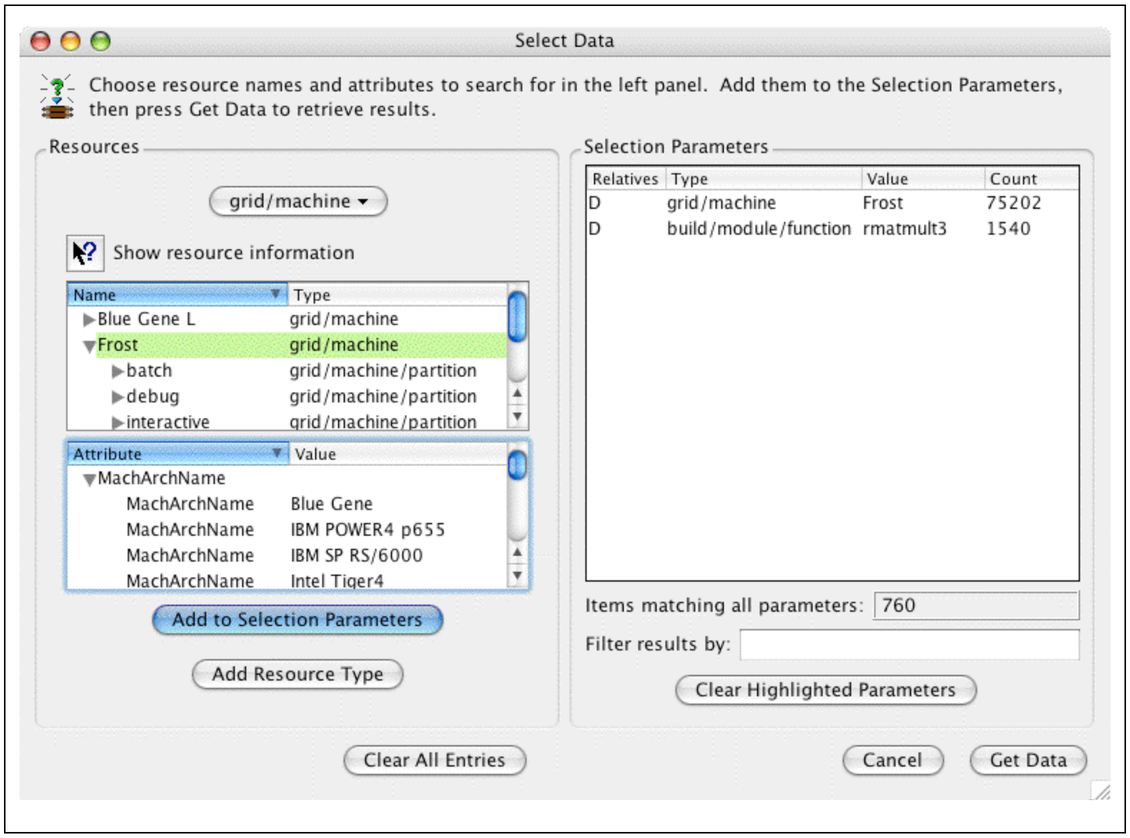
The GUI begins a session by establishing a database connection and presenting a selection dialog window (Figure 3). To build a query, the user first selects a resource type from a menu. The GUI gathers from the database all resources names and attributes with the corresponding type and presents them in two boxes on the left side of the dialog. Each resource name and each attribute in the list represents a set of resources. For hierarchical resources, the user can click a resource name to find its child resources. The same resource could be found by choosing its type directly from the resource menu, but the two different ways of selecting a child resource have different meanings. When a resource name is selected from the top level of the list, it corresponds to a resource set that includes all resources with that base name (and type). Choosing a resource name as a child of another resource name restricts the subset to include only resources that are also children of resources with the named parent. For example, in Figure 3 the "batch" resource shown as a child of "Frost" represents resources whose full names end with "Frost/batch." If the "partition" resource type were selected directly from the resource menu, the "batch" entry would refer to results from batch partitions on any machine.

To see all the attributes and their values for a particular resource, the user can click a button to open a separate viewer (not shown here) and then click on an entry in the resource list. If the entry refers to multiple resources (such as batch partitions on multiple machines), the attribute viewer displays a menu from which the user can choose a resource to examine.

To add resources to a resource family, the user highlights a resource name or attribute value and then clicks a button to add it to a pr-filter, which appears on the right side of the dialog. The user can then select a different resource type from the menu and choose a resource name or attribute value from that list. Users can also add a resource type to the query list without specifying a name or attribute. This is equivalent to specifying a resource family that contains all the resources of that type, with no ancestors or descendants included. A user might do this to get only those results that are machine-level measurements, excluding processor-level or node-level data.

As resource families are added to a pr-filter, the GUI determines how many performance results in the database match each resource family by itself and how many match the entire pr-filter. This lets users tailor queries to return a reasonable number of results.

By default, the resource family corresponding to a selected resource name includes not only the resources with that name but also all of their descendants. For example, choosing the resource "Frost" defines a resource subset that also includes Frost's partitions, all of their nodes, and all of their processors. This lets users quickly choose related sets of performance results, without requiring them to name or know all the descendant resources. The GUI indicates this default choice with the letter *D* in the "Relatives" column of the "Selected Parameters" list. The user can change this value to select a resource's ancestors (by changing the *D* to an *A*), both ancestors and descendants (*B*) or neither (*N*).



Choose resource names and attributes to search for in the left panel. Add them to the Selection Parameters, then press Get Data to retrieve results.

Name	Type
▶Blue Gene L	grid/machine
▼Frost	grid/machine
▶batch	grid/machine/partition
▶debug	grid/machine/partition
▶interactive	grid/machine/partition

Attribute	Value
▼MachArchName	
MachArchName	Blue Gene
MachArchName	IBM POWER4 p655
MachArchName	IBM SP RS/6000
MachArchName	Intel Tiger4

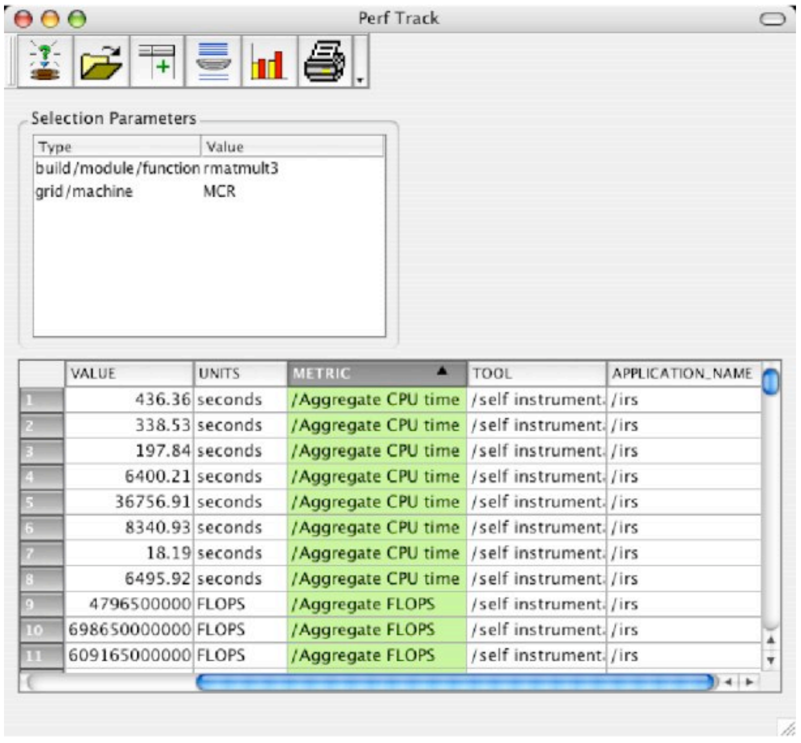
Relatives	Type	Value	Count
D	grid/machine	Frost	75202
D	build/module/function	rmatmult3	1540

Items matching all parameters: 760

Figure 3: Selection dialog for the PerfTrack GUI. *The user selects a resource type from a popup menu and then chooses resource names and attributes from the lists on the left to build up a query. The number of performance results that the query would retrieve appears in a box on the right.*

Once the user has chosen a combination of resource sets, pressing another button causes the GUI to retrieve the data. The GUI presents this data in a table in the main window (Figure 4). When PerfTrack retrieves a set of performance results, any resources in the corresponding context that the user didn't specify in the query constitute potentially interesting descriptive information. We call these *free resources* because their values were not defined by the pr-filter. In many cases, free resources will be the independent variables in a performance experiment, and the result values will be the dependent variables. This table initially shows the result data but only a few of the free resources describing the results. Since each result could have dozens of free resources, it would not be sensible (or efficient) to show all the free resources and their attributes for each result in the table. Instead, the user selects the resources and attributes to display in a second step. Choosing the "Add Columns" command from the main window brings up another dialog that lists the free resources and their attributes for the data listed in the table. The resource types listed include only those whose names are *not identical* for all the listed results. For example, if all the selected results came from applications run on Linux, the resource type "operating system" would not be shown. As the user chooses resources and attributes to display, the GUI adds columns to the table, and pressing a button retrieves the requested data for all the results.

Requiring users to gather data in two steps may initially seem awkward. Early prototypes of the GUI allowed users to choose the resources and attributes to display at the same time as they specified the pr-filter. The problem with this approach was that it was difficult to tell in advance which resource would give interesting information about the results, unless one was already familiar with the dataset. By delaying the selection of resource types until after it retrieves the data, the GUI can help guide the user toward the most useful information. Once the necessary data has been displayed, users can filter it based on various criteria to hide some of the entries, sort the data by any column, plot the data in bar charts (see Figure 5), print the charts, store the data to files, read it back in, and initiate new queries.



Type	Value
build/module/function	rmatmult3
grid/machine	MCR

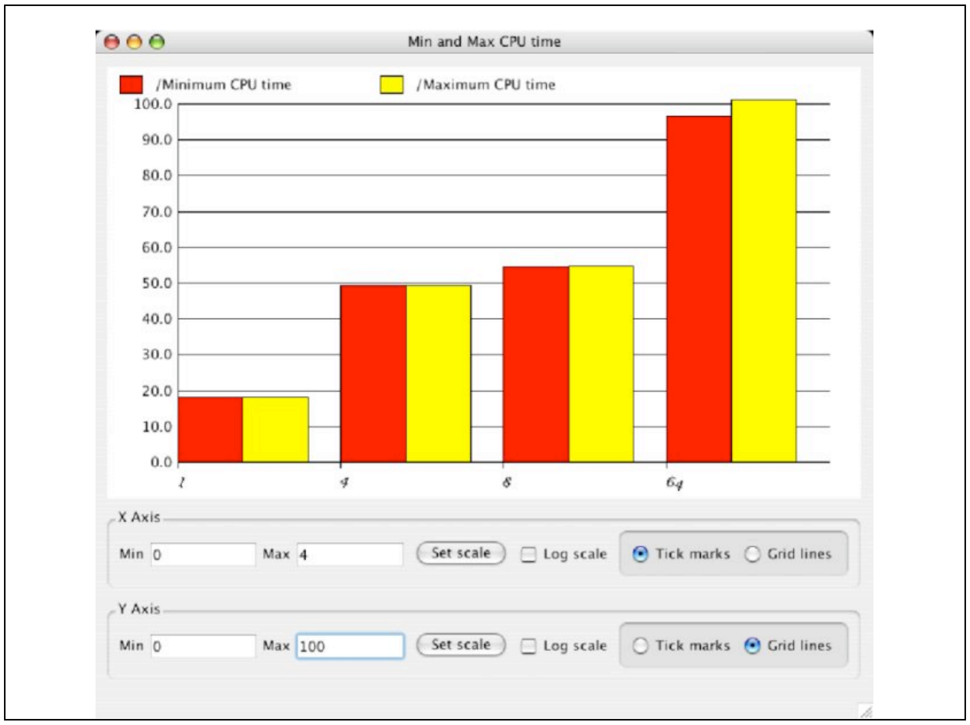
	VALUE	UNITS	METRIC ▲	TOOL	APPLICATION_NAME
1	436.36	seconds	/Aggregate CPU time	/self instrument	/irs
2	338.53	seconds	/Aggregate CPU time	/self instrument	/irs
3	197.84	seconds	/Aggregate CPU time	/self instrument	/irs
4	6400.21	seconds	/Aggregate CPU time	/self instrument	/irs
5	36756.91	seconds	/Aggregate CPU time	/self instrument	/irs
6	8340.93	seconds	/Aggregate CPU time	/self instrument	/irs
7	18.19	seconds	/Aggregate CPU time	/self instrument	/irs
8	6495.92	seconds	/Aggregate CPU time	/self instrument	/irs
9	4796500000	FLOPS	/Aggregate FLOPS	/self instrument	/irs
10	698650000000	FLOPS	/Aggregate FLOPS	/self instrument	/irs
11	609165000000	FLOPS	/Aggregate FLOPS	/self instrument	/irs

Figure 4: PerfTrack GUI main window. *Selected performance results appear in tabular form. The user can sort by any column, filter the data based on various parameters, show additional data on the selected results, and plot, print, or store the data.*

3.2 GUI implementation details

The GUI is implemented in C++ using the Qt GUI toolkit [1]. We chose Qt because it offers a rich set of widgets that run efficiently on a variety of platforms. It also has classes that can communicate directly with various database management systems. To minimize the delays involved in communicating with the database, we have designed the GUI to retrieve information incrementally and only on demand. For example, the GUI does not get the resource names or attribute types until the user selects a resource type, and it doesn't get the list of a resource's children until the user clicks a resource name to display this list.

The barchart display was written from scratch by one of the authors for another tool and adapted for this application. We anticipate that users will want to plot data in a variety of ways. We intend to support the most used functionality directly in the GUI. For more sophisticated analyses, users can always export the data to a spreadsheet or other graphing tool. Although sophisticated third-party graphing tools are available for Qt programs, they are not free, so we chose not to use one in PerfTrack.



X Axis
Min 0 Max 4 ☐ Log scale [✓ Tick marks] Grid lines

Y Axis
Min 0 Max 100 ☐ Log scale Tick marks [✓ Grid lines]

Figure 5: Users can plot selected data from the main window in a bar chart. *Multiple series of values can appear on the same chart, as shown here. This data shows the minimum and maximum running time of a function across all the processors for different process counts, which is a rough indication of load balance.*

3.3 The PerfTrack script interface

In addition to the GUI, PerfTrack includes a script-based database interface, implemented using python [2]. We chose python for rapid prototyping and researching of new PerfTrack functionality. The Python DBI API 2.0 specifies a standard database connection and cursor interface, and a variety of database interface modules have been developed. For our prototype we have implemented two options: an Oracle-based data store with the cx_Oracle [3] module, and a PostgreSQL-based data store with the pyGreSQL 3.6.2 [4] module. PerfTrack's script interface includes functionality for data collection, data loading, and data queries. In this section we describe each of these parts of the interface in greater detail.

PerfTrack includes scripts for automatic capture of build- and runtime- related information, and for collection and translation of performance data. The build information collected fits into two categories: build environment information and compilation information. The build environment information includes the build operating system and machine details, and the environment settings in the build user's shell. We record details about the operating system, such as name, version, and revision. We record the name of the machine or node on which the build takes place. The compilation information includes the compilers used, their versions, the compilation flags used and details about the static libraries that were linked in. In the case that the compiler is an MPI wrapper script, we attempt to gather the compiler used by the wrapper script as well as the flags and libraries used by the wrapper script. The runtime data collected includes details about system software, runtime libraries, and the hardware environment.

To initiate automatic build capture, the user runs a PerfTrack wrapper script to execute their build, supplying the makefile name, arguments, etc., as arguments to the wrapper script. The PerfTrack script executes "make" and captures the output, using it to generate a build data file. This file is then converted to a PTdf file during the data load step. To capture the runtime environment, the user runs a PerfTrack run script. The output of this script is a file containing a variety of data about the execution and its environment, including: environment variables, number of processes, runtime libraries used, and the input deck name and timestamp. The collected information is stored into PerfTrack in the form of resource hierarchies of base type "build," "environment," and "execution," and resources of type `compiler, preprocessor, inputDeck, submission, and operatingSystem,` by storing a resource attribute for each piece of data. Example attributes of a library resource are the version, size, type (e.g., MPI or thread library), and timestamp of the library.

The number of files of performance data and their format will differ for different benchmarks and performance tools. PerfTrack provides support to tool users for converting their original performance tool or benchmark output into PTdf format. First, the script-based interface provides a range of lookup methods that can be used to check the existing contents of the database, and to generate unique resource names. Second, we include a script for converting a variety of benchmark formats into PTdf, as a model in constructing a new script for a different type of performance data. One of the goals for PerfTrack is to be inclusive of all types of performance data, and new performance tools and benchmarks are of course continually being written, therefore, we feel providing conversion support is the most useful way to keep PerfTrack useful to the widest range of users. PerfTrack includes a "PTdfGen" script to generate PTdf for a directory full of files. The user creates an index file, containing a list of entries, one per execution. Each entry lists the execution name, application name, concurrency model, number of processes, number of threads, and timestamps for the build and run. PerfTrack generates PTdf files for the executions listed in one index file.

PerfTrack data format (PTdf) defines an interface used in data loading (see Figure 6). This interface splits the functionality of the script-based interface into two well-defined pieces. PTdf files are used to define all resources and resource types in PerfTrack. This interface is used to initialize PerfTrack with a set of base resource types and default machine definitions. Once users have converted their raw data into PTdf, it is loaded through the PTdataStore interface. The PTdataStore class provides a python interface to PerfTrack's underlying database.

```
Application appName
ResourceType resourceTypeName

Execution execName appName

Resource resourceName resourceTypeName execName
Resource resourceName resourceTypeName

ResourceAttribute resourceName attributeName attributeValue attributeType
PerfResult execName resourceSet perfToolName metricName value units
ResourceConstraint resourceName1 resourceName2
```

Figure 6: PTdataFormat API. *The PTdataFormat interface is used to load data into PerfTrack. attributeType is currently one of two possible values: string or resource. Adding a resourceConstraint is equivalent to adding an attribute of type resource. The attributeType field is partly a placeholder for future, when we will have a richer set of types available for attributes. A resourceSet is one or more lists of resource names separated by a colon; each list consists of a comma separated list of resource names followed by a resource set type name in parentheses.*

The PTdataStore interface also includes methods for querying PerfTrack data. The functionality includes requesting information about resources and their attributes, details of individual executions, and performance results. The user may request one of several simple reports.

4. Results

In order to evaluate the suitability of PerfTrack for use in performance experimentation, we have conducted two case studies using data sets from actual performance analysis projects. In this section we present the results and discuss the lessons learned from full-scale use of our PerfTrack prototype.

4.1 A Purple Benchmark Study

The goal of this study was to demonstrate our ability to collect, store, and navigate a full set of performance data from high end systems using PerfTrack. We tested out our initial PerfTrack prototype implementation with a study of an ASC Purple Benchmark. The ASC Purple benchmarks are published by the U.S. Department of Energy as representative of challenging codes of interest to their scientists. For this study we included the Implicit Radiation Solver (IRS) benchmark. The benchmark code is written in C, and uses MPI, OpenMP, both, or neither (sequential). We ran IRS on two platforms, MCR (a Linux cluster) and Frost (an AIX cluster).

First, the application was built using the PerfTrack PTbuild script. This automatically collects information about the build environment into a file. Next, the application was started using PerfTrack run script, supplying the build file as an argument. Note that for this study, a full set of descriptive machine data was already in our PerfTrack system, from previous studies, so no further collection or entry of machine description was required. Each standard IRS benchmark outputs several data files for each application run. IRS outputs performance data for the whole program, with the values cumulative over all processes. The data includes timings for approximately 80 different functions in the program. For each function, the aggregate, average, max and min values for five different metrics are reported. Sometimes one of the values or metrics doesn't apply, so there are slightly varying numbers of performance results for each IRS execution. In our runs, each IRS execution generated approximately 1000 performance results.

We converted all of the data into PTdataFormat files, then loaded it into PerfTrack using the script interface. Detailed information about the data sizes is shown in Table 1. We used the PerfTrack GUI to examine the data in several combinations. Finally, we output a dataset of interest into a text file, input it into an OpenOffice spreadsheet, analyzed it and plotted a graph. Using PerfTrack, we were able to load and navigate the complete set of performance data for the Purple Benchmark study.

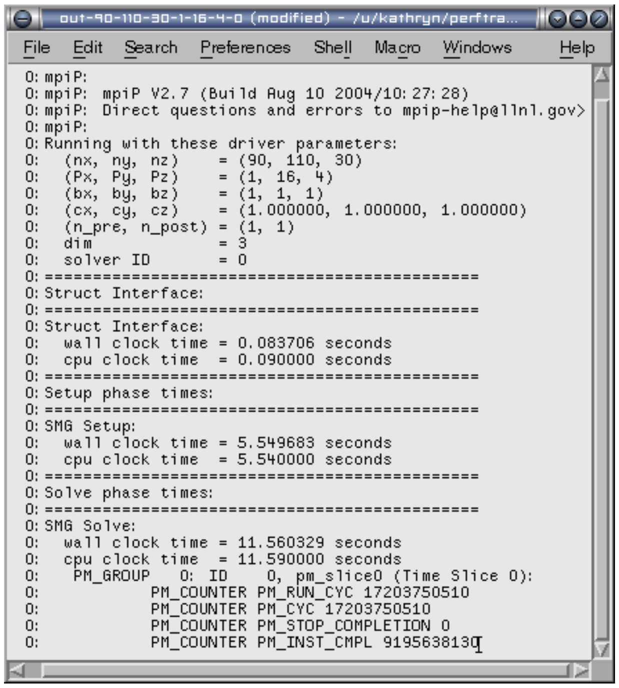
Table 1: Statistics for raw data, PTdf, and data store.

Original Data Set (Per Execution)			**PTdf (Per Execution)**			**PTdf (Total)**	**PerfTrack (Total)**	
Name	**Files**	**Raw Data (~bytes)**	**Resources**	**Metrics**	**Performance Results**	**# Files /size (lines)**	**Exec.s Loaded**	**Approx. DB size increase**
IRS	6	61,100	280	25	1,514	62 / 2,298	62	12 MB
SMG-UV	2	190,800	5,657	259	9,777	247 / 16,056	35	89 MB
SMG-BG/L	1	1,000	522	8	8	1 / 156,274	60	27 MB

4.2 A Noise Analysis Study

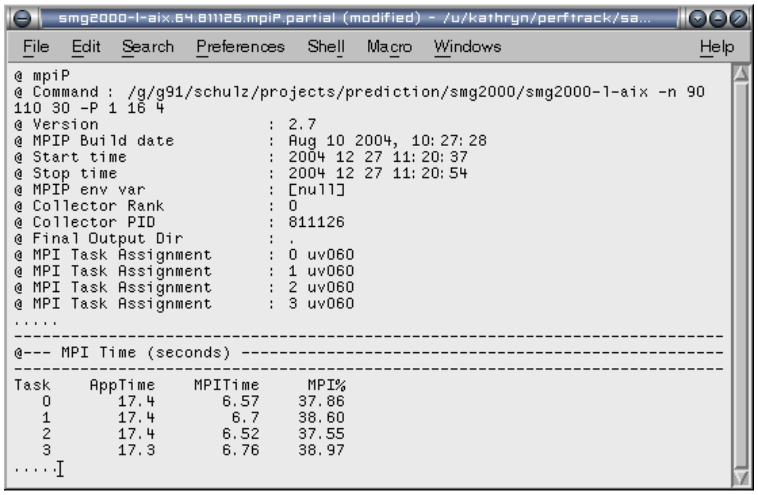
The goal of this study was to test PerfTrack with data that included new types of machines, benchmark data, and performance tool data; and to test its scalability with a large real world data set. In this study we started with a copy of the data from a previously conducted noise analysis study of some of LLNL's production systems [5], that investigated the results of a novel approach for performance prediction using neural networks. Of the total set of study data, we included a subset from measurements on two platforms: BlueGene/L (BG/L) and UV; of the SMG2000 ASC Purple Benchmark. The performance data included results from the standard benchmark output for SMG2000, the mpiP profiling tool [6], and hardware counter data collected with the PMAPI tool [7]. Figure 7 and Figure 8 show the three types of performance data in the original form.

The first step was to add descriptive data for the two platforms into PerfTrack. Neither platform had previously been input into the database. UV is an early delivery component as part of the upcoming ASC Purple platform at LLNL [8]. It has 128 8-way nodes with Power4+ processors running at 1.5 GHz. BG/L is an IBM machine that has achieved more than 130 TeraFlop/s. When this study was done BG/L was in an early installation phase, that included only one partition with 16k nodes based on the PowerPC 440. We used the PerfTrack script interface to input the build and runtime data generated by PerfTrack modules. Finally, we input the performance results themselves. Figure 9 shows a portion of the PTdf file that was created for the translated data. Implementing new code to parse the SMG2000 benchmark output files took approximately one hour, using the supplied benchmark parsing code as a model. Parsing the mpiP and PMAPI output took a bit longer: Some of the measurements reported by mpiP broke down the time spent in each function according to the calling function. We decided to modify PerfTrack to accommodate multiple Resource Sets for each performance result, rather than one as previously implemented. This allows us to record the caller and callee for each value, so we have no loss of granularity of the data. The mpiP data is different from the raw benchmark data because it contains much more detail: the raw SMG2000 benchmark data only contains eight data values on the level of the whole execution, whereas the mpiP data contains multiple measurements broken down by process or whole execution, MPI function, and callsite of the MPI function.


out-90-110-30-1-16-4-0 (modified) - /u/kathryn/perftra...
File Edit Search Preferences Shell Macro Windows Help

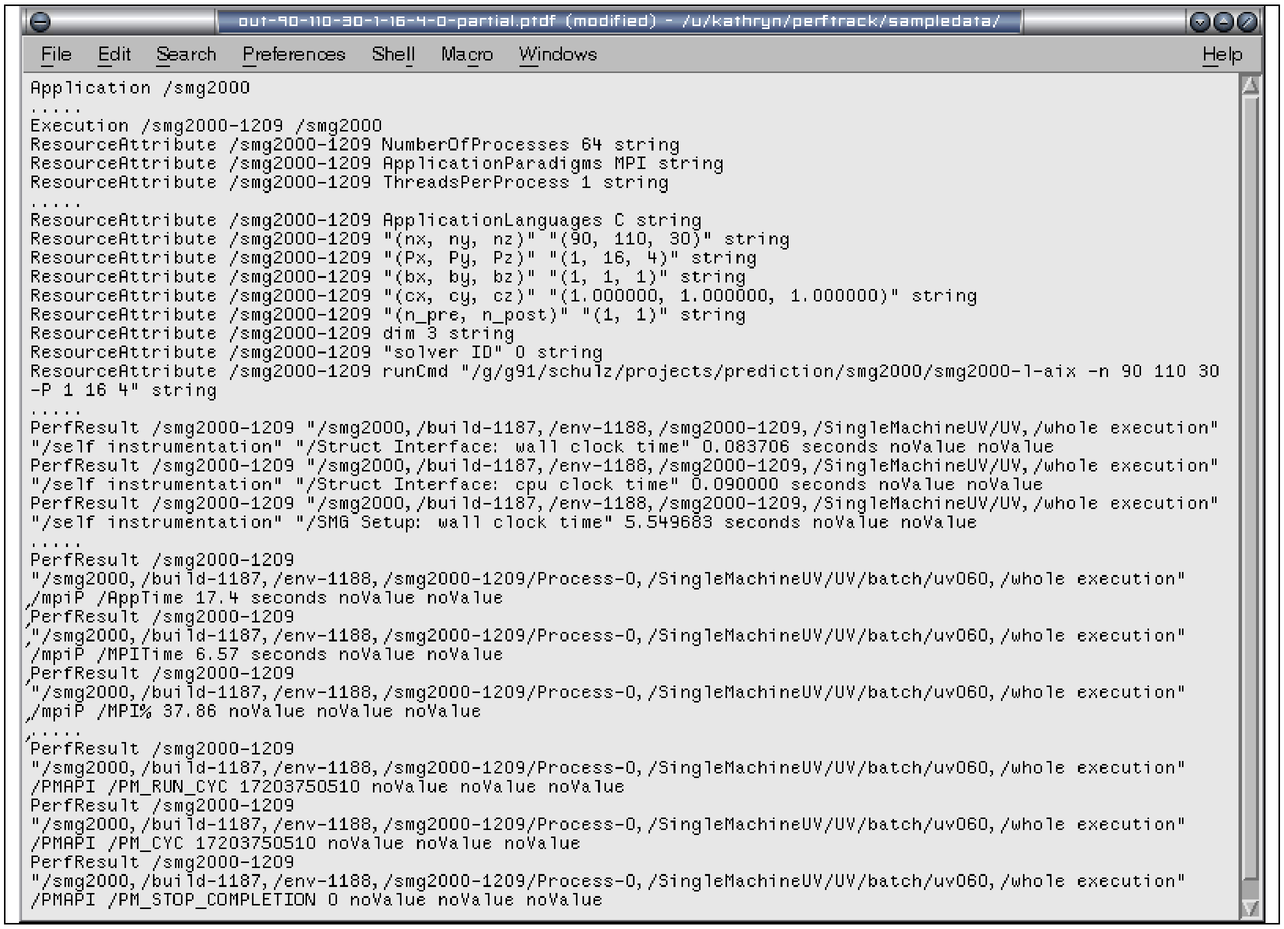
```
0: mpiP:
0: mpiP:  mpiP V2.7 (Build Aug 10 2004/10:27:28)
0: mpiP:  Direct questions and errors to mpip-help@llnl.gov>
0: mpiP:
0: Running with these driver parameters:
0:   (nx, ny, nz)    = (90, 110, 30)
0:   (Px, Py, Pz)    = (1, 16, 4)
0:   (bx, by, bz)    = (1, 1, 1)
0:   (cx, cy, cz)    = (1.000000, 1.000000, 1.000000)
0:   (n_pre, n_post) = (1, 1)
0:   dim             = 3
0:   solver ID       = 0
0: ============================================
0: Struct Interface:
0: ============================================
0: Struct Interface:
0:   wall clock time = 0.083706 seconds
0:   cpu clock time  = 0.090000 seconds
0: ============================================
0: Setup phase times:
0: ============================================
0: SMG Setup:
0:   wall clock time = 5.549683 seconds
0:   cpu clock time  = 5.540000 seconds
0: ============================================
0: Solve phase times:
0: ============================================
0: SMG Solve:
0:   wall clock time = 11.560329 seconds
0:   cpu clock time  = 11.590000 seconds
0:    PM_GROUP   0: ID    0, pm_slice0 (Time Slice 0):
0:            PM_COUNTER PM_RUN_CYC 17203750510
0:            PM_COUNTER PM_CYC 17203750510
0:            PM_COUNTER PM_STOP_COMPLETION 0
0:            PM_COUNTER PM_INST_CMPL 9195638130
```

Figure 7: Screenshot of SMG output. *This screenshot shows the output generated during one SMG application run. In addition to the native benchmark data, the run generated data from additional instrumentation inserted using the PMAPI hardware counter interface. This view shows only the first few lines of hardware counter data.*


smg2000-l-aix.64.811126.mpiP.partial (modified) - /u/kathryn/perftrack/sa...
File Edit Search Preferences Shell Macro Windows Help

```
@ mpiP
@ Command : /g/g91/schulz/projects/prediction/smg2000/smg2000-l-aix -n 90
110 30 -P 1 16 4
@ Version                  : 2.7
@ MPIP Build date          : Aug 10 2004, 10:27:28
@ Start time               : 2004 12 27 11:20:37
@ Stop time                : 2004 12 27 11:20:54
@ MPIP env var             : [null]
@ Collector Rank           : 0
@ Collector PID            : 811126
@ Final Output Dir         : .
@ MPI Task Assignment      : 0 uv060
@ MPI Task Assignment      : 1 uv060
@ MPI Task Assignment      : 2 uv060
@ MPI Task Assignment      : 3 uv060
.....
---------------------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0       17.4       6.57    37.86
   1       17.4        6.7    38.60
   2       17.4       6.52    37.55
   3       17.3       6.76    38.97
.....
```

Figure 8: MPIP data from the SMG application. *This screenshot shows part of the output from MPIP measurements of one SMG application run used in a PerfTrack study. The dotted lines indicate parts of the output that have been omitted from this view.*


out-90-110-30-1-16-4-0-partial.ptdf (modified) - /u/kathryn/perftrack/sampledata/
File Edit Search Preferences Shell Macro Windows Help

```
Application /smg2000
.....
Execution /smg2000-1209 /smg2000
ResourceAttribute /smg2000-1209 NumberOfProcesses 64 string
ResourceAttribute /smg2000-1209 ApplicationParadigms MPI string
ResourceAttribute /smg2000-1209 ThreadsPerProcess 1 string
.....
ResourceAttribute /smg2000-1209 ApplicationLanguages C string
ResourceAttribute /smg2000-1209 "(nx, ny, nz)" "(90, 110, 30)" string
ResourceAttribute /smg2000-1209 "(Px, Py, Pz)" "(1, 16, 4)" string
ResourceAttribute /smg2000-1209 "(bx, by, bz)" "(1, 1, 1)" string
ResourceAttribute /smg2000-1209 "(cx, cy, cz)" "(1.000000, 1.000000, 1.000000)" string
ResourceAttribute /smg2000-1209 "(n_pre, n_post)" "(1, 1)" string
ResourceAttribute /smg2000-1209 dim 3 string
ResourceAttribute /smg2000-1209 "solver ID" 0 string
ResourceAttribute /smg2000-1209 runCmd "/g/g91/schulz/projects/prediction/smg2000/smg2000-1-aix -n 90 110 30
-P 1 16 4" string
.....
PerfResult /smg2000-1209 "/smg2000,/build-1187,/env-1188,/smg2000-1209,/SingleMachineUV/UV,/whole execution"
"/self instrumentation" "/Struct Interface: wall clock time" 0.083706 seconds noValue noValue
PerfResult /smg2000-1209 "/smg2000,/build-1187,/env-1188,/smg2000-1209,/SingleMachineUV/UV,/whole execution"
"/self instrumentation" "/Struct Interface: cpu clock time" 0.090000 seconds noValue noValue
PerfResult /smg2000-1209 "/smg2000,/build-1187,/env-1188,/smg2000-1209,/SingleMachineUV/UV,/whole execution"
"/self instrumentation" "/SMG Setup: wall clock time" 5.549683 seconds noValue noValue
.....
PerfResult /smg2000-1209
"/smg2000,/build-1187,/env-1188,/smg2000-1209/Process-0,/SingleMachineUV/UV/batch/uv060,/whole execution"
/mpiP /AppTime 17.4 seconds noValue noValue
PerfResult /smg2000-1209
"/smg2000,/build-1187,/env-1188,/smg2000-1209/Process-0,/SingleMachineUV/UV/batch/uv060,/whole execution"
/mpiP /MPITime 6.57 seconds noValue noValue
PerfResult /smg2000-1209
"/smg2000,/build-1187,/env-1188,/smg2000-1209/Process-0,/SingleMachineUV/UV/batch/uv060,/whole execution"
/mpiP /MPI% 37.86 noValue noValue noValue
.....
PerfResult /smg2000-1209
"/smg2000,/build-1187,/env-1188,/smg2000-1209/Process-0,/SingleMachineUV/UV/batch/uv060,/whole execution"
/PMAPI /PM_RUN_CYC 17203750510 noValue noValue noValue
PerfResult /smg2000-1209
"/smg2000,/build-1187,/env-1188,/smg2000-1209/Process-0,/SingleMachineUV/UV/batch/uv060,/whole execution"
/PMAPI /PM_CYC 17203750510 noValue noValue noValue
PerfResult /smg2000-1209
"/smg2000,/build-1187,/env-1188,/smg2000-1209/Process-0,/SingleMachineUV/UV/batch/uv060,/whole execution"
/PMAPI /PM_STOP_COMPLETION 0 noValue noValue noValue
```

Figure 9: PTDF generated for the SMG application. *This view shows parts of a PTdf file generated during a PerfTrack study. It shows the resources and performance results created to store the benchmark, PMAPI, and MPIP data,*

We were able to store all of the new types of data into PerfTrack, with one modification to record two function names per metric. Statistics on the study data are included in Table 1. Developing methods to parse the different performance data files took a reasonable amount of effort, and all of the data could be expressed in PTdf. At the time of this report, we had successfully loaded 76 of the executions into PerfTrack. Preliminary observations of data load time indicate this type of data as an area of focus for performance optimization.

4.3 Incorporating Paradyn Performance Data

The goal of this study was to test PerfTrack's ability to incorporate data from different types of performance tools. Paradyn [9] is designed to measure the performance of long-running programs. It uses dynamic instrumentation, the insertion of performance measurement instructions into an executing program, to collect application level performance data. The use of dynamic instrumentation means that the performance data collected may not cover the entire execution time of the program; collection can be started or ended at any time during the program run. This is different than the other performance tools we have supported so far which report data for the entire execution of the program. Another difference is that Paradyn has its own resource hierarchy that includes resource types not found in the PerfTrack base resource types (see Figure 10).

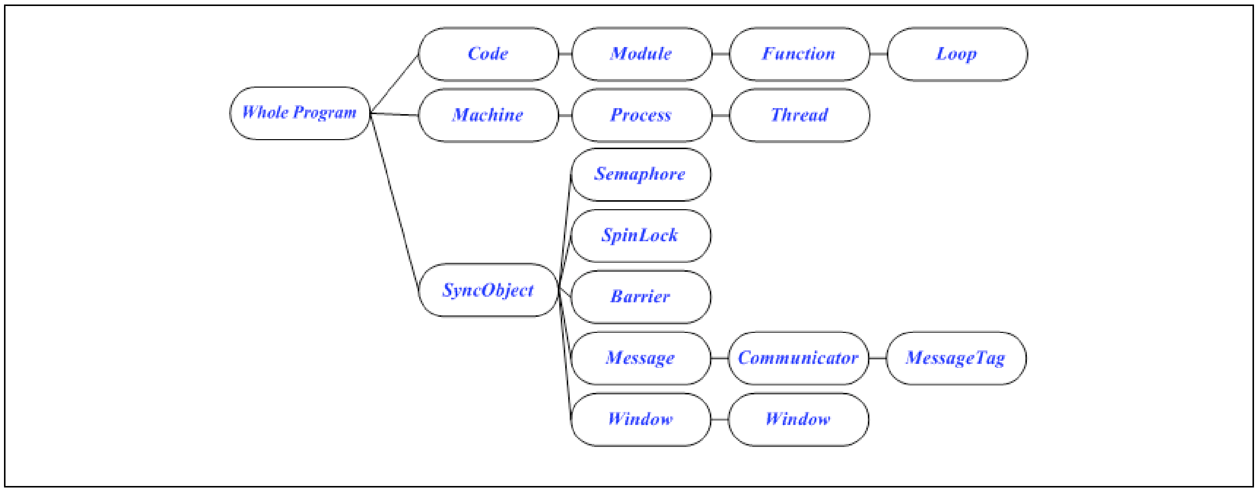


Figure 10: Resource type hierarchy for Paradyn. *Paradyn's resource hierarchy has three main types: Code, which contains the modules, functions, and loops in the program; Machine, which contains the nodes on which the program is running and the processes and threads that run on those nodes; and SyncObject, which contains the synchronization objects used, such as communicators for message passing and Window objects for MPI RMA data transfers.*

Paradyn inserts instrumentation into a running program either in response to a request from its automatic performance bottleneck search module (Performance Consultant) or in response to a request by an interactive user. Either of these methods results in performance data being collected for a metric-focus pair, where a metric is any measurable characteristic and a focus specifies a part of an application for which performance data can be collected, such as a process, a function, or a synchronization object. Performance data for a metric-focus pair is collected in an array of predefined size. The array elements are called bins and contain the values measured for the metric-focus pair for a specific time interval. A user can save all of the data collected in a Paradyn session using the "Export" button in the Paradyn user interface. There are several types of text files that can be exported by Paradyn: histograms, an index file that describes the histogram files, a search history graph, and a list of resources. Each histogram file contains the data for one of the performance data arrays, with a header containing information about the data, including the metric-focus pair, the number of data bins in the file, and the amount of time each bin represents, followed by the data from each bin in the data array. The search history graph file contains the results of the Performance Consultant's search. The resources file contains a list of all the Paradyn resources for the application, including processes, threads, functions, and synchronization objects.

Adding Paradyn data to an existing PerfTrack data store required three steps: Determine a mapping between the Paradyn resource hierarchy and the PerfTrack type hierarchy; write a PerfTrack parser to convert the data in the Paradyn performance data files into PTdf format; and load the PTdf files into PerfTrack using the script interface.

Our resource type mapping is shown in Figure 11. We map the processes and threads of Paradyn's Machine hierarchy to PerfTrack's execution hierarchy. The machine nodes in Paradyn's Machine hierarchy are stored as resource attributes of the process resources. Paradyn's Code hierarchy maps to both the PerfTrack build and environment hierarchies. This is because PerfTrack distinguishes between static modules and dynamically linked modules. It is not always possible to determine if a Paradyn Code resource represents a dynamic or static module, in which case we default to the build (static) hierarchy. An example of a time when it may be impossible to distinguish which category a module belongs to are when we encounter resources in Paradyn's "DEFAULT_MODULE." When Paradyn cannot find the necessary information to determine which module a function belongs to or when the functions do not rightfully belong to any source file or module (as in the case of built-in functions), the function is put into the "DEFAULT_MODULE." For Paradyn's syncObject hierarcy, we created a new top-level PerfTrack resource hierarchy that exactly corresponds to Paradyn's syncObject hierarchy. After converting the Paradyn resource hierarcy, we parse each of the histogram files and create a new performance result for each of the bins in each file. The context used for each performance result is composed of the PerfTrack-mapped resources in the focus reported by Paradyn. In addition, to model the time intervals for Paradyn's performance measurements, we added resources in PerfTrack's time hierarchy. The notion of the entire execution of a program in Paradyn is called the global phase. Paradyn also offers the user the ability to create new local phases during the execution, which are non-overlapping subintervals of the global phase. We map the global phase to the top-level of our time hierarchy. Children of the global phase can be either bins or local phases. The local phases also have bins as children. The bin resources are created when parsing the histogram data files, and have attributes that tell their start time and end time relative to the beginning of the program execution.

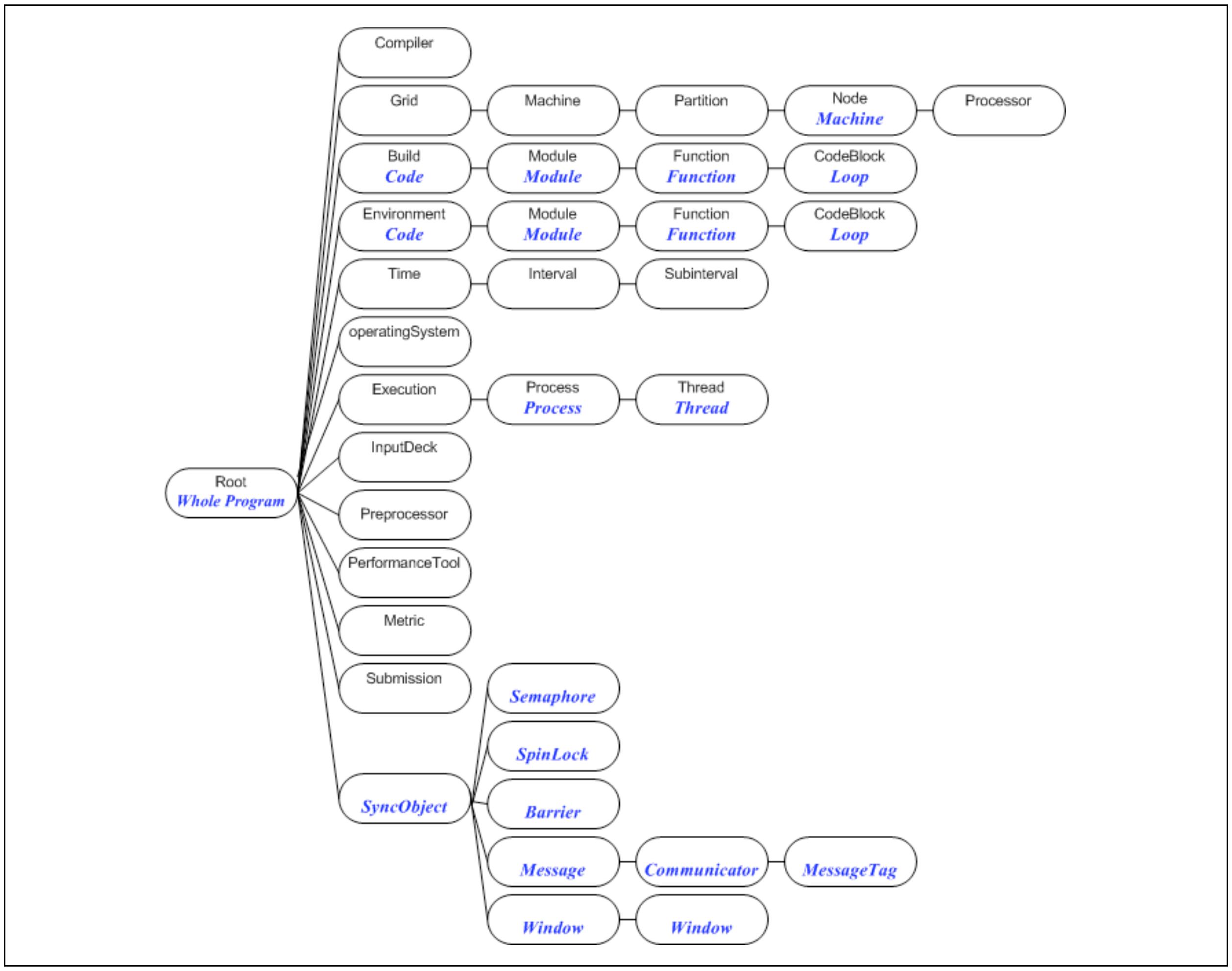


Figure 11: Integration of Paradyn data into the PerfTrack type hierarchy. *The resource type tree shows the result of additional resource types, and mapping of types, to add Paradyn data to an existing PerfTrack data store.*

We used our parser to add three executions of IRS on MCR to the PerfTrack database. The PTdf files included data collected by our automated build and run data gathering scripts, plus histograms and resources exported by Paradyn. We created a separate PTdf file for each execution. Each of these had approximately 17,000 resources, 8 metrics, and 25,000 performance results. The number of resources and performance results differed for each of the executions due to the way Paradyn collects performance data. Because Paradyn utilizes dynamic instrumentation, performance measurement instructions are not necessarily inserted into the program at the beginning of its execution. This means that Paradyn may not have data for some bins in a histogram. In this case, 'nan' (not a number) is printed in the histogram data file. We do not record 'nan' entries as performance results. Because instrumentation for a particular metric-focus pair may not be inserted into the program at exactly the same time across executions, the number of performance results and resources varied between the three executions.

5. Related Work

A number of research efforts include some form of database support for the storage of performance results, going back of course to Snodgrass's seminal work on temporal databases [10]. These projects are usefully grouped according to two factors: the intended use of the data store, and the design and scope of the data store. Two ongoing research efforts integrate database technology with performance data for the purpose of parallel performance prediction and modeling: Prophesy and POEMS. Prophesy [11, 12] uses a relational database to record performance data, system information, and details about an application. However, the schema used is fixed, rather than extensible. The goal of the POEMS project [13] is end-to-end performance modeling of parallel and distributed systems. The models take into account application software, operating system, and hardware. Other research efforts have as a goal improved performance diagnosis. Karavanic and Miller [14] have demonstrated the use of historical performance data in the diagnosis of parallel applications. The PYTHIA-II [15] system provides advice for selecting the best software/machine combination from defined alternatives, using an object relational database to store performance information, and data mining and machine learning techniques. It does not store data about the build environment. Two related projects [16, 17] have integrated database management for performance analysis. In both cases, the data is stored in a fixed form, but data conversion utilities are provided for inputting different kinds of data. The SCALEA project [17] includes a database management system based data repository with a fixed schema, that includes common data fields related to parallel performance experimentation. Like PerfTrack, the data store includes a description of the runtime platform, although it does not include information about the build environment, and the runtime platform information includes a smaller number of attributes. The PerfDMF project [16] has developed a performance data management framework for storing and querying parallel performance data, with greater emphasis on accomodating different forms of data. This work is closely related to PerfTrack, with differences in motivation, approach, and goals. The PerfDMF database began as a means of storing performance data measured with the TAU performance tool suite, and has expanded to be a more flexible means of storing different kinds of performance data. PerfDMF incorporates a relational database, an interface implemented with Java and JBDC, and an analysis toolkit component. The design is less flexible and extensible than PerfTrack's generic resource-attribute design. PerfTrack adds a simple model for describing and selecting performance results, and an extensible resource type system.

Comparison-based performance diagnosis is a relatively recent research area, with early work by Miller and Karavanic. [18-20] Song et al extended this work by the addition of additional comparison operators and use of a more restricted, fixed data design to form their Cube algebra. [21] Our work builds on previous research on parallel performance diagnosis, conducted by the PPerfDB research group at Portland State University. PPerfDB [22] is a tool for automated comparison of application runs and application performance data that incorporates comparative analysis techniques, enables direct comparison of performance data collected using a variety of types of performance tools, and provides the user with a simple interface to navigate what is often a large amount of data. PPerfDB uses the underlying file system for data storage.

Early work in experiment management support for parallel performance analysis was conducted by Miller and Karavanic. [18-20]. Zenturio [23] is an experiment management system designed to automatically conduct large numbers of experiments for parameter and performance studies. It includes a visualization feature to compare multiple runs of the same code, and a database repository for performance data. Unlike PerfTrack, the focus of Zenturio is on launching sets of experiments in an automated fashion. There is no ability to add data generated by other means for direct comparison to Zenturio runs. PerfTrack is designed to incorporate data generated by any performance tool, and includes the ability to compare across applications or across application versions.

6. Future Work

A number of PerfTrack enhancements and extensions are in progress, including the addition of a set of comparison operators to automate the comparison of different executions and performance results in the data store, and support for additional database packages. In the future we plan to extend this work in several key ways. The experiment management infrastructure will be used to investigate methods for comparing performance data across different executions and diagnose performance bottlenecks using data from more than one execution. We plan to develop new data analysis techniques that leverage the available data store to conduct a more efficient and accurate performance analysis. We also plan to develop additional techniques for a richer visualization interface. We plan to explore complex performance results in PerfTrack to accommodate multi-faceted performance data such as that from Paradyn's Performance Consultant and to avoid creating a new performance result for each bin in a Paradyn histogram file. Finally, we plan to explore the incorporation of performance predictions and models into PerfTrack for direct comparison to actual program runs.

7. Conclusions

In this paper we have presented PerfTrack, a data store and interface for the storage and retrieval of performance data describing the runtime behavior of large-scale parallel applications. PerfTrack allows performance data collected in different locations and formats to be integrated and used in a single performance analysis session. PerfTrack uses a database management system for the underlying data store, and includes interfaces to the data store plus a set of modules for automatically collecting descriptive data for each experiment, including the build and runtime platforms. We described a working prototype of PerfTrack that uses the Oracle or PostgreSQL DBMS. We demonstrated the prototype's functionality with three case studies: one, a comparative study of ASC purple benchmarks on high end Linux and AIX platforms; the second, a parameter study conducted at Lawrence Livermore National Laboratory (LLNL) on two high end platforms, BlueGene/L and UV; the third, an investigation of incorporating data from the Paradyn Performance Tool. We conclude that it is both possible and beneficial to integrate database management technology into a tool for comparison-based performance diagnosis.

Acknowledgments

This work was supported in part by UC/LLNL subcontract B539302. We thank Martin Schulz, David Maier, Terence Critchlow, Matthew Legendre, and Phil Roth for their contributions to this work.

References

[1] Trolltech, "QT3.3 Whitepaper," April 20, 2005. (available at www.trolltech.com)

[2] G. v. Rossum, "Python Reference Manual Release 2.3.4," PythonLabs May 20, 2004. (available at www.python.org)

[3] A. Tuininga, "cx_Oracle," Release 4.1 beta1 Computronix, 2004. (available at http://starship.python.net/crew/atuining/cx_Oracle/html/about.html)

[4] D'Arcy J. M. Cain, "PyGreSQL Version 3.6.2." (http://www.pygresql.org/)

[5] E. Ipek, Bronis de Supinski, Martin Schulz, and Sally A. McKee, "An Approach to Performance Prediction for Parallel Applications," submitted for publication, 2005.

[6] "mpiP: Lightweight, Scalable MPI Profiling." (available at www.llnl.gov/CASC/mpip)

[7] "Performance Monitor API Programming," in *Performance Tools Guide and Reference*. (available at http://publib16.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixbman/prftools/mastertoc.htm#mtoc)

[8] "About ASC Purple." (available at www.llnl.gov/asci/platforms/purple/index.html)

[9] B. P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall, "The Paradyn Parallel Performance Measurement Tool," *IEEE Computer*, vol. 28, pp. 37-46, 1995.

[10] R. Snodgrass, "A Relational Approach to Monitoring Complex Systems," *ACM Transactions on Computer Systems*, vol. 6, pp. 157-196, 1988.

[11] V. Taylor, X. Wu, and R. Stevens, "Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, 2003.

[12] X. Wu, V. Taylor, J. Geisler, X. Li, Z. Lan, R. Stevens, M. Hereld and I.R. Judson, "Design and Development of Prophesy Performance Database for Distributed Scientific Applications," presented at 10th SIAM Conference on Parallel Processing for Scientific Computing, Virginia, USA, 2001.

[13] S. Adve, R. Bagrodia, J.C. Browne, E. Deelman, A. Dube, E. Houstis, J. Rice, R. Sakellariou, D. Sundaram-Stuken, P.J. Teller, and M.K. Vernon, "POEMS: End-to-end Performance Design of Large Parallel Adaptive Computational Systems," *IEEE Transactions on Software Engineering*, vol. 26, pp. 1027-1048, 2000.

[14] K. L. Karavanic, and B.P. Miller, "Improving Online Performance Diagnosis by the Use of Historical Performance Data," presented at SC99, Portland, Oregon, USA, 1999.

[15] E. N. Houstis, A.C. Catlin, J.R. Rice, V.S. Verykios, N. Ramakrishnan, and C. Houstis, "PYTHIA-II: A Knowledge/Database System for Managing Performance Data and Recommending Scientific Software," *ACM Transactions on Mathematical Software*, vol. 26, pp. 227-253, 2000.

[16] K. Huck, A.D. Malony, R. Bell, A. Morris, "Design and Implementation of a Parallel Performance Data Management Framework," presented at The 2005 International Conference on Parallel Processing (ICPP), to appear, Oslo, Norway, 2005.

[17] H. Truong, and T. Fahringer, "On Utilizing Experiment Data Repository for Performance Analysis of Parallel Applications," presented at 9th International Euro-Par Conference (Euro-Par 2003), Klagenfurt, Austria, 2003.

[18] K. L. Karavanic, and B.P. Miller, "Experiment Management Support for Performance Tuning," presented at SC97, San Jose, California, USA, 1997.

[19] K. L. Karavanic, "Experiment Management Support for Parallel Performance Tuning," in *Computer Science*. Madison, Wisconsin: University of Wisconsin - Madison, 1999.

[20] K. L. Karavanic, and B.P. Miller, "A Framework for Multi-Execution Performance Tuning," in *On-Line Monitoring Systems and Computer Tool Interoperability*, T. L. a. B. P. Miller, Ed. New York, USA: Nova Science Publishers, 2003.

[21] F. Song, F. Wolf, N. Bhatia, J. Dongarra, S. Moore, "An Algebra for Cross-Experiment Performance Analysis," presented at 2004 International Conference on Parallel Processing (ICPP-04), Montreal, Quebec, CANADA, 2004.

[22] M. Colgrove, C. Hansen, K.L. Karavanic, "Managing Parallel Performance Experiments with PPerfDB," presented at The IASTED International Conference on Parallel and Distributed Computing and Networking (PDCN 2005), Innsbruck, Austria, 2005.

[23] R. Prodan, T. Fahringer, "ZENTURIO: An Experiment Management System for Cluster and Grid Computing," presented at 4th International Conference on Cluster Computing (CLUSTER 2002), Chicago, USA, 2002.