# Performance Co-Pilot User's and Administrator's Guide

Document Number: 007-2614-001

CONTRIBUTORS

Engineering and written contributions by Mark Goodwin, Seppo Keronen, Jonathan Knispel, Ken McDonell, and Jeff Zurschmeide.
Edited by Christina Cary
Production by Lorrie Williams
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz



Performance Co-Pilot User’s and Administrator’s Guide
Document Number: 007-2614-001

# Contents

# List of Figures

# List of Tables

# About This Guide

This guide describes the Performance Co-Pilot (PCP) software package of advanced performance management applications for the Silicon Graphics family of graphical workstations and servers. The Performance Co-Pilot provides a systems-level suite of tools that cooperate to deliver distributed, integrated performance monitoring and performance management services spanning the hardware platform, the operating system, the DBMS, and the users' applications.

The following chapters are provided in this Guide:

- Chapter 1, "Introduction to the Performance Co-Pilot," provides an introduction to the concepts and structure of the Performance Co-Pilot. Instructions are provided for installation and configuration.
- Chapter 2, "PCP Utilities and Tools," details the various software tools and commands that make up the Performance Co-Pilot product.
- Chapter 3, "The Performance Metrics Application Programming Interface (PMAPI)," describes the API that allows you to customize and extend the Performance Co-Pilot with performance monitoring tools of your own design.
- Chapter 4, "Extending and Refining the PCP Toolkit," describes strategies for the design and development of your extensions and customizations of the Performance Co-Pilot.
- Chapter 5, "Troubleshooting the Performance Co-Pilot," details pointers to help troubleshoot common problems.
- Chapter 6, "Glossary of Acronyms," provides a comprehensive glossary of terms and acronyms used in this guide, in the reference pages, and in the Release Notes for the Performance Co-Pilot.

# Information About This Guide

## Audience

This guide is written for the system administrator who are directly using and administering the PCP applications. It is assumed that you have installed InSight or have access to the *IRIX Advanced Site and Server Administration Guide* and the *Personal System Administration Guide* and are familiar with their contents.

## Additional Resources

The primary resources for system administrators are the *IRIX Advanced Site and Server Administration Guide* and the *Personal System Administration Guide*. These guides explain the basic tasks and responsibilities of system administrators. Also, the *NFS Administration Guide* and the *NIS Administration Guide* are useful references if you have the optional NFS software installed on your system.

The IRIX Reference Pages, available online through the *man*(1) command, are always an important resource for system administrators.

## Typographical Conventions

As you read this guide, you will notice that special fonts are used for certain words.

`typewriter font`
: Indicates system output, such as responses to commands that you enter and the text of messages that appear in Warning and other informational windows. This font is also used for examples of the contents of files, filters and filter components, examples of network addresses, Management Information Base (MIB) object names, and example workstation and network names and addresses.

**`typewriter bold`**
: Indicates text you must enter, such as command lines and filter expressions. Names of nonprinting keys on the keyboard, such as the **`<Enter>`** key, also appear in typewriter bold and are surrounded by angle brackets.

**bold**
: Designates literal options to commands.

*italic*s
: Indicates filenames, command names, and manual page names. Lowercase italic words also represent variables—text strings that you must specify. References to other documents, button names, *inst*(1M) subsystem names, user IDs, and group names are also in *italics*.

*Chapter 1*

# Introduction to the Performance Co-Pilot

This chapter provides a brief overview of the individual software components of the Performance Co-Pilot (PCP), and other information to help you use this guide. The following sections are provided in this chapter:

- "Introduction to the Performance Co-Pilot (PCP)" on page 1 provides a general overview of the Performance Co-Pilot components, and their capabilities and intended usage.
- "PCP Conceptual Foundations" on page 8 provides a look at the tools and commands that make up Performance Co-Pilot.
- "Installing and Configuring the PCP" on page 28 discusses the PCP software packages that must be installed to run the Performance Co-Pilot on your network.
- "User Interface Terminology" on page 32 provides information about the style conventions used in creating the PCP tools.

## Introduction to the Performance Co-Pilot (PCP)

The following sections provide a short introduction to the concepts and components of PCP.

### PCP Objectives

The PCP provides a range of services that are designed to be used to monitor and manage system performance. These services are distributed and scalable to accommodate the most complex system configurations and performance problems.

The following objectives are met by the Performance Co-Pilot software product:

### Target Usage of the PCP

The PCP is targeted at the performance analyst, benchmarker, engineering developer, database administrator, capacity planner, or system administrator with an interest in overall system performance and a need to quickly isolate and understand performance behavior, resource utilization, activity levels, and bottlenecks in large complex systems. Platforms that benefit from this level of performance analysis include large servers or clusters of servers, DBMS providers, and video, computing or file servers.

### Empowering the User

To deal efficiently with the dynamic behavior of complex systems, you need services that filter the “noise” from the overwhelming stream of performance data, allowing the performance manager to concentrate on the exceptional scenarios. The ability to go back and review previous performance data, performance visualization, and the automated reasoning about performance data provide the necessary high bandwidth filtering.

From the PCP end-user’s perspective, the PCP presents an integrated suite of tools, user interfaces, and services that support real-time and retrospective performance analysis, with a bias towards eliminating mundane information and focusing attention on the exceptional and extraordinary performance behavior. When this is done, the user can concentrate upon in-depth analysis or target management procedures, for the critical system performance problems.

### Unification of Domains of Performance Metrics

At the lowest level, performance metrics are collected and managed in autonomous performance domains such as the IRIX operating system, a Database Management System, or an end-user application. These domains support a multitude of access-control policies, access methods, data semantics, and multi-version support. All of this detail is irrelevant to the developer and user of a performance monitoring tool, and is hidden by the PCP infrastructure.

**Uniform Naming and Access to Performance Metrics**

Usability and extensibility of performance management tools mandate a single scheme for naming performance metrics. The set of defined names constitute a Performance Metrics Names Space (PMNS). Within the PCP, the PMNS is adaptive so that it can be extended, re-shaped, and pruned to meet the needs of particular applications and users.

A single interface is provided to retrieve the values for all performance metrics.

**Distributed Operation**

From a purely pragmatic viewpoint, a single workstation must be able to concurrently monitor the performance of multiple remote hosts. At the same time, a single host may be subject to monitoring from multiple remote workstations.

These requirements suggest a classical “client-server” architecture, which is exactly what PCP uses to provide seamless and concurrent access to performance metrics, independent of their host location.

**Dynamic Adaptation to Change**

Complex systems are subject to continual changes as network connections fail and are re-established; nodes are taken out of service and rebooted; hardware is added and removed; and software is upgraded, installed, or removed. Often these changes are asynchronous and remote (perhaps in another geographic region, or domain of administrative control).

The distributed nature of the PCP (and the modular fashion in which performance metrics domains may be installed, upgraded and configured on a host-by-host basis) enables the PCP to readily adapt to changes in the monitored system(s). Variations in the available performance metrics as a consequence of configuration changes are handled automatically and become visible to all clients as soon as the re-configured host is rebooted or the responsible agent is restarted.

The PCP also detects loss of client-server connections, and supports subsequent automated client re-connection.

**Logging and Retrospective Analysis**

A range of tools are provided to support adaptive and flexible logging of performance metrics for archival, playback, remote diagnosis, and capacity planning. Archive logs may be accumulated either at the host being monitored, at a monitoring workstation, or both.

A universal replay mechanism, modeled on a VCR paradigm, supports “stop, rewind and replay at variable speed” processing of performance information for both archived and real-time data.

Unification of real-time access and access to the archive logs, in conjunction with the VCR services, provides new and powerful ways to build performance tools and to review both current and historical performance data.

**PCP Extensibility**

The PCP encourages the integration of new performance metrics into the Performance Metrics Name Space (PMNS), the collection infrastructure and the logging framework. The guiding principle is “if it is important for monitoring system performance, and you can measure it, you can easily integrate it into the PCP framework”.

For many PCP end-users, the most important performance metrics are not those already supported, but new performance metrics that characterize the essence of “good” or “bad” performance at their site, or within their application environment. An example is an application that measures the round-trip time for a benign “probe” transaction against an ORACLE™ database. A source code implementation of this application is provided in the distribution, and by using the PCP toolkit and the services of the PMAPI, the times measured by this application can easily be integrated into the PCP framework at a site running ORACLE.

## Overview of PCP Components

The PCP is made up of several graphical tools and some related commands. Each tool or command is documented completely in a reference page. These reference pages are named for the tools and commands they describe. The reference pages are accessible through the *man*(1) command. For example, the reference page for the tool *mpvis*(1) is viewed by giving the command:

```
man mpvis
```

A list of all tools and commands is provided below as a directory to the reference pages.

The major component tools of the Performance Co-Pilot are:

| | |
|---|---|
| *dkvis*(1) | The *dkvis* tool is a graphical disk device utilization viewer. It displays a three-dimensional bar chart showing activity in the disk subsystem. |
| *memvis*(1) | The *memvis* tool is a graphical physical memory usage viewer that displays a bar chart depicting physical memory use on a per-process, and a per-region-per-process basis. |
| *mpvis*(1) | The *mpvis* tool displays a three-dimensional bar chart of multiprocessor CPU utilization. |
| *nfsvis*(1) | The *nfsvis* tool displays a bar chart showing NFS™ (Network File System) client and server request activity, for systems on which the optional NFS software product has been installed. |
| *opsview*(1) | The power of performance visualization is demonstrated by *opsview*, an application that provides a high-level view of the performance of two nodes in an ORACLE Parallel Server (OPS) configuration, with “drill-down” navigational links to other visualization tools. Even when OPS is not installed, the capabilities of *opsview* may be demonstrated using platforms not running the ORACLE product. |
| *pmcd*(1) | The Performance Metrics Collection Daemon. This daemon must run on each system being monitored, to collect and export the performance information necessary to monitor the system. |

| | |
|---|---|
| *pmchart*(1) | The *pmchart* tool displays trends over time for arbitrarily selected performance metrics from one or more hosts, and one or more domains of performance metrics. |
| *pmdbg*(1) | The Performance Co-Pilot tools include internal diagnostic facilities that may be activated by run-time flags. *pmdbg* describes the available diagnostic facilities and the associated control flags. |
| *pmdumplog*(1) | The *pmdumplog* command may be used to dump selected state and control information from a PCP archive log, as created by *pmlogger*. |
| *pmerr*(1) | The *pmerr* command translates Performance Co-Pilot error codes into human-readable error messages. |
| *pmgenmap*(1) | The *pmgenmap* command is a program development tool that generates C declarations and *cpp* macros to aid the development of customized programs that use the facilities of the PCP. |
| *pmie*(1) | The *pmie* tool is an inference engine to evaluate predicate-action rules over the domain of performance metrics, for performance alarms, automated system management tasks, dynamic tuning configuration, and so on. |
| *pminfo*(1) | The *pminfo* tool displays various types of information about performance metrics available through the facilities of the Performance Co-Pilot. |
| *pmlc*(1) | The *pmlc* command is used to exercise control over an instance of the PCP archive logger *pmlogger*, to modify the profile of which metrics are logged and how frequently their values are logged. |
| *pmlogger*(1) | The *pmlogger* command is used to create PCP archive logs of performance metrics over time. Many tools accept these PCP archive logs as alternative sources of metrics for retrospective analysis. |
| *pmkstat*(1) | The *pmkstat* command provides text-based display of metrics that summarize system performance at a high level, suitable for ASCII logs or enquiry over a modem. |

*pmnsadd*(1) The *pmnsadd* command adds a subtree of new names into a Performance Metrics Name Space (PMNS), as used by the components of the Performance Co-Pilot.

*pmnsdel*(1) The *pmnsdel* command removes a subtree of names from a Performance Metrics Name Space (PMNS), as used by the components of the Performance Co-Pilot.

*pmstore*(1P) The *pmstore* command is used to re-initialize counters or to assign new values to metrics that act as control variables. The command changes the current values for the specified instances of a single performance metric.

*pmval*(1) The *pmval* command provides text-based display of the values for arbitrary instances of selected performance metrics, suitable for ASCII logs or enquiry over a modem.

*pmview*(1) The *pmview* tool is a generalized 3D Inventor application that supports dynamic displays of clusters of related performance metrics as utilization blocks (or towers) on a common base plane.

### Additional Performance Co-Pilot Features

Platform Support

Performance Co-Pilot supports domains of performance metrics that include all IRIX Version 5.3 (and later) kernel instrumentation, process-level resource utilization, environmental monitors for CHALLENGE™ systems, ORACLE Version 7 (and later) performance tuning views, and Cisco® router statistics. The distributed agents support a large number of distinct performance metrics; over 500 for IRIX, including 100 per process; 250 for ORACLE 7; plus assorted metrics for Cisco routers and CHALLENGE environmental monitors.

Secure Operation

A host-based security model provides optional control over the execution of PCP service requests from designated remote hosts and / or workstations.

| | |
|---|---|
| API | An exported Performance Metrics API (PMAPI) for building site-specific or application-specific performance-related tools; this PMAPI provides access to all of the services of the underlying PCP infrastructure, and source for sample programs is provided. |

## PCP Conceptual Foundations

The following sections provide a detailed overview of the concepts that underpin the services and facilities of the PCP.

### Sources of Performance Metrics and Their Domains

Instrumentation for the purpose of performance monitoring typically consists of counts of activity or events, attribution of resource consumption, and service-time or response-time measures. This instrumentation may exist in one or more of the following functional **domains**, each with an associated access method (see Figure 1-1):

- The IRIX kernel. For example, *sar* data structures, per-process resource consumption, disk activity, or the memory management instrumentation.
- A layered system product. For example, the temperature, voltage levels and fan speeds from the environmental monitor in a Challenge, or the length of a printer spool queue as reported by *lpstat*.
- A DBMS product. For example, the V$ views and `bstat`/`estat` summaries of ORACLE, or the *tbmonitor* statistics from Informix.
- External equipment such as network routers and bridges.
- An application program. For example, measured response time for a production application running a periodic and benign “probe” transaction (as often used in service quality agreements), or throughput in jobs per hour for a batch stream.

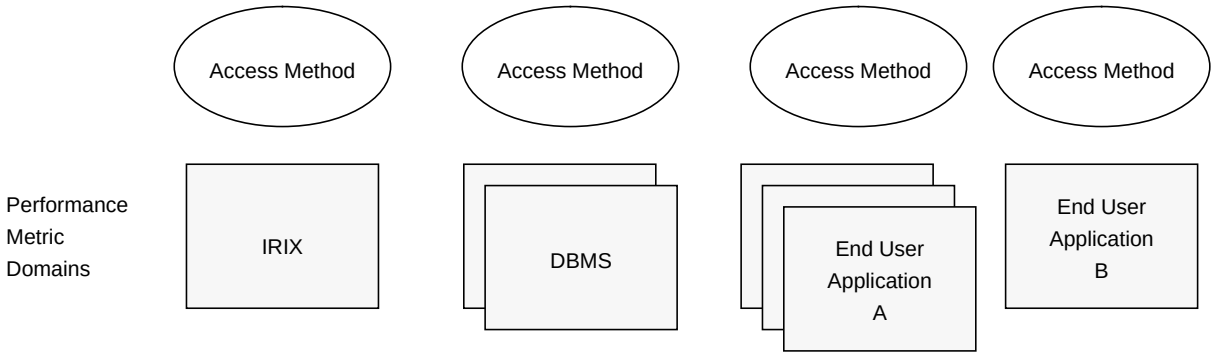


**Figure 1-1** Performance Metric Domains as Autonomous Collections of Data

For each domain, the set of performance metrics may be viewed as an abstract data type, with an associated set of **methods** that may be used to:

- interrogate the meta-data that describes the syntax and semantics of the performance metrics
- control (enable or disable) the collection of some or all of the metrics
- extract instantiations (current values) for some or all of the metrics

We refer to each such domain as a Performance Metrics Domain (PMD) and assume that PMDs are functionally, architecturally and administratively independent and autonomous. Obviously the set of PMDs available on any host is variable, and changes with time as software and hardware are installed and removed.

The number of PMDs may be further enlarged in cluster-based or network-based configurations, where there is potentially an instance of each Performance Metrics Domain on each node. Hence, the management of PMDs must be both extensible at a particular host, and distributed across a number of hosts.

Each PMD on a particular host must be assigned a unique PMD identifier. In practice, this means unique identifiers shall be assigned globally for each PMD type. For example, the same identifier would be used for the IRIX PMD on all hosts.

## Performance Metrics Name Space

Internally, each unique performance metric is identified by a Performance Metric Identifier (PMID) drawn from a universal set of identifiers, including some that are reserved for site-specific, application-specific and customer-specific use.

An external **name space** (the Performance Metrics Name Space, or PMNS) maps from a hierarchy (or tree) of external names to PMIDs. Each node in the name space tree is assigned a label that must begin with an alphabet character, and be followed by zero or more alphanumeric characters or the underscore ("_") character. The root node of the tree has the special label of *root*. A metric name is formed by traversing the tree from the root to a leaf node with each node label on the path separated by a period. The common prefix *root.* is omitted from all names. For example, in the small subsection of a PMNS shown in Figure 1-2, the following are valid names for performance metrics;

```
irix.kernel.percpu.syscall
irix.network.tcp.rcvpack
oracle.demo.all.dbwr.lruscans
```

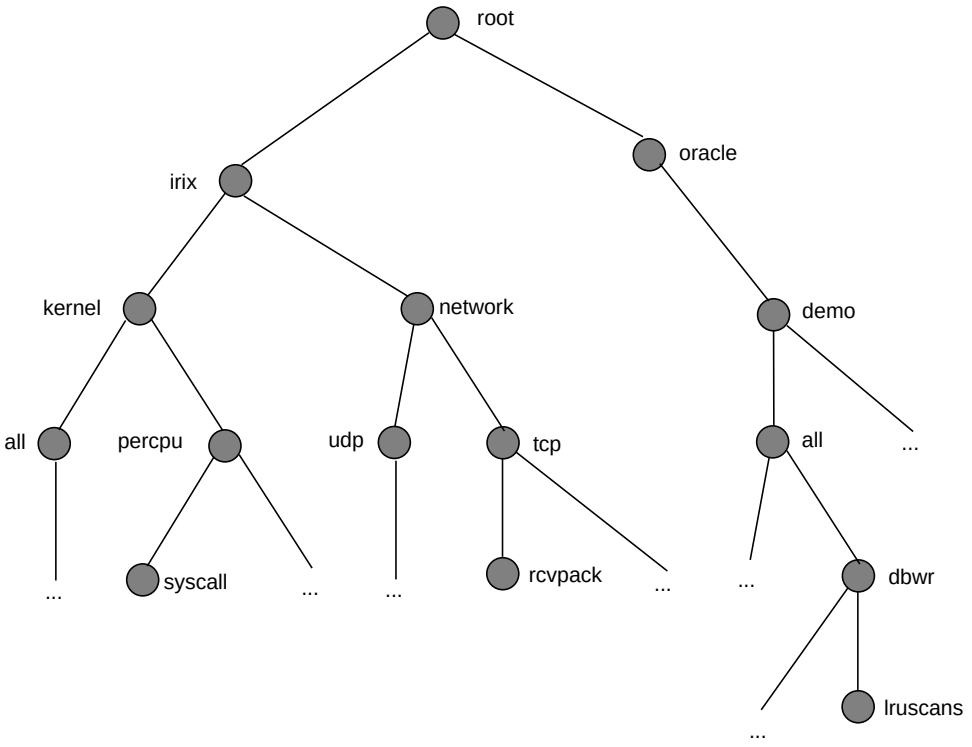


**Figure 1-2** A Small Performance Metrics Name Space (PMNS)

Although a default PMNS is shipped and updated by the components of the Performance Co-Pilot, individual users may create their own name space for metrics of interest, and all tools may use a private PMNS, rather than the default PMNS.

## Descriptions for Performance Metrics

Through the various Performance Metric Domains, the PCP must support a wide range of formats and semantics for performance metrics. This "metadata" describing the performance metrics includes

- the internal identifier for the metric
- the format and encoding for the values of the metric
- the semantics of the metric, particularly the interpretation of the values as free-running counters or instantaneous values
- the dimensionality and scale of the values, in the dimensions of Events, Space and Time
- an indication if the metric may have one or many associated values
- short and extended help text, describing the metric

For each metric, this metadata is defined within the PMD, and the PCP arranges for the information to be exported to the performance tools applications that use the metadata when interpreting the values for performance metrics.

## Values for Performance Metrics

The following types of values apply to the performance metrics.

### Singular Performance Metrics

Some performance metrics have a singular value within their PMD. For example, available memory, or the total number of context switches have only one value per PMD, but multiple PMDs.

The metadata describing the metric makes this fact known to applications that process values for these metrics.

### Set-Valued Performance Metrics

Some performance metrics have a set of values or **instances** in each implementing PMD. For example:

- one value for each disk
- one value for each process
- one value for each CPU
- one value for each activation of a given application

When a metric has multiple instances, the PCP framework does not pollute the name space with additional metric names; rather, a single metric may have an associated set of values. These multiple values are associated with the members of an **instance domain**, such that each instance has a unique instance identifier within the associated instance domain. For example, the "per CPU" instance domain may use the instance identifiers 0, 1, 2, 3, ... to identify the configured processors in the system.

Internally, instance identifiers are encoded as binary values, but each PMD also supports equivalent external names for the instance identifiers, and these names are used at the user interface to the PCP utilities.

Multiple performance metrics may be associated with a single instance domain.

Each PMD may dynamically establish the instances within an instance domain; for example, there may be one instance for the metric irix.kerel.percpu.idle on a workstation, but multiple instances on a multiprocessor Challenge server. The PCP arranges for information describing instance domains to be exported from the PMDs to the applications that require this information.

Applications may also choose to retrieve values for all instances of a performance metric, or some arbitrary subset of the available instances.

## Performance Metrics Collection System

The Performance Co-Pilot provides an infrastructure through the Performance Metrics Collection System (PMCS). It unifies the autonomous and distributed PMDs into a cohesive pool of performance data, and provides the services required to create generalized and powerful performance tools.

The PMCS provides the framework that underpins the PMAPI, which is described in Chapter 3, “The Performance Metrics Application Programming Interface (PMAPI).”.

The PMCS is responsible for the following services on behalf of the performance tools developed on top of the PMAPI:

- Instantiation of all metric values.
- Coordination with the processes and procedures required to control the description, collection and extraction of performance metric values from the agents that interface to the Performance Metric Domains (PMDs).
- Archive logging of performance metric values.
- Support for the seamless rewind and replay functionality of the PMAPI.
- Communication and coordination with one or more remote PMCS instances.
- Servicing incoming requests from applications running on a remote system for local performance metric values and metadata.

## PCP Functional Infrastructure

At the highest level, the PCP provides a collection of integrated functions, upon which the various performance utilities are built. These facilities are described in the following sections.

### Automated Reasoning About Performance

Automated reasoning within the Performance Co-Pilot is provided by the Performance Metrics Inference Engine, *pmie*(1), which is an applied artificial intelligence application.

The *pmie* tool accepts a set of expressions, and proceeds to periodically evaluate these against the values of performance metrics from one or more sources. The facilities are very general, and are designed to accommodate the automated execution of a mixture of generic and site-specific performance monitoring and control functions.

The "expressions" may include the following operators and functions:

- Generalized predicate-action pairs, where a predicate is a logical expression, and the action is arbitrary. Supported actions include:
  - execute a *sh*(1) command or script
  - launch a visible alarm with *xconfirm*(1)
  - post an entry to *syslog*
  - echo a message on standard output
- Arithmetic and logical expressions in a C-like syntax.
- Per-expression evaluation frequency, to support both short-term and long-term monitoring and control functions.
- Full support for the semantic richness of the Performance Metrics Collection System (PMCS), and exploitation of the metadata to automate scale conversion and to detect nonsense comparisons in expressions.
- Aggregation functions of *sum*, *avg*, *min*, and *max*, that may be applied to collections of performance metrics values clustered over multiple hosts, or multiple instances, or multiple consecutive samples in time.
- Universal and existential quantification, to handle expressions of the form "for every ...." and "at least one ....".
- Percentile aggregation to handle statistical outliers, such as "for at least 80% of the hosts, ....".
- Macro processing to expedite expression definition.
- Transparent operation against either live-feeds of performance metric values from PMCD instances on one or more hosts, or against archive logs of previously accumulated performance metric values.

The power of *pmie* may be harnessed to automate the most common of the deterministic system management functions that are responses to changes in system performance, for example, to disable a batch stream if the DBMS transaction commit response time at the 90th percentile goes over 2 seconds, or to stop accepting "news" if the average CPU utilization over the past five minutes is above 75%.

At the same time, the power of *pmie* can be directed towards the exceptional and sporadic performance problems; for example, if a "network packet storm" is looming, enable IP header tracing for 10 seconds, and send e-mail to advise that data has been collected and is awaiting analysis. Or, if production batch throughput falls below 50 jobs per hour, activate a pager to the duty systems administrator.

## Performance Visualization With the PCP

For the most interesting and complex problems in performance management, the volume of available information is daunting. Coupled with the power for automated reasoning that *pmie* provides is the considerable potential for human visual processing to absorb, analyze, and classify large amounts of information.

The Performance Co-Pilot has been developed with an assumption that being able to draw three dimensional "pictures" of system performance is a critical requirement, and one that offers vast potential to the human charged with some aspect of performance monitoring and management.

Building on Silicon Graphics' technologies of high-performance graphics at the workstation, OpenGL and OpenInventor, the PCP delivers a range of utilities, services, and toolkits that are designed to provide both basic visualization tools and to foster the crafting of tailored tools to meet the needs of end-user application and operational environments.

Key components to this performance visualization strategy are;

- Time-series strip charts with *pmchart*(1) that allow performance metrics from multiple hosts and multiple Performance Metric Domains to be concurrently displayed on a single correlated time axis.
- Basic three-dimensional models for:
  - per-processor CPU utilization with *mpvis*
  - per-disk spindle activity with *dkvis*
  - NFS request traffic with *nfsvis*
- Visual representation of physical memory allocation on a per-process or per-region-per-process basis, *memvis*.

- A generalized, three-dimensional performance model viewer, *pmview*, that can easily be configured to draw scenes animated by the values of arbitrarily selected performance metrics.
- A sample multi-level performance visualization for ORACLE parallel server configurations, that supports "drill down" navigation and links several different visualization paradigms.

When combined with the VCR and archive services of the Performance Co-Pilot, these visualization tools provide both real-time and retrospective analysis of system performance at many different levels of detail.

## PCP Archive Logging

Within the Performance Co-Pilot, the *pmlogger* utility may be configured to collect archives of performance metrics. The archive creation process is easy and very flexible, incorporating the following features:

- Log creation at either the monitored system (typically a server) or the monitoring system (typically a workstation).
- Concurrent independent logging, both local and remote—the performance analyst can activate a private *pmlogger* instance to collect only the metrics of interest for the problem at hand, independent of other logging on the workstation or the remote host.
- Independent determination of logging frequency for individual metrics or metric instances. For example, log the "5 minute" load average every half hour, the write I/O rate on the DBMS log spindle every 10 seconds, and aggregate I/O rates on the other disks every minute.
- Dynamic adjustment of what is to be logged, and how frequently. This may be used to disable logging or to increase the sample interval during periods of low activity or of chronic high activity (to minimize logging overhead and intrusion).
- Self-contained logs that include all system configuration and metadata required to interpret the values in the log. These logs can be kept for analysis at a much later time, potentially after the hardware and/or software has been reconfigured, and shipped as discrete, autonomous files for remote analysis.

### PCP Logs and the PMAPI

Critical to the success of the PCP archive logging scheme is the fact that the library routines that provide access to real-time feeds of performance metrics also provide access to the archive logs.

Live-feeds and archives are literally interchangeable, with a single PMAPI that preserves the same semantics for both styles of metric source. In this way, applications and tools developed against the PMAPI can automatically process either current or historical performance data.

### Retrospective Analysis Using PCP Logs

One of the most important applications of the archive logging services provided by the Performance Co-Pilot is in the area of retrospective analysis. In many cases, understanding today's performance problems can be greatly assisted by using side-by-side comparison with yesterday's performance.

By routine creation of performance archive logs, you can concurrently replay pictures of system performance for two or more periods in the past.

Archive logs are also invaluable sources of intelligence when trying to diagnose what went wrong, such as for a performance post-mortem.

Since the archives can be replayed against the inference engine (*pmie* is an application that uses the PMAPI), you can automate the regular, first-level analysis of system performance by constructing suitable expressions to capture the essence of common resource saturation problems, then periodically creating an archive and playing it against the expressions.

### Using Archive Logs for Capacity Planning

By collecting performance archives with relatively long sampling periods, or by reducing the daily archives to produce summary logs, the capacity planner can collect the base data required for forward projections.You can estimate resource demands and explore "what if" scenarios by replaying this data with visualization tools and the inference engine.

### PCP Support for the VCR Paradigm

At the PMAPI, the Performance Co-Pilot provides all of the services needed to implement a typical VCR paradigm, namely:

- stop
- rewind
- fast forward
- replay at a variable speed

The applications built over the PMAPI that support a graphical user interface typically extend this paradigm to include a graphical control panel that looks, and behaves, in a manner akin to a VCR.

## PCP Extensibility

Much of the Performance Co-Pilot potential for attacking difficult performance problems in production environments comes from the design philosophy that considers extensibility critically important.

Specifically, the user can tailor the PCP's services and value in the following manner:

1. Easy extension of the PMCS and PMNS to accommodate new performance metrics and new sources of performance metrics.
2. Generalized toolkits that operate on any performance metric.
3. Unrestricted distribution of the PCP components across the network to place the service where it will do the most good.
4. Dynamic adjustment to changes in system configuration.

# PCP Architecture and Operations

This section describes the process architecture and high-level interactions between those processes, as required to support the services of the PCP.

## Local Process Structure

Initially, consider the operation of the PMCS where the performance tools are running on the same host that is providing performance data; that is, there are **no** network operations.

Each PMD requires a definition of methods or procedures for controlling the collection, and extracting the values, for **all** instances of **all** performance metrics maintained in the associated PMD, in addition to exporting the necessary metadata describing these metrics.

Conceptually, we may consider the aggregation of these methods for a particular PMD to constitute a Performance Metrics Domain Agent (PMDA). As we shall see later, the PMDA may be instantiated in one of several ways, but for the moment think of the PMDA as a server that knows how to extract values for, and possibly control the collection of, performance metrics in its own PMD. Further, the PMDA receives and processes requests on behalf of other processes in the PMCS.

For a variety of reasons (local and remote symmetry, reliable operations, access control, information hiding, archive logging, cache management, and so on.) it is prudent to provide a process to coordinate activity within the PMCS. This process, known as the Performance Metrics Coordination Daemon (PMCD), must be running on each host with one or more active PMDAs.

When performance tool applications call down through the PMAPI and demand services from the PMCS, they interact with the PMCD, which in turn may then interact with the PMDAs on behalf of the PMAPI clients. Note that hidden below the PMAPI, the performance tools communicate directly only with the PMCD.

The architecture of this non-distributed deployment is shown in Figure 1-3.

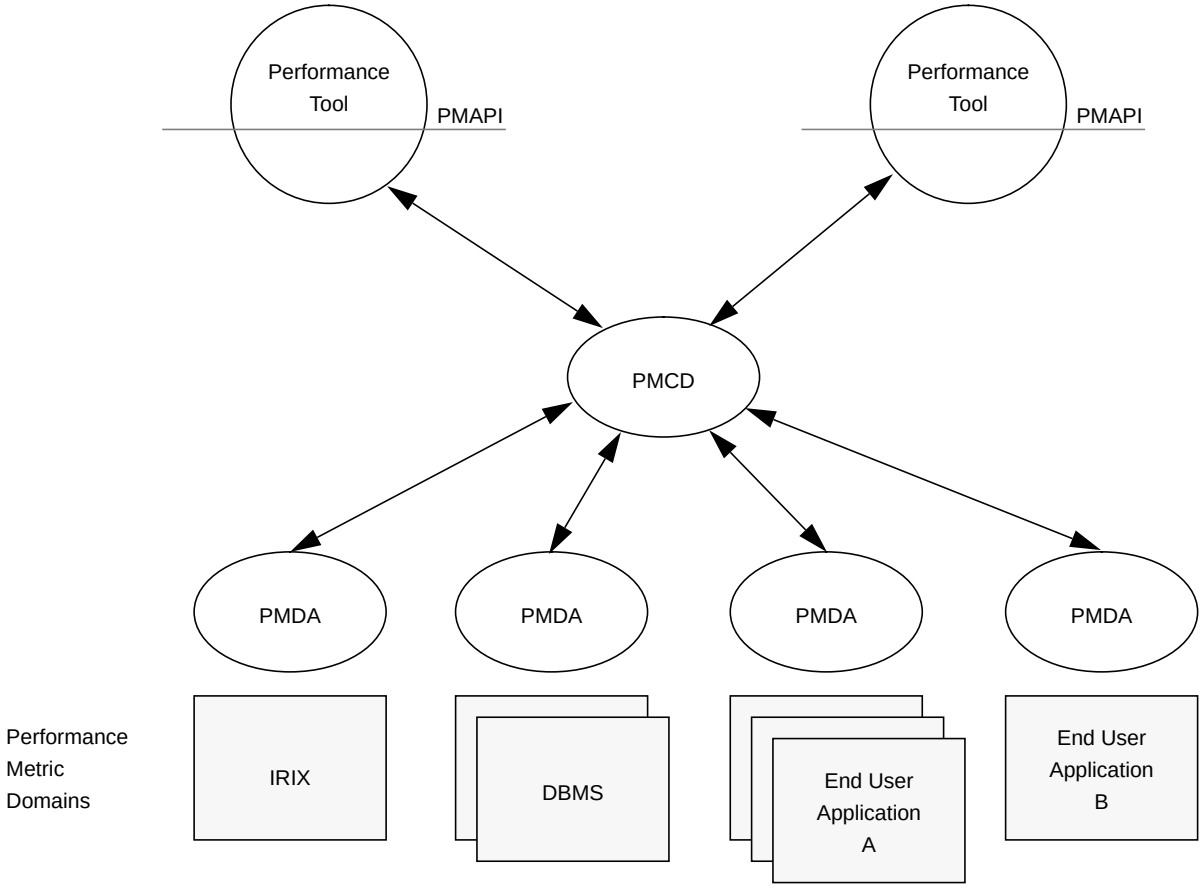


**Figure 1-3** Process Structure for Local Operation

The PMCD executes incoming requests with a “service to completion” model. This simplifies the PMCD (it is single-threaded code, with no possibility of deadlock), and helps ensure the temporal consistency of all results returned from a single client request.

## Distributed Operation of Performance Metrics Collection

In the more general multi-host case, the performance tools execute on one host, while the performance metrics are being collected on one or more remote hosts.

This distributed structure is a simple extension of the local case in which the applications (below the PMAPI) establish and maintain direct communication with the required PMCDs, be they local or remote.

The distributed case is shown in Figure 1-4.

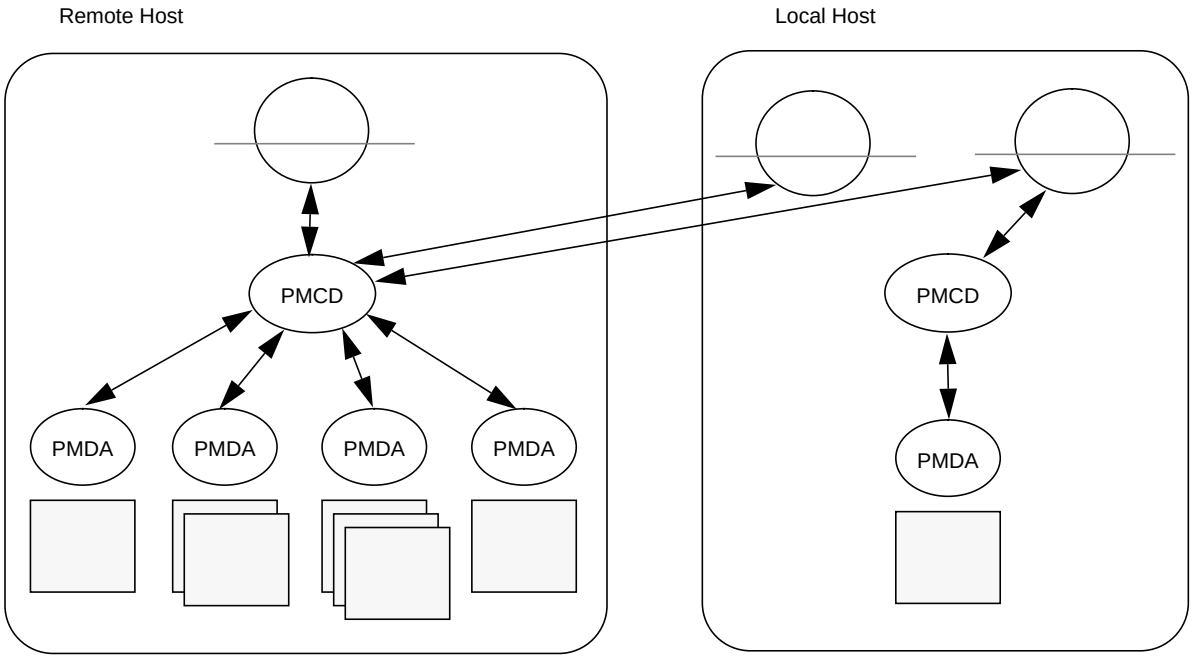


**Figure 1-4** Process Structure for Distributed Operation

The relative number of PMDs on each system suggests that the situation shown in Figure 1-4 corresponds to our most likely assumed scenario: the

remote, large, Challenge system is the one you want to monitor, and the smaller local system is where the performance tools execute and the analysis of the performance data is performed.

Note that if there is **no** active PMDA of interest on the local system (a common situation for workstations), then there is no requirement for a PMCD to be running on the local system.

In all cases, the applications may establish concurrent connections with multiple PMCDs, but they maintain a single "current" context across the PMAPI, so that requests are unambiguously known to require processing in the context of the PMCS on a particular host. Hence, the dynamic choice of the correct PMCD is straightforward. Since the code below the PMAPI must support communication with the local PMCD, minimal additional functionality is required to support communication with multiple and/or remote PMCDs.

Responses from a PMCD go directly to the requesting application.

## PCP Client-Server Architecture

The Performance Co-Pilot follows a classical client-server architecture. For each host on which performance is to be monitored, the PMCD and the associated PMDAs combine to provide an integrated server, delivering values and metadata describing the performance metrics in response to incoming requests.

On each monitoring host (usually a workstation), there are one or more client applications (the performance monitoring tools), each connected to one or more PMCDs.

Each PMCD can service multiple client connections, and each client can maintain multiple PMCD connections.

The distributed nature of the PMCS means the same application can easily monitor system performance for a host next door, or on the other side of the world, or monitor both hosts at the same time.

## Performance Tool and PMCD Interactions

Above the PMAPI, the performance tools are unaware of the process architecture of each PMCS, and hence they remain oblivious to all IPC activity and distributed operations.

Below the PMAPI, the library routines manage the connection(s) to the local and/or remote PMCDs as a set of connection-oriented TCP/IP sockets (AF_INET address family and SOCK_STREAM socket type).

At all times the current context at the PMAPI identifies the correct host, and hence the socket for communication with the required PMCD.

The PMCDs and their clients (the performance application tools) exchange typed messages to encode requests and results. These messages or Protocol Data Units (PDUs) define the complete interface between these processes. All communication is client-server synchronous, in the sense that each PMCD is, by default, idle awaiting incoming request PDUs or new connection requests, and the application's library routines below the PMAPI sends one or more request PDUs and then blocks waiting for a response from the PMCD.

## PMCD-PMDA Protocols

As each request PDU arrives at the PMCD, the request parameters may be used to determine to which PMDA the request (or part thereof) should be routed.

There are some PMDs (the IRIX kernel instrumentation is the most obvious candidate) for which we'd like to be able to include the associated PMDA in the PMCD, and thereby avoid IPC and context switching. For this option to be attractive, the PMCD-PMDA interactions must be frequent enough to produce a useful saving. This efficiency is important because this is happening at the host where the performance data is being collected. These PMDAs are each implemented as a Dynamic Shared Object (DSO) to which the PMCD dynamically attaches at startup.

A second class of PMDA would be those implemented as independent server processes, which block waiting for requests from the local PMCD

(their sole client). For these PMDAs, we need an IPC protocol suited to local process-to-process message passing.

The final style of PMDA is the one required for easy extensibility and infrequently requested metrics. Typically this may be no more than a shell command than can be executed to generate a number on standard output; however, to implement the servicing of continual requests from the PMCD, we provide a simple text-based protocol and a skeletal PMDA implementation as a Bourne shell script. This fully functioning example PMDA is implemented as a generic 400-line script that can easily be adapted for any PMD where metrics are available via executable shell commands; that is, it can be extended or cloned easily to support additional performance metrics or a whole new PMD.

The PMCD supports all three classes of interaction with a PMDA.

Clearly we require a configuration definition to be used in PMCD startup and PMDA initialization to allow the PMCD to know which PMDs are available locally, and which is the correct communication protocol to be used.

## PMCD Startup and Re-Initialization

When the PMCD is started, or it receives a SIGHUP signal, it attempts to confirm the status of all existing PMDA connections, and to try and initiate connections for any new or previously inactive PMDAs. This implies the existence of a configuration file (*/etc/pmcd.conf*) on each host that tells the PMCD about the local PMDAs.

The configuration file is an ASCII text file that specifies the following information for each PMDA:

5. The name of the PMD (mostly for documentation in the configuration file and for PMDA-specific error messages) and the PMD identifier.
6. The communication protocol between the PMDA and the PMCD (DSO, Socket or Pipe).
7. For communication via the DSO protocol, the name of the DSO and the name of the initialization routine within the DSO.

8. For communication via the Socket protocol, the command line required to start the PMDA (omitted if the PMDA is expected to be running already), the IPC protocol, and IPC end-point designation. For example, the AF_INET and TCP/IP port number, or AF_UNIX and FIFO name in the UNIX file system.
9. For communication using the Pipe protocol, the format of the PDU protocol to be used (binary or ASCII) and the command line required to start the PMDA.

The use of asynchronous re-initialization via SIGHUP provides a mechanism for PMDAs to be dynamically added and deleted, without affecting other aspects of the PMCS.

The syntax and semantics of the PMCD configuration file specification language, and examples are fully described in the *pmcd*(1) reference page.

## Timeout Handling and Failure Protocols

The PCP architecture is extensible. It is easy to incorporate user-written PMDAs into the PCP. It is also easy for users to incorporate errors into such PMDAs. Because the communication protocol within the PMCS is synchronous, a misbehaving PMDA could potentially hang a PMCD. If a PMCD sent a PDU to the PMDA requesting some information and the PMDA did not send an appropriate response PDU back (for example, because it was hung, or in a loop), the PMCD could block indefinitely on a read from the PMDA.

A similar situation could arise when a PMCD was sending a PDU to a PMDA. If the PMDA did not read data from the PMCD (presumably due to some failure in the PMDA), and the PMCD attempted to send a PDU larger than the kernel buffer used to communicate with the PMDA, the PMCD would block while writing the PDU.

This kind of failure might even occur in the PDU exchanges between the PMCD and a client program using the PMAPI. For example, if a SIGSTOP is sent to a client of the PMCD (such as a PCP application) while it is waiting for a response from the PMCD, the PMCD is in a similar situation to the one described above when a PMDA has hung. It may have a large PDU to send, with no reader emptying the buffer.

Unfortunately, with DSO-based PMDAs there is little that can be done in such situations. The PMCD actually passes control to the DSO by calling a procedure in the DSO. If the DSO chooses not to return control to the calling PMCD, there is no way for the PMCD to regain control.

The PMCD uses a "dead-hand" timer implemented as an *sproc* to detect a timeout for every non-DSO PDU exchange. The timeout applies to both sending and reception of PDUs. The timeout is also used in situations when the PMCD dispatches multiple PDUs to a collection of PMDAs and then uses the *select* system call to determine when responses from the PMDAs have arrived.

If a PDU exchange times out, it is considered to be a protocol failure.

If any kind of protocol failure occurs, the PMCD terminates its connection(s) with the client or PMDA that was involved. It does this by closing any file descriptors associated with the client or PMDA and marking the client or PMDA as no longer connected.

If a client or PMDA terminates prematurely, any file descriptors that it uses to communicate with PMCD are closed as part of the normal operating system cleanup for the terminating process. The PMCD notices that the client or PMDA has terminated the next time a PDU transfer involving the client or PMDA is attempted, and marks the client or PMDA as disconnected.

Notice that the PMCD’s timeout policy does not extend to killing a client or PMDA. There is currently no PCP defined protocol for killing a hung PMDA, although the usual IRIX service for process termination (such as the *kill* command) may be used.

## Installing and Configuring the PCP

The sections below describe the basic installation and configuration steps necessary to run the PCP on your network.

### PCP Product Structure

There are three software packages shipped as part of Performance Co-Pilot. These are the client software (*pcp_client*), which must be installed on the system that runs the monitoring programs; the shared software, (*pcp_share*) which must be installed on all systems; and the server software, (*pcp_server*) which must be installed on each system to be monitored. You must install all three packages on a system if you wish to monitor that system from its own console.

In a typical deployment, the Performance Co-Pilot would be installed in a server configuration on one or more hosts, from which the performance information could then be collected, and in a client configuration on one or more workstations, from which the performance of the server systems could then be monitored.

The relationship between the PCP client / server capability and the installable packages is shown in Table 1-1.

**Table 1-1** PCP Software Packages Required for Servers and Clients

| PCP Mode | Performance | Required inst Products Data |
|---|---|---|
| Server only | producer of information | *pcp_share* and *pcp_server* |
| Client only | consumer of information | *pcp_share* and *pcp_client* |
| Client and Server | consumer and producer of information | *pcp_share*, *pcp_client* and *pcp_server* |

For complete information on the installable software packages, see the Performance Co-Pilot release notes, available through the *relnotes*(1) or *grelnotes*(1) commands.

## Optional PMDA installation

For hosts that are to act as producers of information (servers, or systems to be monitored), you may wish to configure some of the optional Performance Metrics Domain Agents (PMDAs).

These are installed, one directory per PMDA, below */usr/demos/PerfCoPilot/pmdas*. In each directory there is a *README* file that describes the metrics provided by the PMDA; a *Remove* script to un-configure the PMDA, remove the associated metrics from the PMNS, and restart *pmcd*(1); and an *Install* script to install the PMDA, update the PMNS, and restart *pmcd*(1).

To guard against potential changes between one version of a PMDA and another, it is recommended that you always select "Remove" before "Install".

For example, to install the PMDA for the environmental monitor on a Challenge system, you might enter the following commands as **root** on the system to be monitored:

```
cd /usr/demos/PerfCoPilot/environ
./Remove
./Install
```

If the user creates additional PMDAs, it is recommended that these follow the same file naming conventions and configuration procedures as the optional PMDAs shipped in the standard *pcp_server* product.

## PCP License Constraints

On the monitoring system, all of the display, visualization, and automated reasoning tools are licensed using "nodelocked" NetLS licenses.

Refer to the Performance Co-Pilot release notes for details.

On the monitored systems, the PMDAs, *pmcd*(1) and *pmlogger*(1) instances may be installed and executed without license constraints.

Some of the PCP maintenance tools for updating the PMNS, interrogating the PMCS, dumping an archive log, and so on, are not constrained by any license restrictions.

## Maintaining the PMCD Daemon

On each system to be monitored, you must be certain that the *pmcd*(1) daemon is running. This daemon gathers the statistics that are displayed on the monitoring system. To start the daemon, issue the following commands as **root** on each system to be monitored:

```
chkconfig pcp on
/etc/init.d/pcp start
```

These commands instruct the system to start the daemon immediately, and again whenever the system is booted. It is not necessary to start the daemon on the monitoring system unless you wish to monitor it as well.

To stop *pmcd* immediately on any system, give the command:

```
/etc/init.d/pcp stop
```

Often, if a daemon is not responding on a monitored system, the problem can be resolved by stopping and then immediately restarting a fresh instance of the daemon. If you need to stop and then immediately restart *pmcd* on a monitored system, use the **start** argument provided with the script in */etc/init.d*. The command syntax is:

```
/etc/init.d/pcp start
```

On startup *pmcd* looks for a configuration file named */etc/pmcd.conf*. This file specifies which agents cover which performance metrics domains and how *pmcd* should make contact with the agents. An optional section specifying host-based access controls may follow the agent configuration data. A comprehensive description of the configuration file syntax and semantics can be found in the *pmcd* reference page.

If the configuration is changed, *pmcd* reconfigures itself when it receives the SIGHUP signal. Use the command:

```
killall -HUP pmcd
```

## Tailoring the Primary Archive Logger

On each system for which *pmcd*(1) is active (each system on which performance is to be monitored), there is an option to have a distinguished instance of the archive logger *pmlogger*(1) (the "primary" logger) launched each time *pmcd* is started. This would be typically used to ensure the creation of the archive logs required for on-going system management and capacity planning.

Issue the following command as **root** on each system where you want to activate *pmlogger*:

```
chkconfig pmlogger on
```

The primary logger launches the next time *pmcd* is started. If you wish this to happen immediately, follow up with the command:

```
/etc/init.d/pcp start
```

When started in this fashion, the file */etc/config/pmlogger.options* provides command line options for *pmlogger*, which in turn means that the initial logging state and configuration is specified in the file */usr/lib/pcp/config/pmlogger.config*. Either or both of these files may be modified to tailor the primary *pmlogger* operation to the local requirements.

Refer to the *pmlogger* reference page for more details.

## PCP Client Configuration

The following tools must be configured on each system to be monitored.

### The opsview Tool

The *opsview* tool for monitoring an ORACLE parallel server configuration must be tailored to match the OPS deployment. This may be done using *Configure*, an interactive script, in the *opsview* distribution directory.

First, make sure that the *pcp_share* and *pcp_server* products are installed on both OPS nodes, and the ORACLE PMDA (see */usr/demos/PerfCoPilot/pmdas/oracle7*) is installed on both nodes. On the monitoring system, ensure that the *pcp_share* and *pcp_client* products are installed.

To configure *opsview*, execute the following commands as **root** on the monitoring system:

```
cd /usr/demos/PerfCoPilot/opsview
./Configure
```

Note that for systems that do not have OPS installed, *opsview* still makes an interesting demonstration of the power of the performance visualization capabilities of the Performance Co-Pilot. Indeed, *opsview* can be "configured" for a default "fake" OPS deployment in which your local workstation pretends to be both OPS nodes, and you need not even have ORACLE installed.

### The pmclient Tool

To help you develop tailored performance tools using the services of the PMCS and the functionality of the PMAPI, source code for a simple performance monitoring tool, *pmclient*, is shipped in the *pcp_client* product.

The directory */usr/demos/PerfCoPilot/pmclient* contains the required source and Makefiles to build and install *pmclient*.

## User Interface Terminology

Figure 1-5 and Figure 1-6 show windows labeled with the window terms used in this guide.

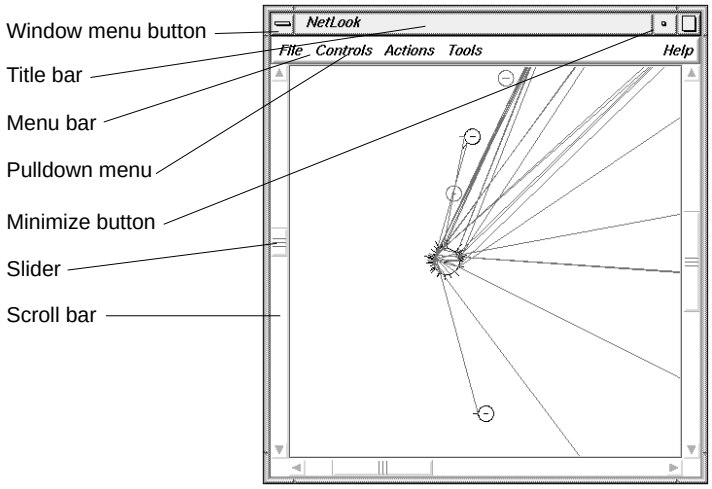


**Figure 1-5** Window Terms

The mouse buttons have these functions:

| | |
|---|---|
| left | Perform most basic tasks: click buttons, select an entry field to type into, select menu choices, select items in a display, select text to modify, and so on. |
| middle | Reposition windows and icons. |
| right | Access popup menus. Popup menus appear when you press the right mouse button in certain locations on the screen. |

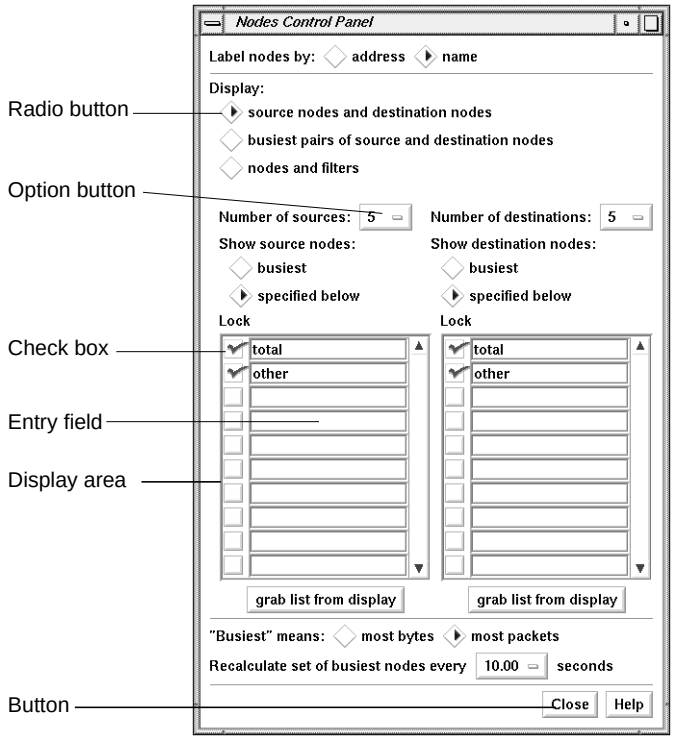


**Figure 1-6** More Window Terms

This guide uses the following terms to describe the use of the mouse:

| | |
|---|---|
| press | Hold down a mouse button. |
| drag | Move the mouse while a mouse button is pressed. |
| click | Press a mouse button and immediately release it without moving the mouse. |
| double-click | Press and release a button twice in quick succession without moving the mouse. |
| select | The term “select” is used in the following ways:<br>• Click the left mouse button on an item line to highlight it. |

- Press the left mouse button in an entry field, drag the cursor across some or all of the text, and release the mouse button. The text becomes highlighted.
- Press the left mouse button on a menu title in a menu bar, move the cursor to a menu choice, and release the mouse button while a menu choice is highlighted.
- To select a traffic line, node, or network in the NetLook main window, double-click on it.

deselect — Click on a highlighted item to turn off the highlighting.

## Common User Interface Operations

The graphical PCP tools have a common look-and-feel for consistent operation and easy switching between tools. This guide assumes that you are familiar with using the mouse, working with windows, and using pulldown and rollover menus. These operations are described in the *IRIS Essentials*.

The sections below explain how to use additional components of the user interface that are common to several of the tools.

### Using Scroll Bars

You can use scroll bars (see Figure 1-7) to change the area and scale of a viewing area and to display different lines or portions of lines in a display area. The size of the slider is proportional to the amount of the total that you are viewing. You operate scroll bars by pressing the left or middle mouse button when the cursor is in the scroll bar. There are several ways to operate the scroll bar:

- Press the left mouse button on the slider, drag the cursor to a new slider position, and release the button.
- Move the slider incrementally by clicking the triangles at each end of the scroll bar.
- Move the slider up or down by positioning the cursor in the trough above or below the slider and clicking the left mouse button.

- Move the slider to a specific position by positioning the cursor at that position and clicking the middle mouse button.

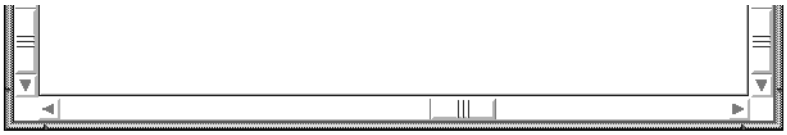

**Figure 1-7** A Horizontal Scroll Bar

## Entering and Removing Text in a Field

Editing text in the entry fields (see Figure 1-8) is the same as editing text in the entry fields of other applications:

- Position the text insertion point by moving the mouse to the entry field and clicking the left mouse button.
- Select (highlight) text by pressing the left mouse button at one end of the text that you want to select and dragging to the other end.
- Select a word, including a space or punctuation-delimited characters, by moving the cursor to the word and double-clicking the left mouse button.
- Select the entire contents of an entry field by moving the cursor over the entry field and triple-clicking the left mouse button.
- Delete selected (highlighted) text by pressing the **<Backspace>** key.
- Delete the character to the left of the insertion point by pressing the **<Backspace>** key.


**Figure 1-8** An Entry Field

## Using Option Buttons

Option buttons (on the left in Figure 1-9) let you select a numeric value from among a predefined set of choices. To use an option button, first press the

option button with the left mouse button. A menu pops up (on the right in Figure 1-9). Move the cursor to your selection and release the mouse button.

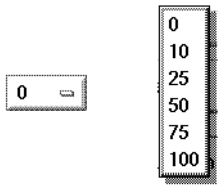


**Figure 1-9** An Option Button and an Option Button Menu

## Using a File Prompter

File prompter windows (like the one in Figure 1-10) are used to specify filenames. You can choose a filename by double-clicking a name in the display area. You can also type the name into the filename entry field and press **<Enter>** or click the *Accept* button to complete your filename selection. You can change directories to the parent of the current directory by clicking the *Up* button, or return to the directory where you started the tool by clicking the *Original* button.


**Figure 1-10** A File Prompter Window

## Using Online Help

The PCP provides many online help files to help you as you learn to use the tools. You access these files from the Help menu in the menu bar of many PCP windows (shown in Figure 1-11 on top) and from the *Help* button that appears in some PCP windows (on the bottom in Figure 1-11).

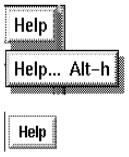


**Figure 1-11** A Help Menu and a Help Button

When you choose “Help...” from a menu or click a *Help* button, a Showcase window appears and displays the first help card.

Some help files contain several cards. Page through these cards using the **<Page Up>** and **<Page Down>** keys in the cluster of six keys just to the right of the **<Backspace>** key, or click the left mouse button on the arrows at the bottom of the pages. Make sure the cursor is in the Help window when you press these keys.

When you finish reading a help file, you can close the Help window just as you close any other window, for instance, by double-clicking the Window menu button in the upper left corner of the window or by selecting “Quit” from the Window menu.

## Product Support

Silicon Graphics provides a comprehensive product support and maintenance program for its products. For further information, contact the Technical Assistance Center at 1-800-800-4SGI.

*Chapter 2*

# PCP Utilities and Tools

This chapter deals with the graphical tools and text-based utilities that make up the pre-developed portion of the Performance Co-Pilot product. The major sections in this chapter include:

- "Common Conventions and Arguments" details some basic standards used in the development of the PCP tools.
- "Monitoring System Performance With the PCP" introduces the utilities and tools provided to monitor basic system performance. These include *pmkstat*, *pmchart*, *pmval*, and *pminfo*.
- "Performance Visualization With the PCP" introduces the tools provided to display performance statistics in an easily visible format. These tools include *dkvis*, *memvis*, *mpvis*, *nfsvis*, *opsview*, and *pmview*.
- "Archive Logging With PCP" describes the process of making a log of selected performance metrics for future review.
- "The Performance Metrics Inference Engine (pmie)" describes the *pmie* utility.
- "Changing PCP Metric Values With pmstore" describes the use of the *pmstore* utility to arbitrarily set or reset performance metric values.

## Common Conventions and Arguments

Many of the utilities provided with the Performance Co-Pilot (PCP) conform to a common set of naming and syntactic conventions for command line arguments and options. This section outlines these conventions and their meaning. The options may be generally assumed to be honored for all utilities supporting the corresponding functionality.

In all cases, the reference pages for each utility fully describe the supported command arguments and options.

### Fetching Metrics From Another Host

The option **-h** *host* is used to direct the utility to make a connection with the *pmcd*(1) instance running on *host*. Once established, this connection serves as the principal source of performance metrics and metadata.

The default source, in the absence of a **-h** option, is usually *pmcd*(1) on the local host.

### Fetching Metrics From an Archive Log

The option **-a** *archive* is used to direct the utility to treat the PCP archive log with the base name *archive* as the principal source of performance metrics and metadata.

Archive logs are created with *pmlogger*(1), and the utilities typically operate with equal facility for performance information coming from either a real-time feed from *pmcd*(1) on some host, or for historical data from an archive log.

The options **-h** and **-a** are mutually exclusive in most cases.

## Alternate Performance Metric Name Spaces

The Performance Metrics Name Space (PMNS) defines a mapping from a collection of external names for performance metrics (convenient to the user) into corresponding internal identifiers (convenient for the underlying implementation).

A default PMNS is supported, but alternates may be specified using the **-n** *namespace* argument.

Refer to the *pmns*(4) and *pmnscomp*(1) reference pages for details of PMNS structure and creation.

## Performance Monitor Reporting Frequency and Duration

Many of the performance monitoring utilities have periodic reporting patterns.

The **-t** *delta* and **-s** *samples* options are used to control the sampling (reporting) frequency, usually expressed as a real number of seconds (*delta*), and the number of *samples* to be reported, respectively. In the absence of the **-s** flag, the default behavior is typically for the performance monitoring utilities to run forever.

## Starting Time for an Archive Log

The **-S** *numsec* option may be used in conjunction with an archive to request that display start at the time *numsec* seconds from the start of the archive.

## Timezone

All utilities that report time of day use the local timezone by default.

The **-z** option forces times to be reported in the timezone of the host that provided the metric values (the monitored host).

The **-Z** *timezone* option may be used to set the TZ variable to a timezone string, as defined in *environ*(5), for example,. **-Z utc** for universal time.

## The VCR Controls in PCP Tools

Those utilities with a graphical user interface typically support a set of standard widgets to control the display, particularly when processing information from an archive log. The controls are modeled after the controls on an industry-standard videocassette recorder or tape recorder. Figure 2-1 shows the VCR control panel.

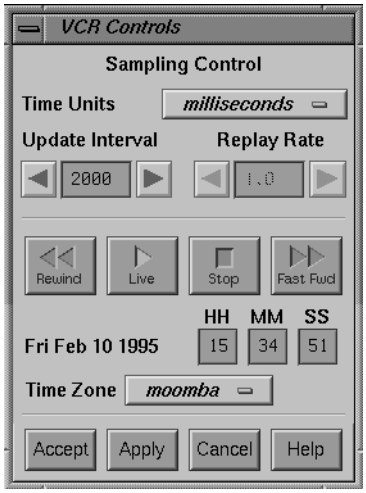


**Figure 2-1** VCR Controls for the PCP Tools

When invoked from a running tool such as *pmchart*, the *Sampling Control* section of the VCR control can be used to alter the rate at which metric values are retrieved, and the Replay Rate is insensitive. When the utility is processing metric values from an archive log, the Update Interval controls the time interval between successive retrievals, and the Replay Rate controls the ratio between the log time interval and the real-time pause in the display - a value of 1.0 replays at a real-time rate equal to the sampling rate; a value of 2.0 replays at approximately double the sampling rate, and so on.

The Timezone option menu may be used to change the timezone for the current time display. The UTC timezone is universal and is hence useful when several archives and / or live sources of data are being displayed in multiple instances of the tools, and comparisons between performance metrics are required to be temporally correlated. Whenever a new source of metrics is opened, whether an archive or live, the timezone at that source of metrics is added to the list in the option menu. The default timezone is that of the local host where the tool is being run.

The *VCR Control Buttons* in the window are used to view an archive log with Stop / Rewind / Replay and Fast Forward capability. They may also be used to pause (stop) and resume (play) the display when metrics are being fetched from a host.

Most PCP tools may be used to replay a log, but meaningful display requires the relevant performance metrics to have been included in the archive log when it was created. The directory */usr/demos/PerfCoPilot/pmlogger* is a repository for useful *pmlogger*(1) configuration files. These files may be used to create archive logs that are viewed with the various PCP utilities.

Most three-dimensional display utilities are based on the generalized *pmview*(1) utility, and you can obtain a VCR control panel from any tool based on *pmview* by using the mouse to click the VCR *Play* button in the upper left-hand corner of the tool window (see Figure 2.9). The same VCR button icon is located in the lower left-hand corner of the *pmchart* window (see Figure 2.2), and selecting it also launches the VCR control panel.

For more information on the VCR controls, see the reference pages for *pmview*(1) or *pmchart*(1). For more information on archive logging, see the section in this chapter titled “Archive Logging With PCP” on page 89 and the section titled “Metric Values Not Available” in Chapter 5.

## Monitoring System Performance With the PCP

The PCP provides a group of commands and tools for measuring system performance. Each tool is described completely in its own reference page. The reference pages are accessible through the *man*(1) command. For example, the reference page for the tool *pmchart*(1) is viewed by giving the command:

```
man pmchart
```

### The pmkstat Command

The *pmkstat* command provides a periodic, one-line summary of system performance. The *pmkstat* command is intended to monitor system performance at the highest level, after which other tools may be used to examine subsystems in which potential performance problems may be observed in greater detail.

Give the command:

```
pmkstat
```

And you see output similar to the following:

```
# hostname load avg: 0.26, interval: 5 sec, Thu Jan 19 12:30:13 1995
runq    | memory    |  system          | disks| cpu
mem swp | free page | scall ctxsw  intr| rd wr|usr sys idl wt
0   0    16268 0     64     19     2396  0  0  0   1   99  0
0   0    16264 0     142    45     2605  0  8  0   2   97  0
0   0    16268 0     308    62     2532  0  1  1   1   98  0
0   0    16268 0     423    88     2643  0  0  1   1   97  0
```

An additional line of output is added every five seconds. The update frequency may be varied using the **-t** *delta* option.

The output from *pmkstat* is directed to standard output, and the columns in the report are interpreted as follows:

runq — Average number of runnable processes in main memory (mem) and in swap memory (swp) during the interval.

| | |
|---|---|
| memory | The free column indicates average free memory during the interval, in Kilobytes. The page column is the average number of page out operations per second during the interval. I/O operations caused by these page-out operations are included in the write I/O rate. |
| system | System call rate (scall), context switch rate (ctxsw) and interrupt rate (intr). Rates are expressed as average operations per second during the interval. |
| disks | Aggregated physical read (rd) and write (wr) rates over all disks, expressed as physical I/O operations issued per second during the interval. These rates are independent of the I/O block size. |
| cpu | Percentage of CPU time spent executing user code (usr), system and interrupt code (sys), idle loop (idl) and idle waiting for resources, typically disk I/O (wt). |

Like all PCP utilities, real-time sources of metrics and archive logs may be used interchangeably. For example

**pmkstat -a foo -z**

uses the archive log *foo* to produce the following:

```
Note: timezone set to local timezone of host “tokyo”

# tokyo load avg: 1.06, interval: 5 sec, Thu Feb  2 08:42:55 1995
 runq  |     memory |     system       |  disks  |      cpu
mem swp|   free page| scall ctxsw  intr|  rd   wr|usr sys idl  wt
  0   0    4316    0    195    64  2242   32   21   0   3   8  89
  0   0    3976    0    279    86  2143   50   17   0   5   8  87
  1   0    3448    0    186    63  2304   35   14   0   4   9  87
  0   0    4364    0    254    81  2385   35    0   0   4   9  87
  0   0    3696    0    266    92  2374   41    0   0   3   9  88
  0   0    2668   42    237    81  2400   44    2   1   4   7  89
  0   0    4644  100    206    68  2590   25    1   0   3   5  91
  0   0    5384    0    174    63  2296   32   22   0   2   8  89
  0   0    4736    0    189    65  2197   31   28   0   3   8  89
_sample Fetch: End of PMCS log file
```

For complete information on the usage and syntax of *pmkstat*, see the *pmkstat* reference page.To launch a tutorial, enter these commands in order:

1. **`cd /usr/demos/PerfCoPilot/Tutorial`**

2. **`showcase -v Tutorial.sc`**

## The pmchart Tool

The *pmchart* utility supports interactive selection and plotting of trends over time for arbitrarily selected performance metrics from one or more hosts and one or more domains of performance metrics. When you issue the command

```
pmchart
```

you see the window shown in Figure 2-2.

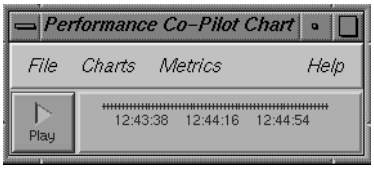


**Figure 2-2** The pmchart window

Normally, *pmchart* operates in “live” mode where performance metrics are fetched in real time and plotted against a time axis. The user can choose performance metrics and monitor the current values for these metrics from any host that is accessible on the network and has the *pmcd*(1) server running. In addition, *pmchart* can also replay archive logs of performance metrics created by *pmlogger*(1). It is possible to replay an archive while *pmlogger*(1) is creating the archive; when the current end of the archive is reached a VCR STOP is forced; note however it is not generally possible to creep forward and replay metrics just behind real-time because *pmlogger*(1) uses buffered I/O.

The reference page for *pmchart* explains how to configure charts based on performance metrics. Once charts have been configured and applied, the charts are placed in the window, as shown in Figure 2-3.

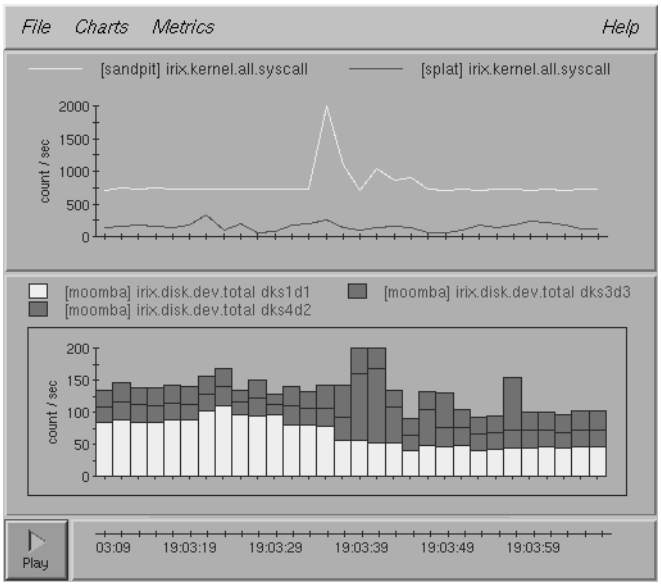


**Figure 2-3** The pmchart Window With Two Charts Configured

All metrics in the Performance Metrics Name Space (PMNS) with numeric value semantics can be graphed, and metrics from multiple hosts may be plotted on a common time axis. The *pmchart* utility examines the semantics of selected metrics, and where sensible, uses the metadata provided by the Performance Metrics Collection Subsystem (PMCS) to convert fetched metric values to a rate before plotting. In the case where different metrics are plotted in the same chart (for example, against a common Y-axis), *pmchart* may also scale metric values where necessary, to produce comparable values with common units and scale.

By default, *pmchart* initially allows the user to select metrics to be plotted from the local host. However, the graphical user interface allows other hosts or archives to be chosen at any time as alternate sources of performance metrics.

When replaying archive logs, the user may interactively control the current replay time, the direction of replay, and the replay rate, using a VCR-like control panel, as shown in Figure 2-3.

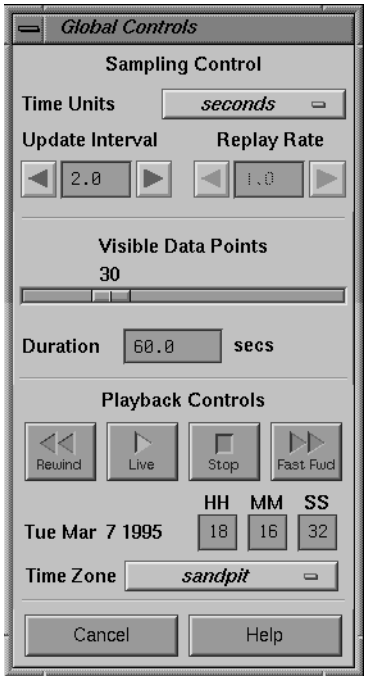


**Figure 2-4** The Global Control Window With VCR Controls

This mode of operation is particularly useful for retrospective comparisons and for *post-mortem* analysis of performance problems, where a remote system is not directly accessible or a performance analyst is not available on site.

### Mouse Controls

The *pmchart* tool uses the left and right mouse buttons as follows:

| | |
|---|---|
| Left Button | Apart from interacting with the menus and dialogs, the left mouse button is also used to select the current chart by clicking anywhere on the desired chart. At all times, the current chart has a border drawn around the graph area and the legend (if visible) is rendered in red. |
| Middle Mouse | The middle mouse button is not used. |
| Right Mouse | The right mouse button is used to display metric values in a popup dialog. Clicking the right mouse in the graph drawing area of any chart displays information about the nearest metric and its value, as plotted, at that point. |

### pmchart Metric Selection

The Metric Selection dialog window allows interactive navigation of the Performance Metrics Name Space, giving the user the ability to choose metrics, change the current host, select metric instances, and then plot current or archived metric values on a common time axis. This dialog is shown in Figure 2-5.

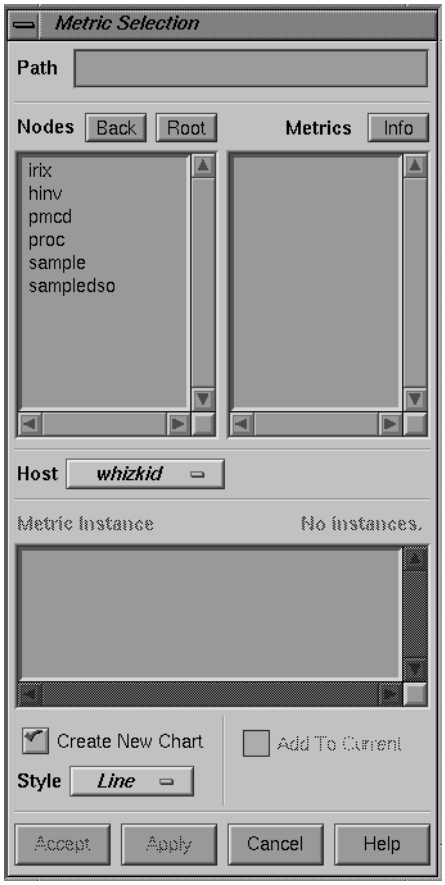


**Figure 2-5** The Metric Selection Dialog

If you enter a partial metric specification in the *path* field in the Metric Selection dialog, you can avoid having to navigate through the PMNS to

select the metrics you need. For example, if you enter the path *irix.network.interface*, the window changes dynamically, as shown in Figure 2-6.

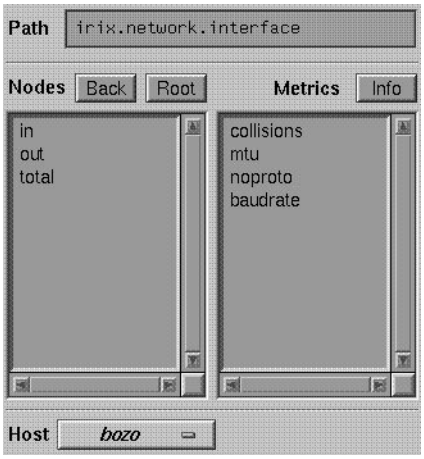


**Figure 2-6** Further Metric Selection

You can also continue the selection process by clicking the subcategory or metric you desire, as shown in Figure 2-7.

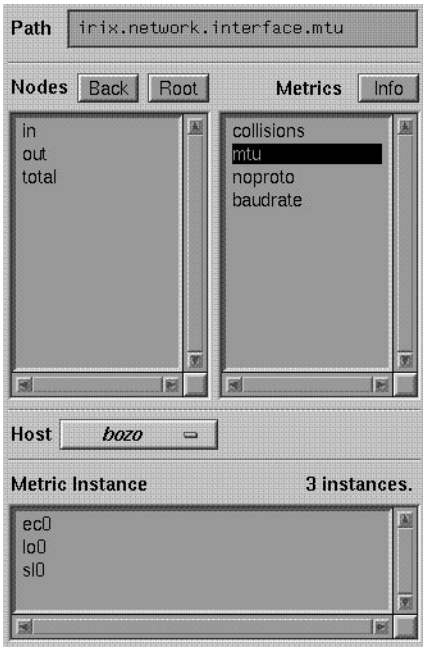


**Figure 2-7** Selecting a Final Metric

Finally, you may have to select from several instances of a metric. In the example shown in the above figures, you are monitoring the Maximum Transmission Unit on a network interface. On the system being monitored, there are three network interfaces configured. You must select an interface, as shown in Figure 2-8.

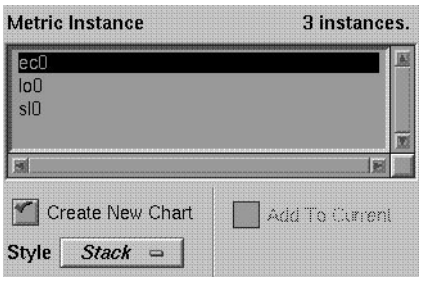


**Figure 2-8** Selecting a Metric Instance

The annotated examples in the *pmchart* chapter of the Performance Co-Pilot Tutorial provide a guided illustration to the user's typical interactions with *pmchart*. To launch this chapter of the Tutorial, enter these commands:

1. **`cd /usr/demos/PerfCoPilot/Tutorial`**
2. **`showcase -v Tutorial.sc`**

Once the showcase application has opened the Tutorial, page down to the Tutorial "Home" page, select the "Monitoring trends - *pmchart*" heading, and work through the examples.

## The pmval Command

The *pmval* command dumps the current values for the named performance metrics. For example, the command

```
pmval proc.nprocs
```

reports the value of the performance metric *proc.nprocs* once per second (by default), and produces output similar to this:

```
metric:    proc.nprocs
host:      localhost
semantics: instantaneous value
units:     none
samples:   indefinite
interval:  1.00 sec

         73
         72
         70
         75
         75
```

In the above example, the number of processes running on the monitored system is reported once per second.

Where the semantics of the underlying performance metrics indicate that it would be sensible, *pmval* reports the rate of change or resource utilization.

For example, the command

```
pmval -h moomba -t 5 -s 4 irix.kernel.percpu.cpu.idle
```

reports idle processor utilization for each CPU on the remote host *moomba*, over four samples, each five seconds apart. This produces output of the form:

```
metric:    irix.kernel.percpu.cpu.idle
host:      moomba
semantics: cumulative counter (converting to rate)
units:     millisec (converting to time utilization)
samples:   4
interval:  5.00 sec

       cpu0        cpu1        cpu2        cpu3
     0.8193      0.7933      0.4587      0.8193
     0.7203      0.5822      0.8563      0.7303
     0.6100      0.6360      0.7820      0.7960
     0.8276      0.7037      0.6357      0.6997
```

Similarly, the command

```
pmval -t 3 -i dks0d1 irix.disk.dev.read
```

reports disk I/O read rate for just the disk */dev/dsk/dks0d1* every 3 seconds. You see output similar to the following:

```
metric:    irix.disk.dev.read
host:      localhost
semantics: cumulative counter (converting to rate)
units:     count (converting to count / sec)
samples:   indefinite
interval:  3.00 sec

     dks0d1
      33.67
      48.71
      52.33
      11.33
      2.333
```

The **-r** flag may be used to suppress the rate calculation (for metrics with counter values) and display the raw values of the metrics.

The *pmval* command is documented completely in the *pmval* reference page. There are annotated examples of the use of *pmval* in the PCP tutorial (see page 55).

## The pminfo Command

The *pminfo* command displays various types of information about performance metrics available through the facilities of the Performance Co-Pilot.

Without any options, *pminfo* verifies that the specified metrics exist in the namespace, and echoes those names. Metrics may be specified as arguments to *pminfo* using their full metric names. For example, the command

```
pminfo hinv.ncpu irix.network.interface.total.bytes
```

returns the following response:

```
hinv.ncpu
irix.network.interface.total.bytes
```

A group of related metrics in the namespace may also be specified. For example to list all of the *hinv* metrics you would use the command

```
pminfo hinv
```

The response to this command is:

```
hinv.ncpu
hinv.cpuclock
hinv.dcache
hinv.icache
hinv.secondarycache
hinv.physmem
hinv.pmeminterleave
hinv.ndisk
```

If no metrics are specified, *pminfo* displays the entire collection of metrics. This can be useful for searching for metrics, when only part of the full name is known. For example, the command

```
pminfo | grep nfs
```

returns the following response:

```
irix.nfs.client.badcalls
irix.nfs.client.badcalls
irix.nfs.client.calls
irix.nfs.client.nclget
irix.nfs.client.nclsleep
irix.nfs.client.reqs
irix.nfs.server.badcalls
irix.nfs.server.calls
irix.nfs.server.reqs
irix.nfs.client.badcalls
irix.nfs.client.calls
irix.nfs.client.nclget
irix.nfs.client.nclsleep
irix.nfs.client.reqs
irix.nfs.server.badcalls
irix.nfs.server.calls
irix.nfs.server.reqs
```

The **-d** option causes *pminfo* to display descriptive information about metrics (refer to the *pmLookupDesc*(3) reference page for an explanation of this metadata information). Consider the following command and its response:

```
pminfo -d proc.nprocs irix.disk.dev.read irix.filesys.free
```

```
proc.nprocs
    Data Type: 32-bit int  InDom: PM_INDOM_NULL 0xffffffff
    Semantics: instant  Units: none
irix.disk.dev.read
    Data Type: 32-bit unsigned int  InDom: 1.2 0x400002
    Semantics: counter  Units: count
irix.filesys.free
    Data Type: 32-bit int  InDom: 1.7 0x400007
    Semantics: instant  Units: Kbyte
```

The **-f** option to *pminfo* forces the current value of each named metric to be fetched and printed. In the example below, all metrics in the group *hinv* are selected:

```
pminfo -f hinv

hinv.ncpu
    value 1
hinv.cpuclock
    value 100
hinv.dcache
    value 8192
hinv.icache
    value 8192
hinv.secondarycache
    value 1048576
hinv.physmem
    value 64
hinv.pmeminterleave
    value 0
hinv.ndisk
    value 1
```

If the metric has an instance domain, the value associated with each instance of the metric is printed:

```
pminfo -f irix.filesys.mountdir
irix.filesys.mountdir
    inst [1 or "/dev/root"] value "/"
    inst [2 or "/dev/usr"] value "/usr"
    inst [3 or "/dev/dsk/dks3d2s2"] value "/proj"
    inst [4 or "/dev/dsk/dks3d3s2"] value "/disk6"
    inst [5 or "/dev/dsk/dks4d3s2"] value "/disk4"
    inst [6 or "/dev/dsk/dks1d3s3"] value "/dist"
    inst [7 or "/dev/dsk/dks1d3s2"] value "/disk3"
    inst [8 or "/dev/dsk/dks1d2s3"] value "/home"
    inst [9 or "/dev/dsk/dks1d2s2"] value "/disk2"
```

The **-t** option displays the one-line help text associated with the selected metrics. The **-T** option prints the more detailed help text.

The **-m** option prints the Performance Metric Identifiers (PMIDs) of the selected metrics. This is useful for finding out which PMDA supplies the metric. For example, the output below identifies the PMDA supporting domain 4 (the left-most part of the PMID) as the one supplying information for the metric *environ.extrema.mintemp*.

```
pminfo -m environ.extrema.mintemp
environ.extrema.mintemp PMID: 4.0.3
```

The **-M** option prints the PMID as an integer and in hexadecimal as well as in the dotted notation.

The **-v** option verifies that metric definitions in the name space correspond with supported metrics, and checks that a value is available for the metric. Descriptions and values are fetched, but not printed. Only errors are reported.

Some instance domains are not enumerable. That is, it is not possible to ask for all of the instances at once. Only explicit instances may be fetched from such instance domains. This is because instances in such a domain may have a very short lifetime or the cost of obtaining all of the instances at once is very high. The *proc* metrics are an example of such an instance domain. The **-f** option is not able to fetch metrics with non-enumerable instance domains; however, the **-F** option tells *pminfo* to obtain a snapshot of all of the currently available instances in the instance domain and then to retrieve a value for each.

Complete information on the *pminfo* command is found in the *pminfo* reference page. There are examples of the use of *pminfo* in the PCP tutorial (see page 55)**.**

## Performance Visualization With the PCP

Several graphical tools are provided by the Performance Co-Pilot (PCP) to assist you in visualizing performance on your monitored systems. Each tool is described completely in its own reference page. The reference pages are accessible through the *man*(1) command. For example, the reference page for the tool *mpvis*(1) is viewed by giving the command

```
man mpvis
```

## The pmview Tool

The *pmview* tool is a generalized 3D Inventor™ application that supports dynamic displays of clusters of related performance metrics as utilization blocks (or towers) on a common base plane. The *pmview* tool is the basis for the *dkvis*, *mpvis*, and *nfsvis* tools.

The Open Inventor 3D Toolkit is an object-oriented toolkit that simplifies and abstracts the task of writing graphics programming into a set of easy-to-use objects. Inventor run-time support is distributed with the IRIX operating system software.

The *pmview* command displays performance metrics as multicolored blocks arranged in a grid on a grey baseplane. The height of each block changes as the value of its corresponding metric (or metric instance) changes. Labels identify each metric or group of metric instances as shown in Figure 2-9.

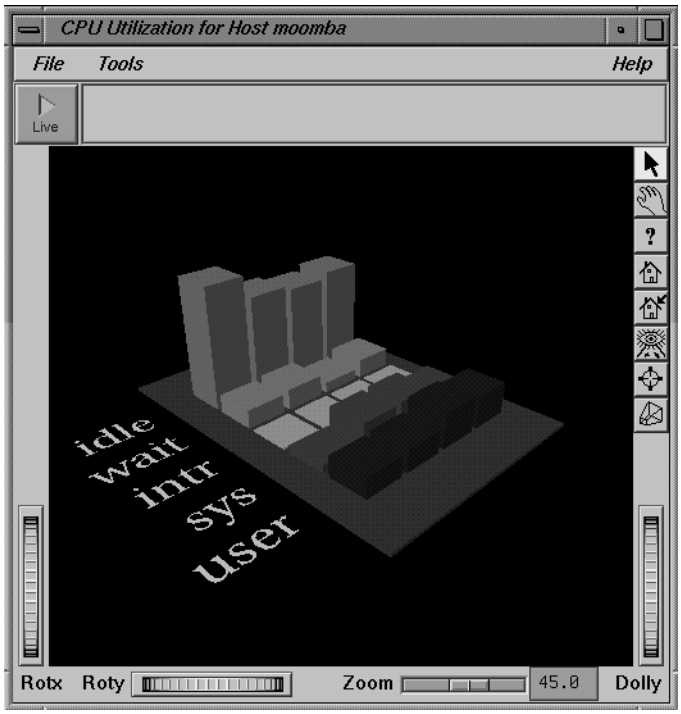


**Figure 2-9** A pmview Window

A configuration file is used to specify which metrics and instances should be displayed, and their colors, positions, textual labels, and so on. The contents of the configuration file are described in detail below.

Around the outside of the 3D scene is a collection of control icons. Thumbwheels may be used to rotate the scene around the X and Y axes or to “dolly” the virtual camera used to view the scene (move it closer or further away). A slider allows the viewer to zoom in or out. On the right edge is a collection of buttons that perform various actions on the view of the 3D scene or change the way the user interacts with it. Above the scene is a button similar to those on a VCR, and an area where informative text is displayed and a menu bar.

When the arrow tool (on the right edge of the scene) is selected and the mouse pointer is moved over one of the blocks in the scene, the name of the corresponding metric (or instance) is displayed in the information window along with its value. The value is shown as the raw value (for non-counter metrics) or the average rate during the most recent update interval (for counters). In addition, the percentage utilization for the metric is shown in brackets; this is the percentage of the maximum value, where the maximum is defined in the configuration file - refer to the reference page for *pmview*(1) for details of the configuration file format. As a special case, metrics which have time-counter semantics (for example, cpu utilization metrics) are displayed as a percentage of time. This is because a time counter converted to a rate has no dimensions (it becomes a ratio). The utilization value, clipped to 100%, is used to normalize the height of the bar for the metric.

As the mouse pointer is moved over other blocks, the details of the block appear in the information window.

Clicking a block with the left mouse button selects the block. When a block is selected, its details are displayed in the information window regardless of whether or not the mouse pointer is over the block. To deselect the block, click the left mouse button on the grey base plane underneath the blocks.

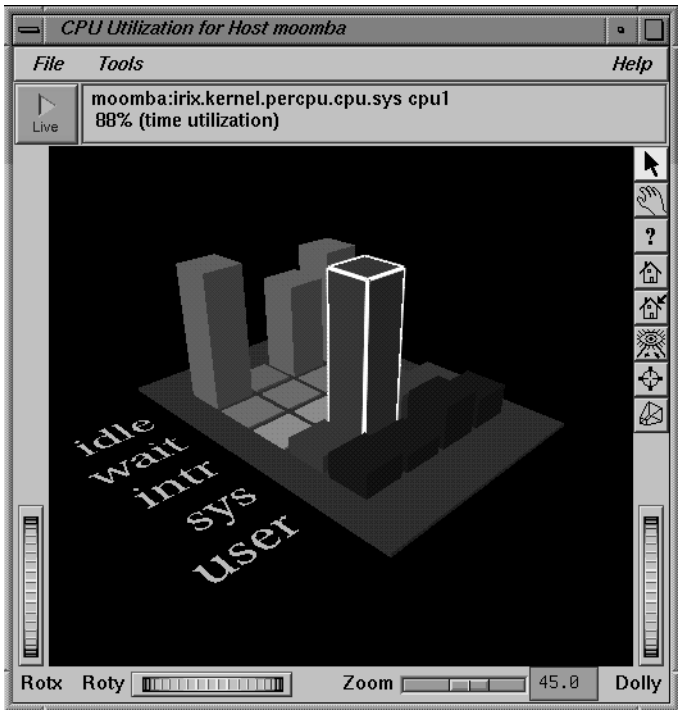


**Figure 2-10** A pmview Window With a Block Selected

When the hand tool is selected, the mouse may be used to pan, rotate, and zoom the 3D scene. Dragging the mouse with the left mouse button down rotates the scene. If the button is released while the mouse is still moving, the scene continues to spin until either the left or middle mouse buttons is pressed or another tool is selected. Dragging the mouse with the middle mouse button down pans the scene. Dragging the mouse with the left and middle buttons down dollies in and out (moves forward and backward).

Holding down the control key while dragging with the left mouse button down rolls the scene around an axis perpendicular to the screen. Pressing and releasing the “s” key, then clicking on an object with the left mouse button, ‘‘seeks’’ to that object; that is, moves the object to the center of the viewing area.

The right mouse button brings up a menu of other viewing options. In addition to various other operations, the Preferences item at the bottom of the menu may be used to enable stereo viewing.

The escape key toggles between the arrow tool and the hand tool.

Additionally, the following tools are available (from top to bottom as seen on your *pmview* window):

- The *question mark* tool brings up help if you have it installed. To install online help, use *inst*(1) to install the *inventor_eoe.sw.help* package from your default IRIX distribution. See the Performance Co-Pilot release notes for more information on prerequisite subsystems.
- The *home* tool returns you to the original (home) scene orientation. The home tool with an arrow saves the current scene as the new home scene orientation.
- The *eye* tool resizes the scene so that it completely fits in the 3D viewing area.
- The *cross-hairs* tool moves an object to the center of the screen (similar to the “s” key and the hand tool as described above) but works only if the hand tool is already selected.
- The *perspective* tool toggles between perspective and orthogonal projections of the scene.

The menu bar at the top of the window contains the File menu, which allows you to quit; the Tools menu, for launching related tools for examining performance metrics; and the Help menu.

Currently, the Tools menu contains items for:

*dkvis*, *mpvis* and *nfsvis*
: *pmview*-based tools for visualizing disk activity, cpu utilization, and NFS statistics

*pmkstat*
: A tool that displays a high-level textual summary of system performance.

*pmchart*
: A tool for graphically displaying and correlating time-series trends for performance metrics.

*pmval* A tool that displays the values of performance metrics textually.

If a block is selected when *pmchart* is chosen from the Tools menu, *pmchart* starts with a graph displaying the metric or instance corresponding to the selected block. The *pmval* tool may be used only if a block in the scene has been selected.

At the top of the window, next to the information window, is the VCR button. The *pmview* tool can display data from an archive and can display the current values of metrics (live data). Clicking on the VCR button brings up the VCR Controls dialog, as shown in Figure 2-11.

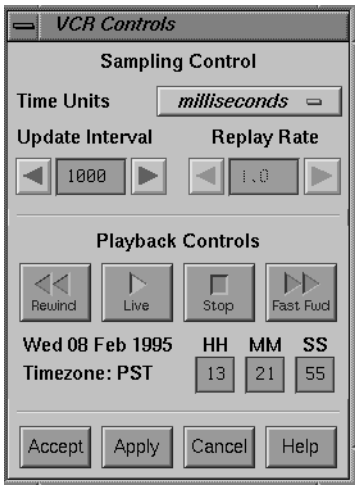


**Figure 2-11** The VCR Controls Dialog

For live data, the Sampling Control section of the dialog box is used to specify how often the 3D scene is updated. The update interval is specified by the Time Units and the Update Interval. The Playback controls allow you to pause and resume display of the live data with the Stop and Live buttons, respectively. The time at which the displayed data was collected is displayed under the VCR buttons in the dialog box. The Rewind and Fast Fwd buttons are not available for live operation.

The **-a** option to *pmview* indicates that data is to be read from the specified archive. For example, the following command specifies an archive:

```
pmview -a tue09dec94.gonzo ...
```

In archive mode, the Live button on the *pmview* window becomes a Play button, the Rewind and Fast Fwd buttons are enabled, and the Replay Rate may be adjusted to change the rate at which playback occurs, as shown in Figure 2-12.

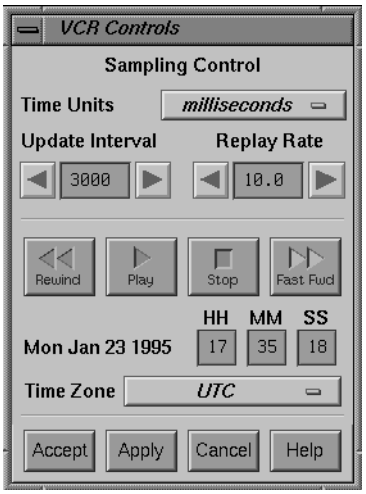


**Figure 2-12** A VCR Dialog in Archive Mode

If a selection is made from the Tools menu while *pmview* is in archive mode, the tool selected is started using the same archive that *pmview* is using. The new tool starts displaying data from the same point in time in the archive that *pmview* is using.

### Creating Custom Visualization Tools With pmview

At startup time, a configuration file is read that specifies

- The geometry for the scene to be displayed by *pmview.*
- The association between the visual appearance of the “blocks” and particular performance metrics.

The configuration file specified with the **-c** option, or else the configuration is read from the standard input.

The format for this configuration file is as follows:

1. Lines beginning with a “#” are treated as comments, and ignored.
2. Words are delimited by white space (space or tab).
3. Each line specifies a block in the scene in the default orientation, according to the following parameters (in order):
    - An X co-ordinate on the base plane (0 in the top left, increasing towards the bottom left).
    - A Z co-ordinate on the base plane (0 in the top left, increasing towards the top right).
    - A color, encoded as three real numbers in the range 0.0 to 1.0, representing the saturation of red, green, and blue.
    - A scaling or normalization factor. If the performance metric is a “counter,” then this should be the maximum expected rate of change in the counter per second; if not, the expected maximum absolute value of the metric.
    - A label to be drawn to the left of the block (preferably short; spaces are not allowed, and underscores are silently removed). The special value “-” (hyphen) may be used to suppress the display of any label associated with this metric.
    - An optional hostname, followed by a colon; if missing, the default host (**-h** or localhost) or archive (**-a**) is used as the source of metrics. As a special case, a hostname containing a “/” (slash) is interpreted as the name of an archive log.
    - The name of a performance metric.
    - An optional comma-separated list of instance identifiers; if no instances are specified, this implies all instances. For every specified instance of the metric at the selected source, a new block is created with the same attributes, except the Z coordinate is increased by one for each successive instance, and the label (if any) is drawn once to the left of the first instance.

For example, the following specification produces a scene similar to that shown in Figure 2-13.

```
1 0  0.0 0.0 0.8    400.0    colour   sample.colour
3 0  0.8 0.8 0.0     20.0    pdus     sample.pdu
3 2  0.8 0.0 0.0     10.0    -        sample.recv_pdu
3 3  0.0 0.8 0.0     10.0    -        sample.xmit_pdu
5 0  1.0 1.0 1.0   2000.0    drift    sample.drift
5 6  0.8 0.0 0.8    100.0    step     sample.step
7 2  0.0 0.8 0.8     10.0    load     irix.kernel.all.load
```

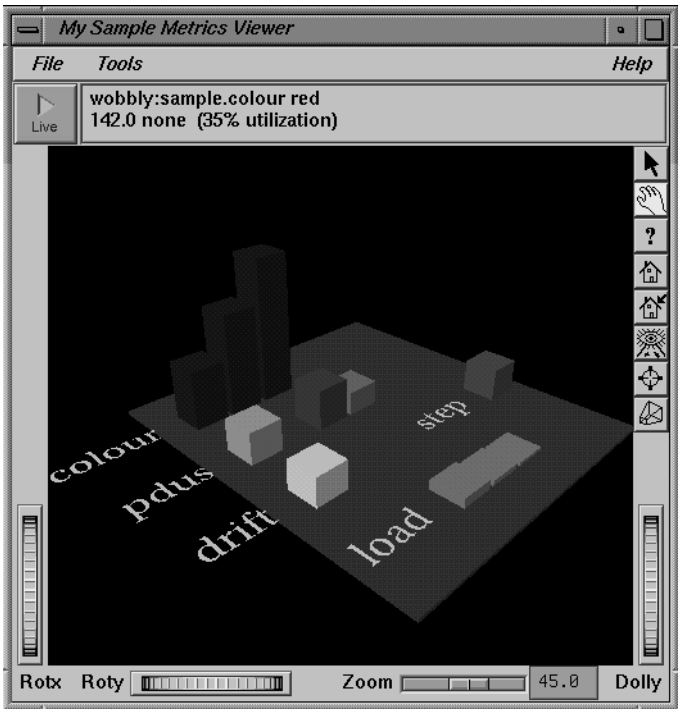


**Figure 2-13** A Custom pmview Example

## The dkvis Disk Visualization Tool

The *dkvis* tool is a graphical disk device utilization viewer, displaying a bar chart showing disk activity. When you give the *dkvis* command, you see a bar chart displaying activity on each disk on the monitored system. You see a window similar to the one shown in Figure 2-14

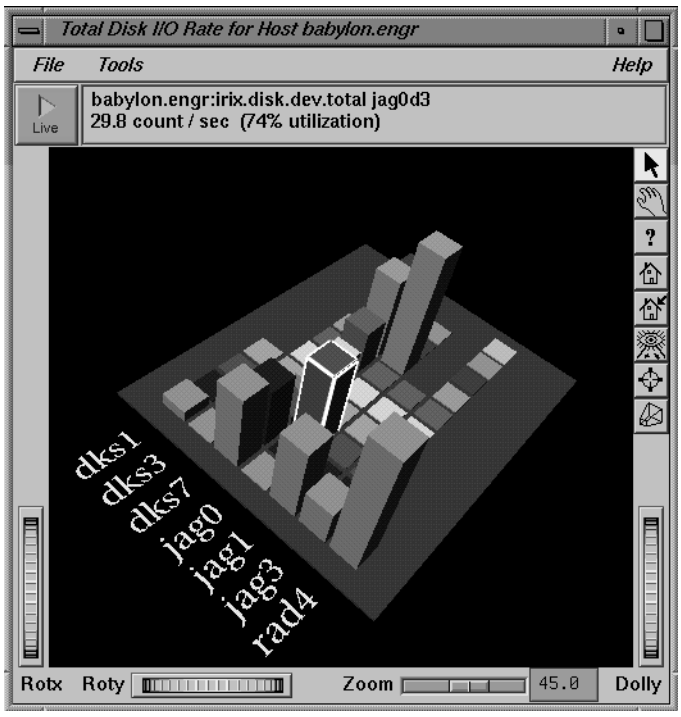


**Figure 2-14** The dkvis window

Each row of blocks on the base plane represents the group of disks connected to a single disk controller (or host adaptor or SCSI bus). The label for each row is generated from the characters common to the names of all of the disks on the controller. For example, in the diagram above, the disks in the row labelled *jag0* (the same row as the selected block for *jag0d3*) are *jag0d1*, *jag0d2*, *jag0d3*, *jag0d4*, *jag0d9*, *jag0d10*, *jag0d11*, *jag0d12*, and *jag0d14*.

The command-line options for *dkvis* are the same as the "common" ones for *pmview*. *dkvis* normally displays the total number of I/O operations per second (IOPS). The **-r** option may be used to restrict the display to just the read operations or **-w** may be specified for just the writes.

The *dkvis* command expresses the utilizations in the information window as percentages of some maximum expected rate (clipped to 100%). The **-m** flag allows you to override the default maximum value. This is useful if all of the utilizations are small compared to the maximum. In such a situation specifying a smaller maximum has the effect of magnifying the differences between the blocks. Similarly, if some of the blocks are almost always at full height, there is a good chance that they are being clipped. A suitable value for the **-m** option can be determined by clicking on the blocks in question, observing the values displayed in the information window for a while, and adding about 10% to the highest value observed.

Complete information on the *dkvis* command is available in the *dkvis* reference page. The PCP tutorial contains additional examples on the use of *dkvis* (see page 55).

## The mpvis Processor Visualization Tool

The *mpvis* tool is a graphical multiprocessor activity viewer, displaying a bar chart that shows processor activity. When you enter the *mpvis* command, you see a bar chart displaying activity on each processor on the monitored system. You see a window similar to the one shown in Figure 2-15:

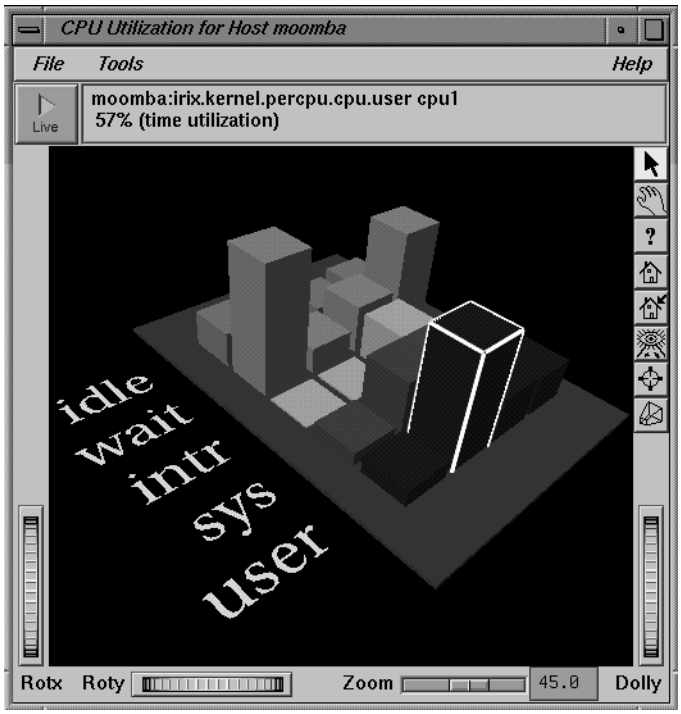


**Figure 2-15** The mpvis Window

Figure 2-15 shows *mpvis* monitoring a machine with four CPUs. Notice in the figure that CPU 0 and CPU 3 are waiting for I/O; CPU 1 is selected and is spending more than half the time executing user code; and CPU 2 is more balanced in its work.

The display contains five labeled rows of blocks, which represent the breakdown of the activity of a single CPU into five states. There is one

column of five blocks for each CPU on the system being monitored. These five states are:

| | |
|---|---|
| idle | no activity |
| wait | like idle but waiting for I/O |
| intr | processing an interrupt |
| sys | executing in the IRIX kernel |
| user | executing user code |

Complete information on the *mpvis* command is available in the *mpvis* reference page.The PCP tutorial contains additional examples on the use of *mpvis* (see page 55).

## The nfsvis NFS Activity Visualization Tool

The *nfsvis* tool is a graphical NFS (Network File System) activity viewer, displaying a bar chart that shows NFS request activity on the monitored system. NFS is optional software, and may not be present on all systems or at all sites.

When you give the *nfsvis* command, you see a bar chart displaying NFS load on the monitored system. You see a window similar to the one shown in Figure 2-16:

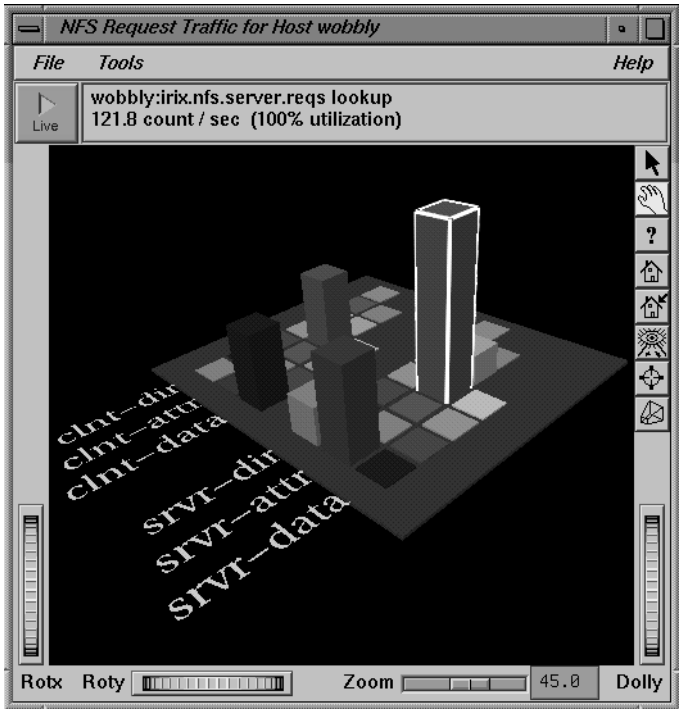


**Figure 2-16** The nfsvis Window

The statistics are broken into two groups: server statistics (requests from other machines for the NFS server on the machine being monitored) and client statistics (requests made by the monitored machine to NFS servers on other machines). The statistics in each of these two groups are the same, save that the client group is for outgoing requests and the server group is for incoming requests. Within each group, the requests are further broken down into requests relating to data within files;, requests for directory operations (for example, to rename a file); and requests involving other attributes of files.

Complete information on the *nfsvis* command is available in the *nfsvis* reference page.The PCP tutorial contains additional examples on the use of *nfsvis* (see page 55).

## The memvis Memory Usage Visualization Tool

The *memvis* tool is a graphical memory usage viewer. The display is updated every 0.5 seconds (the update interval may be changed using the **-i** command line flag) to provide a real time view of memory use. There are four basic viewing modes: "Physical Memory Breakdown," "Total Sizes of Processes," "Resident Sizes of Processes," and "Resident Mappings." In each of these modes, you can select the memory used by a set of processes or a detailed breakdown of memory use by a single process.

When you give the command

```
memvis
```

you see a window similar to the one shown in Figure 2-17.

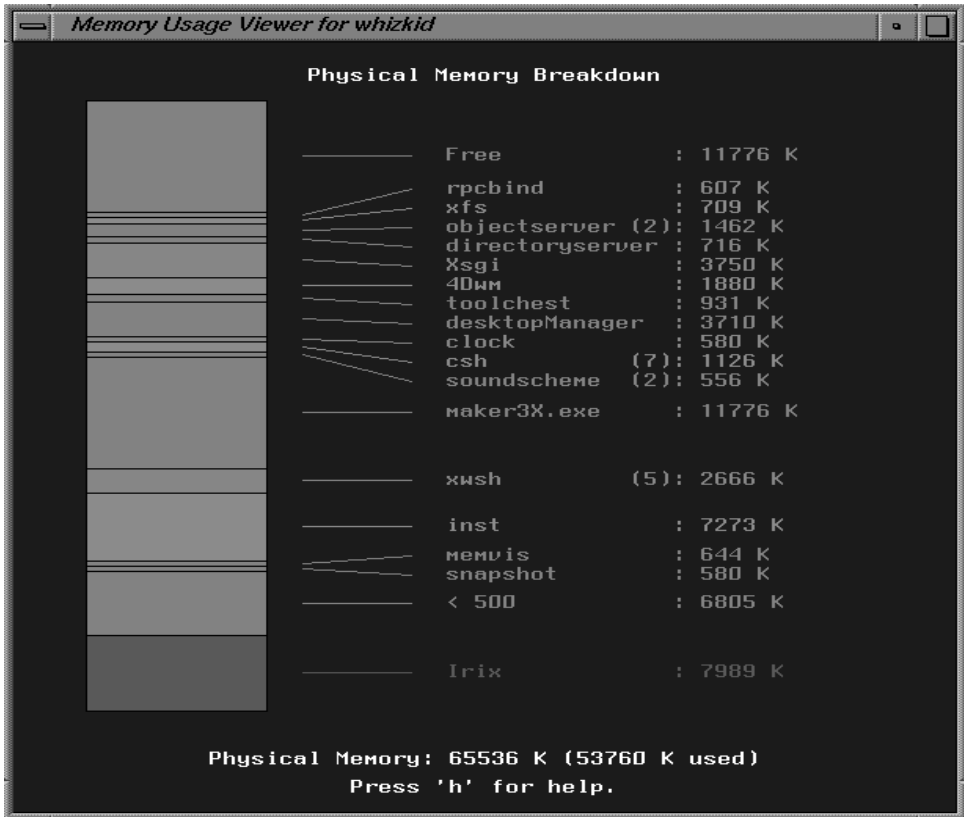


**Figure 2-17** The memvis Window

If more than one copy of a program is running, the number of copies is displayed in parentheses after the program name, and any sharing of physical pages is prorated accordingly.

There is a comprehensive help screen available for *memvis* by placing the mouse cursor in the window and pressing the “h” key. When you do so, you see the screen shown in Figure 2-18.

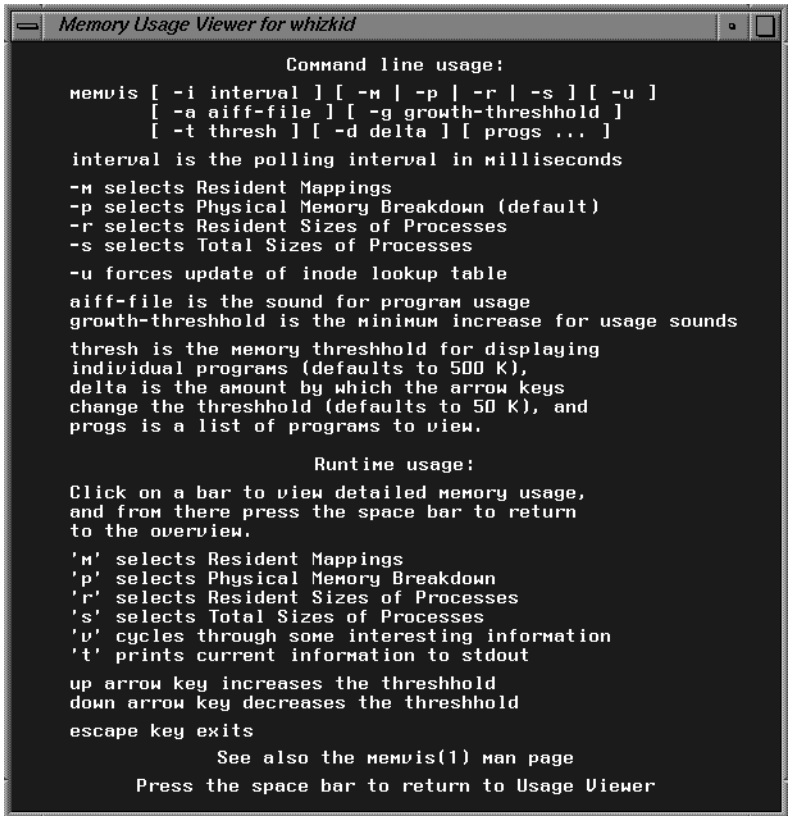


**Figure 2-18** The memvis Help Screen

The section of the window titled *Runtime Usage* provides concise information on the immediate graphical manipulations of the *memvis* window. For additional information on runtime usage, see “memvis Runtime Controls.”

### Basic memvis Viewing Modes

On startup, *memvis* displays a bar chart depicting the breakdown of memory use. Each bar is labeled with the name of the program using the memory and the number of kilobytes of memory used. If more than one copy of a program is running, the number of copies is displayed in parentheses after the program name.

By default, *memvis* only displays programs that are using more than 500 kilobytes of memory; programs using less than this are lumped together and labeled "< 500." This threshold is specifiable on the command line and changeable at run time. Press the down-arrow key to decrease, and the up-arrow key to increase the threshold.

The *memvis* command has four different modes of viewing:

Physical Memory Breakdown
: The default mode shows the amount of physical memory being used by each process, the amount of free memory, and the amount of memory being used by IRIX. The amount of memory charged to each process is calculated by taking the pages each process has in memory and pro-rating the sizes with the number of processes using each page.

Total Sizes of Processes
: This mode shows the total sizes of all the processes in the system. This corresponds to the SZ field of *ps*(1) output.

Resident Sizes of Processes
: This mode shows the resident sizes of all the processes in the system. This corresponds to the RSS field of *ps*(1) output.

Resident Mappings
: This mode shows the resident sizes of all mapped objects in the system. A mapped object can correspond to an executable file, a dynamic shared object, a memory-mapped file, or a region attached to a process by *rld*(1).

Alternately, a list of programs to monitor can be specified on the command line. In this case, a bar for each of the programs specified appears (as long as that program is running) and any threshold is ignored.

### Program Region Breakdown

If you click on a bar or program name in the main chart *memvis* displays a breakdown of the regions within the selected program as shown in Figure 2-19. Each region is labeled with its type and (if possible) the base name of the file or device corresponding to that region. If *memvis* is unable to determine the base name of the file or device for a region that does correspond to a file or device, *memvis* displays the inode number of the file or device.

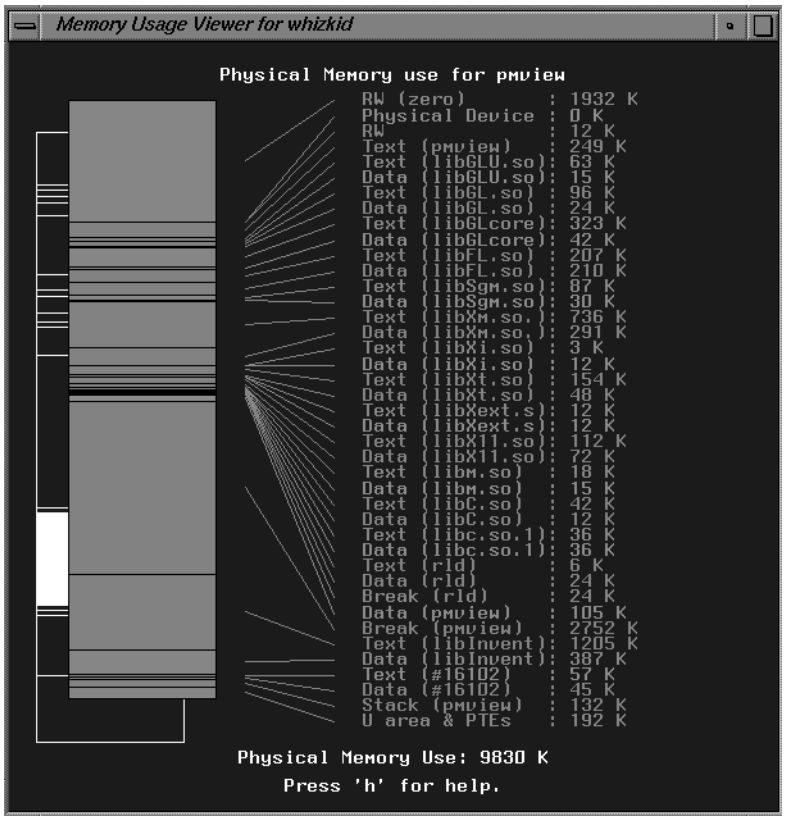


**Figure 2-19** A memvis Program Region Breakdown

The region types are:

| | |
|---|---|
| Text | This region contains executable instructions. These instructions most likely came from an executable program file or a dynamic shared object. |
| Data | This region contains program data. Regions marked Data are usually associated with a particular executable program file or a dynamic shared object. |
| Break | Data region that is grown with *brk*(2). This is the region that contains memory allocated by *malloc*(3C). |
| Stack | Runtime stack. This region is used for procedure call frames, and can grow if the program makes deeply nested procedure calls or calls procedures that allocate large amounts of stack space for temporary variables. |
| Shmem | A System V shared memory region. |
| Physical Device | Region corresponds to a physical device other than main memory, such as a graphics device. |
| RW | Read / Write data without the Copy on Write bit set. This did not come from an executable program file or a dynamic shared object, and could be a memory-mapped file. |
| RO | Read only data. |
| U area & PTEs | The user area and page table entries. The kernel uses this information to administer a process. |

Clicking on the IRIX bar in Physical Memory Breakdown mode causes *memvis* to display a breakdown of the memory that it is charging to the operating system. Separate items include FS Cache, Buffer Data, Heap, Streams, Zone, BSD Networking, Age Frame Data, Kernel Tables, UNIX Data Space, UNIX Code Space, Symmon, and Other, as shown in Figure 2-20.

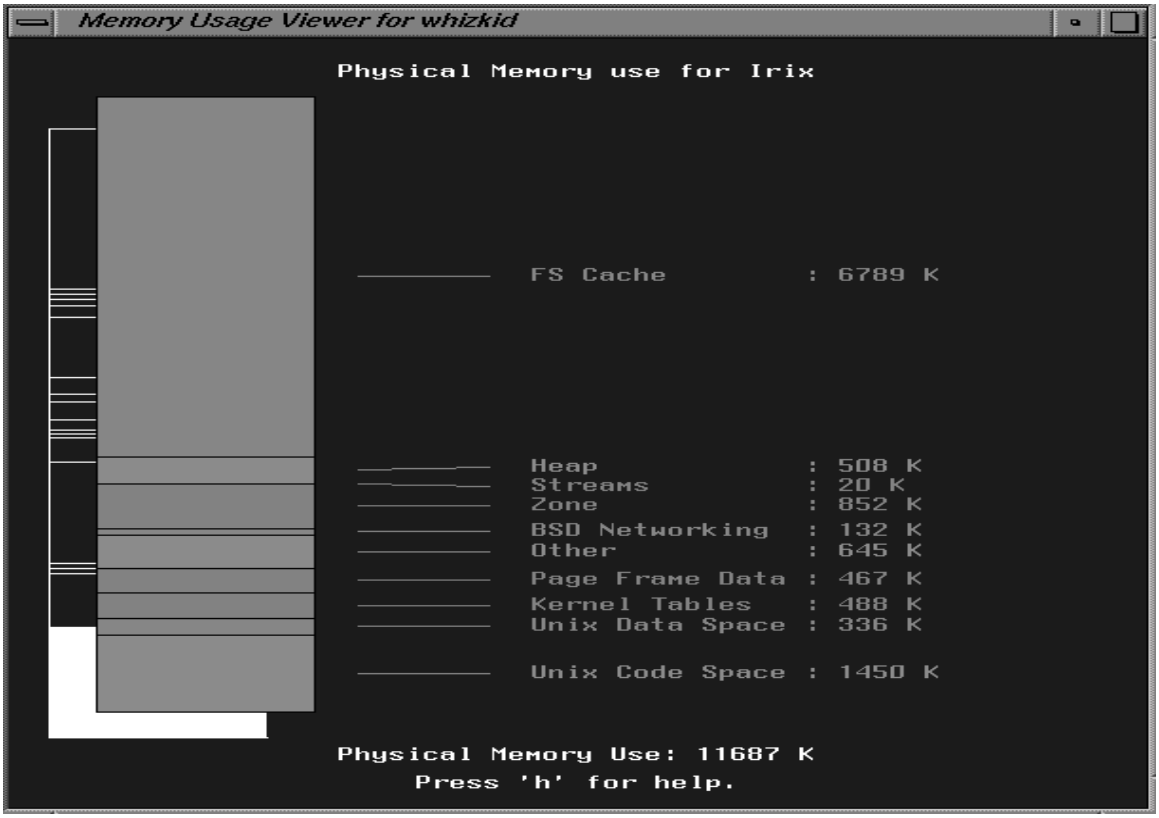


**Figure 2-20** IRIX Physical Memory Use

When viewing the breakdown of program memory usage, clicking and dragging on the shadow bar switches the display to another program.

The first time a particular program is selected within *memvis*, the program reads in information about executables and libraries on the system while displaying a wait message. The *memvis* tool keeps this information in its database file (*$HOME/.memvis.inodes*). If this file does not exist or is older than */unix*, *memvis* creates it, which can take as long as a minute. If the database already exists and is newer than */unix*, reading it takes only a few seconds. See "The memvis Environment" for information on customizing *$HOME/.memvis.inodes*.

### Additional Information About memvis

In addition to the four basic viewing modes and the process region breakdown, *memvis* cycles through displays of additional information when the *v* key is pressed. This additional information is a subdivision of each bar in the chart, with the right portion of each bar corresponding to the additional information. Down the right side of the window, the values corresponding to the right portion of each bar are displayed.

The following additional information is available:

| | |
|---|---|
| Private | The portion of each bar that is private memory; that is, memory that is not being shared. This additional information is available in all modes, except when viewing the IRIX breakdown. |
| Shared | The portion of each bar that is shared between more than one process. This is calculated by subtracting the Private amount from the Physical amount for each bar. Shared is available in all modes, except when viewing the IRIX breakdown. |
| Physical | The portion of each bar that is consuming physical memory. Physical is available in Resident Sizes of Processes and Total Sizes of Processes modes. |
| Resident | The portion of each bar that is resident in memory (without taking sharing into account). Resident is available in Total Sizes of Processes mode. |

### Command Line Options

The *memvis* tool has the following command line options:

**-i** *interval* Update display every interval milliseconds. The default in the absence of the **-i** option is 500.

**-m** Start using "Resident Mappings" mode.

**-p** Start using "Physical Memory Breakdown" mode. This is the default.

**-r** Start using "Resident Sizes of Processes" mode.

**-s** Start using "Total Sizes of Processes" mode.

**-u** Rebuild the inode database *$HOME/.memvis.inodes* even if it is not older than */unix*.

**-t** *thresh* Use *thresh* kilobytes instead of 500 kilobytes as the starting display threshold. Programs using less than this amount of memory in a particular view are not displayed separately, but rather are lumped together in a single bar.

**-d** *delta* Change the display threshold by *delta* kilobytes when the up and down arrow keys are pressed. The default is to change the threshold by 50 kilobytes.

Any command line arguments following the arguments described above are interpreted as names of programs. If program names are specified, *memvis* displays only the memory usage of the programs specified, with all other programs lumped together in a bar labeled Other. In this case, any threshold or delta is ignored. This is useful when one is interested in the behavior of a particular program or set of programs, such as when testing for memory leaks.

### memvis Runtime Controls

Some *memvis* display parameters can be modified at runtime. The following keys are hot:

*p* This key selects "Physical Memory Breakdown" mode

*r* This key selects "Resident Sizes of Processes" mode

*s* This key selects "Total Sizes of Processes" mode

| | |
|---|---|
| *v* | This key cycles through the available additional information for the current mode. (See "Additional Information About memvis.") |
| *up-arrow* | This key increases the display threshold by 50 kilobytes (by default) or if the **-d** option was specified, by *delta* kilobytes. |
| *down arrow* | This key decreases the display threshold by the same amount. When the threshold is decreased to 0, all programs running are displayed, even those that use no memory (such as kernel processes). |
| *t* | This key causes *memvis* to print statistics about the current view to the terminal window. The fields in each line are separated by tab characters to simplify the parsing of the output by other programs (they are also padded with spaces). There are three different types of print outs: All Programs, Resident Mappings, and Program Breakdown. The mode in use when the *t* command is entered is printed. |
| *h* | This key brings up an online help screen, and the space bar returns from there to viewing memory. |
| *escape* | This key quits *memvis*. |

In the main view, clicking on a program's bar causes *memvis* to display a detailed memory usage chart for that program. In the detailed usage view, clicking on the shadow bar switches the program being displayed, and clicking outside the shadow bar or pressing the space bar returns to the main view.

### memvis Examples

The following examples demonstrate typical *memvis* usage:

**`memvis -p -t 1000 -d 100`**
This command brings up *memvis* in Physical Memory Breakdown mode, with programs using 1000 kilobytes or more of memory displayed separately in their own bars. The up and down arrow keys increase and decrease the threshold by 100 kilobytes, respectively.

**`memvis -r xwsh toolchest 4Dwm Xsgi fm`**
This command brings up *memvis* in Resident Sizes of Processes mode. This command directs *memvis* to display bars for *xwsh*(1), *toolchest* (1), *4Dwm* (1), *Xsgi* (1) and *fm*(1). All other programs are combined into a bar labeled *Other*.

### The memvis Environment

The *$HOME/.memvis.inodes* file created by *memvis* is a table of files that are likely to correspond to regions mapped into processes, along with inode numbers. The *memvis* tool builds this table if it doesn't exist, if it is older than */unix*, or if the **-u** option is supplied, and uses it to label the bars when viewing memory breakdown within a process.

The USAGEPATH environment variable can be used to alter the way *$HOME/.memvis.inodes* is built. Set USAGEPATH to a colon-separated list of directories to recursively search when building the inode database. If USAGEPATH is not found in the environment, *memvis* uses the following default path:

```
/usr/ToolTalk:/usr/bin:/usr/lib:/usr/local:/usr/Cadmin:
/usr/CaseVision:/usr/sbin:/usr/bsd:/usr/etc:/lib:/sbin:
/bin:/etc:/usr/gfx
```

The *memvis* tool gets memory information for processes from the */proc* file system.

### memvis Caveats

The totals displayed for the breakdown of a program's regions do not always add up exactly to the amount of memory in the main view. In Physical Memory mode, this discrepancy is due to rounding error. In Total Size mode, this is often due to the inclusion of physical devices in the breakdown.

Beware of *shared object amortization*. When a program that uses a shared object (for example, *libXm.so*) is started, the memory usage of all other programs that use that shared object can decrease. This is because the amount of memory charged to each program for shared object usage is prorated, based on the amount of sharing.

## The opsview ORACLE Parallel Server Visualization Tool

The *opsview* tool demonstrates capabilities of the Performance Metrics Collection System (PMCS) to drive visualization tools. While the application is a prototype written specifically to monitor a pair of Challenge servers running the ORACLE Parallel Server (OPS) software, the utility can provide an interesting display even for a single workstation.

It is necessary to configure *opsview* for the particular local system setup. Enter these commands:

```
cd /usr/demos/PerfCoPilot/opsview
cat README
./Configure
```

Once this has been done, the command

```
opsview
```

produces a window similar to that shown in Figure 2-21.

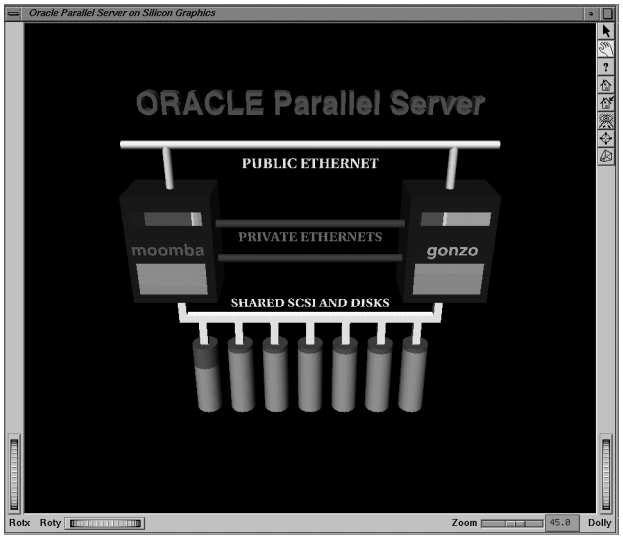


**Figure 2-21** The opsview Window

On startup, *opsview* displays a three-dimensional model of two Challenge servers sharing a common SCSI bus with attached disks, two private Ethernet connections, and a public Ethernet connection. One of the hosts is designated as the ‘‘red’’ host (the front panel hostname is ‘‘red’’), the other is designated ‘‘green’’ (the front panel hostname is ‘‘green’’). These two primary colors are used to display data about the corresponding hosts as detailed below.

The *opsview* application is based on Inventor, like *pmview*, so many of the generic scene manipulation operations described for *pmview* also apply to *opsview*.

Briefly, the scene is displayed within an Inventor examiner viewer. This viewer allows the user to interactively change the view of the object by click-dragging the left and middle mouse buttons in the 3D window or using the various controls along the window border (for example, zoom). This application also allows the user to pick various objects within the scene for a more detailed view of performance within a subsystem.

Pick mode is entered by selecting the arrow button (the topmost icon in the right window border). Pick mode is indicated by an arrow-shaped cursor, and performed by clicking the left mouse button while the cursor is on top of the scene object to be selected.

The following elements of the scene are driven by live performance data fetched from the two hosts:

CPU
: The multi-colored horizontal bar at top front and back (Just above the host name) faces of both hosts is a thumbnail display of total CPU utilization. The thumbnail display follows the *gr_osview*(1) format. If you select this region the PCP tool *mpvis* is launched to provide a more detailed per-CPU view.

Memory
: A two-colored rectangular region at the bottom front and back faces of both hosts is a thumbnail display of physical memory use. The display follows the *memvis*(1) format. If you select this region, the PCP tool *memvis* is launched, allowing a very detailed examination of memory use.

Public-Ethernet
: The brightness and color of the Ethernet cable part of the display is determined by the number of packets processed by the two hosts. Green or red color is modulated by packets on the green or red host, respectively. If you select the Ethernet cable part of the display or its name label, two instances of *gr_osview* (one instance for the green host and one for the red host) are launched to display per-protocol and per-interface network traffic on the two hosts. The default startup files for *gr_osview* are found in */usr/lib/pcp/config/gr_osview-a.rc* and */usr/lib/pcp/config/gr_osview-b.rc*. Alternate startup files for each host may be specified with the **-g** option.

Disks
: Total disk I/O from the red and green machine are displayed as red and green cylinders, respectively. The number of I/O operations performed determines the height of these colored cylinders.

ORACLE
: If you are running OPS software, you can select the ORACLE Parallel Server banner text to bring up two instances of the PCP tool *pmchart*. This tool allows you to display interesting OPS performance data from the two machines. The default startup file for *pmchart* is found in */usr/lib/pcp/config/pmchart.rc*. An alternate startup file may be specified with the **-p** option.

Complete information on *opsview* is provided in the *opsview* reference page, and additional information on ORACLE database servers is provided in the section of this guide titled “ORACLE Metrics and the ORACLE PMDA” on page 174.

## Archive Logging With PCP

You can direct any of a number of the PCP tools and commands to “replay” the values for performance metrics from an archive log. The GUI tools provide the VCR controls described in “Common Conventions and Arguments” on page 42. You can control the position, replay rate, and direction within an archive. The non-GUI tools support a variety of

command-line options to affect the starting position, replay rate, and direction of replay.

In all cases, the archive logs must be created with the *pmlogger*(1) utility.

Each archive log consists of a number of physical files that support the descriptions of the metadata and a temporal index, in addition to the values of selected performance metrics. The collection of files that constitute an archive log forms an autonomous unit of information that fully describes the performance data and how to interpret it. Consequently, an archive log may be shipped from one system to another, and can always be replayed without access to any other information. This is most useful for remote diagnosis or retrospective analysis comparing current performance with historical performance (collected at a time when the system configuration may have been very different).

## The pmlogger Command

The *pmlogger* command is used to create archive logs of selected metric values. The usage and syntax of this command are documented in the *pmlogger* reference page.

To support the required flexibility and control over what is logged and when, *pmlogger* maintains an independent, two-level state for each instance of each performance metric. At the first (mandatory) level, logging is allowed to be “on” (with an associated interval between samples), “off,” or “maybe.” In the latter case, the second (advisory) level logging is allowed to be “on” (with an associated interval between samples), or “off.” The mandatory level allows universal specification that some metrics must be logged or must not be logged. The default state for all metrics when *pmlogger* starts is mandatory "maybe" and advisory "off."

The utility *pmlc* described below provides the interface that allows a user to interrogate and change the logging state once *pmlogger* is running.

### The Primary Instance of pmlogger

Optionally, each system running a *pmcd*(1) may also be configured to run a “primary” logger instance. Like *pmcd*, this *pmlogger* instance is launched by */etc/init.d/pcp*, and is configured by the following files.

*/etc/config/pmlogger*
: Use *chkconfig* to activate or disable the primary *pmlogger* instance.

*/etc/config/pmlogger.options*
: Command line options passed to the primary pmlogger.

*/usr/lib/pcp/config/pmlogger.config*
: Default initial configuration file for the primary *pmlogger*, usually named with the **-c** option in */etc/config/pmlogger.options*.

The syntax for the *pmlogger* utility is documented in the *pmlogger* reference page. However, the following is a simple example configuration file:

```
log mandatory on once { hinv.ncpu hinv.ndisk }

log mandatory on 1 hour
    irix.kernel.all.load [ “1 minute” ]

[access]
disallow * : all except enquire;
allow localhost : mandatory, advisory;
```

The configuration requests *pmlogger* to record the number of disks initially in the system, and log the one-minute load average once an hour thereafter.

### Other Instances of pmlogger

In addition to any primary *pmlogger*, a user may choose to create an archive log with arbitrary choices of performance metrics, instances, and mixtures of logging frequencies. This would be appropriate to capture detailed information (both the range of metrics logged and the logging frequency) about a transient performance problem when it occurs, for capacity planning or for auditing system performance.

The *pmlogger* instances may execute locally on the system being monitored, in which case the disk writes and disk storage must be accommodated on the monitored system. Or they may also run on the local workstation, but connect to *pmcd* on the system being monitored. (This trades off network traffic at the monitoring system for disk writes and disk storage at the local system.)

### Access Control for pmlogger

When *pmlogger* starts up, it reads a configuration file. Within this configuration file is an optional access control section, which allows the user who launches *pmlogger* to specify which hosts can send requests to *pmlogger*. These requests control the *pmlogger* state that determines which metrics are logged and how often.

## The pmdumplog Tool

Once an archive log has been created with *pmlogger*, the most common usage would be to replay the log with one of the utilities such as *pmchart*, *pmkstat*, *pmval*, or one of the visualization tools.

However, sometimes it is necessary to characterize the contents of an archive log, independent of the tool to be used to replay the log. This is the function of *pmdumplog*.

Given an existing archive *babylon.percpu*, the command:

```
pmdumplog -l babylon.percpu
```

displays the "label record" that identifies the origin of the archive as follows:

```
Log Label (Log Format Version 1)
Performance metrics from host babylon, commencing Wed Jan 11
17:50:50.990 1995
```

**Note:** Several sample archives are provided as part of the Performance Co-Pilot Tutorial in the directory */usr/demos/PerfCoPilot/Tutorial/Archives*.

To see which metrics have been logged at some time during the life of the archive *tokyo.perdisk*, use the command:

```
pmdumplog -d tokyo.perdisk
```

This command produces:

```
Descriptions for Metrics in the Log ...
PMID: 1.18.3 (irix.kernel.all.load)
   Data Type: float  InDom: 1.5 0x400005
   Semantics: instant  Units: none
PMID: 1.80.1 (irix.disk.dev.read)
```

```
   Data Type: 32-bit unsigned int  InDom: 1.2 0x400002
   Semantics: counter  Units: count
PMID: 1.80.2 (irix.disk.dev.write)
   Data Type: 32-bit unsigned int  InDom: 1.2 0x400002
   Semantics: counter  Units: count
PMID: 1.80.7 (irix.disk.dev.total)
   Data Type: 32-bit unsigned int  InDom: 1.2 0x400002
   Semantics: counter  Units: count
```

Other command-line options request reporting of instance domain information and the dumping of raw values for selected performance metrics.

The *pmdumplog* utility is documented in the *pmdumplog* reference page.

## The pmlc Tool

The *pmlc* command is an interactive utility designed to support interrogation of, and control over, the internal logging state maintained by a *pmlogger* instance.

The particular *pmlogger* instance is identified by a combination of *host* (the host where *pmlogger* is running, which defaults to the local host) and either a process ID, or the special “primary” logger designation. For example:

```
pmlc
pmlc> connect primary
....
pmlc> connect 12345 @far.away.host.com
....
```

The status command supports interrogation of the current logging status for one or more metrics.

The log command causes requests to modify the logging status for selected metrics to be forwarded to a *pmlogger* instance.

The *pmlc* utility is documented in the *pmlc* reference page.

## The Performance Metrics Inference Engine (pmie)

The Performance Metrics Inference Engine (*pmie*) is a tool for defining certain conditions on monitored systems and specifying actions to take when those conditions are met. The *pmie* tool accepts a collection of arithmetic, logical, and rule expressions, which it evaluates at specified times on specified host systems. Using *pmie*, you can access and interpret the large volume of performance data made available by the Performance Metrics Collection Subsystem (PMCS) and delivered through the Performance Metrics Application Programming Interface (PMAPI).

The *pmie* utility was designed to meet the following requirements:

- Expressive Power: Our first requirement is that the user can conveniently express the conditions and associated actions needed in performance analysis practice.
- Ease of Use: The correctness of a set of rules can be difficult to verify. The language must have clear, simple semantics. Rule debugging support is essential.
- Robustness: The system must do the right thing with missing or unexpected performance data.
- Source of Data: The evaluation of the expressions takes place either in real time from multiple hosts or against archives of stored performance data.
- Performance: Our final requirement is to achieve the above at a very small computational cost and with minimum disturbance (probe effect) of the system being monitored. The PMCS provides for transport of metrics from a host or hosts being monitored to a separate workstation performing the analysis. This architecture is right for achieving the small probe effect.

### Introduction to pmie

This section presents and explains some basic examples of *pmie* usage. The *pmie* tool is invoked according to the following syntax:

```
pmie [-a archives] [-h host] [-i] [-n namespace]
     [-T runtime] [-v] [-Z timezone] [-z] [filename]
```

One of the most basic invocations of this tool is the form

**pmie** *filename* **&**

In this form, the expressions to be evaluated are read from *filename*. In the absence of a given *filename*, expressions are read from standard input, usually your system keyboard.

Given the **-i** flag, *pmie* executes in interactive mode, and the user is presented with a menu of options like this:

```
Performance Co-Pilot Inference Engine (pmie) V0.0
   f [file-name]       - load rules
   l [rule-name]       - list rule(s)
   r [period]          - run evaluator
   s                   - single step evaluation
   q                   - quit

pmie?
```

The interactive mode is useful mainly for debugging new rules.

If both the **-i** flag and a *filename* are present, the expressions in the given file are loaded before entering interactive mode.

### pmie and the Performance Metrics Collection System

Before you use *pmie*, you need to familiarize yourself with some Performance Metrics Collection System (PMCS) basics. This section provides just enough information to get you started.

The PMCS makes available hundreds of performance metrics, that you can use when formulating expressions for *pmie* to evaluate. If you want to find out which metrics are currently available on your system, use the command:

```
pminfo
```

The *pminfo* command is documented in the *pminfo*(1) reference page and in "The pminfo Command."

Use the *pminfo* command to find out more about a particular metric. The next example is a command to fetch (specified with the **-f** flag) a value for the metric *irix.disk.dev.total* from the host (specified with the **-h** flag) named *moomba*. The command

```
pminfo -f -h moomba irix.disk.dev.total
```

produces the following response:

```
irix.disk.dev.total
    inst [131329 or "dks1d1"] value 970853
    inst [131330 or "dks1d2"] value 53581
    inst [131331 or "dks1d3"] value 5353
    inst [131332 or "dks1d4"] value 225
    inst [131333 or "dks1d5"] value 9674
    inst [131334 or "dks1d6"] value 14383
    inst [131335 or "dks1d7"] value 5578
```

This reveals that the metric has seven instances, one for each disk on the system. The instance names are "dks1d1," "dks1d2," and so on up to "dks1d7."

Use the following command to request help text (specified with the **-T** flag) to read an explanation of the values:

```
pminfo -T irix.disk.dev.total
```

You see a response similar to this:

```
irix.disk.dev.total
```

```
Help:
The cumulative number of transfers to or from
a disk device since boot.
```

The PMCS provides a cumulative counter of disk transfer operations. You can confirm this by examining the descriptor (specified with the **-D** flag) for the metric, as shown in this command:

```
pminfo -D -h moomba irix.disk.dev.total
```

In response, you see output similar to this:

```
irix.disk.dev.total
    Data Type: 32-bit unsigned int InDom: 1.2 0x400002
    Semantics: counter Units: count
```

Because a cumulative counter such as this is not much use in its raw form, the inference engine automatically converts all such values to *rates*. That is, instead of the counter values reported by the *pminfo* command, *pmie* reports the number of disk transfers over a known interval of time. The rate value is measured in events per second (*count* / *sec*).

### A Simple pmie Example

The following example directs the inference engine to evaluate and print values (specified with the **-v** flag) for the *irix.disk.dev.total* metric:

```
pmie -h moomba -v
irix.disk.dev.total;
^D
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| expr_1: | ? | ? | ? | ? | ? | ? | ? |
| expr_1: | 8.7 | 0 | 0 | 0 | 0 | 50.51 | 0 |
| expr_1: | 4.5 | 3 | 0.1 | 0.1 | 0 | 47 | 0 |
| expr_1: | 5.495 | 2.198 | 0.0999 | 0.1998 | 0 | 44.96 | 0.8991 |
| expr_1: | 4.1 | 1.9 | 0 | 0 | 0 | 180.8 | 16.5 |
| expr_1: | 4.2 | 1.9 | 0 | 0 | 0 | 207.1 | 6.7 |
| expr_1: | 4.8 | 1.9 | 0.3 | 0 | 0 | 228.1 | 3.6 |
| expr_1: | 4.493 | 2.097 | 0 | 0 | 0 | 224.4 | 3.095 |

In the above example output, the values come live from the host (specified with the **-h** flag) *moomba*. Notice that the seven values for the first sample are unknown (represented by the question marks (?) in the first row of output). This is because rates can be computed only when at least two samples are available. The subsequent samples are produced every ten seconds by default. The second sample reports that during the preceding ten seconds there were 8.7 transfers per second on one disk and about 50.5 on another disk, with the remaining disks idle.

Rates are computed using time-stamps delivered by the PMCS. Due to unavoidable inaccuracy in the actual sampling time (the sample interval is not exactly 10 seconds), you often see more decimal places in values than you expect. Notice, however, that these errors do not accumulate but cancel each other out as subsequent samples come in.

In the above example, the expression to be evaluated was entered on standard input (the keyboard), followed by the end-of-file character **<Ctrl-D>**. Usually it is more convenient to enter expressions into a file (for example, *myrules*) and ask *pmie* to read the file. Use the command syntax:

```
pmie -v -h moomba myrules
```

Please refer to the *pmie*(1) reference page for more command line format and options information.

### More Complex Examples

This section illustrates more complex *pmie* expressions. The following arithmetic expression

```
(irix.disk.dev.write / irix.disk.dev.total) * 100;
```

computes the percentage of write operations over the total number of disk transfers. This expression also produces a value for each disk device.

The following logical expression

```
irix.disk.dev.total > 10 &&
irix.disk.dev.write > irix.disk.dev.read;
```

tells you for each disk, whether the number of writes exceeds the number of reads, given that there is some reasonably significant disk activity (more than 10 transfers / second).

Finally, the following rule expression

```
some_inst irix.disk.dev.total > 100 ->
print “high disk i/o”;
```

prints a message to the standard output (your screen or a designated file) whenever the total number of transfers for some disk (instance) exceeds 100 transfers per second.

Using *pmie* to evaluate the above expressions, you see output similar to the following:

```
pmie -h moomba -v
irix.disk.dev.total;
(irix.disk.dev.write / irix.disk.dev.total) * 100;
irix.disk.dev.total > 10 &&
    irix.disk.dev.write > irix.disk.dev.read;
some_inst irix.disk.dev.total > 100 ->
    print “high disk i/o”;
^D
expr_1:      ?      ?      ?      ?      ?      ?      ?
expr_2:      ?      ?      ?      ?      ?      ?      ?
expr_3:      ?      ?      ?      ?      ?      ?      ?
expr_4:      ?

expr_1:   22.4   30.9    0.4    1.3      0   15.9      0
expr_2:  18.75  6.149      0      0      ?      0      ?
expr_3:  false  false  false  false   false  false  false
expr_4:  false

expr_1:    9.1   12.3    0.1    0.1      0     50      0
expr_2:  94.51    100    100    100      ?      2      ?
expr_3:  false   true  false  false  false   false  false
expr_4:  false

expr_1:  4.597  10.99      0 0.0999      0  42.17      0
expr_2:  86.96     95      ?      0      ?  21.09      ?
expr_3:  false   true  false  false   false  false  false
expr_4:  false

expr_1:  4.599  7.999 0.9997 0.0999      0  40.59      0
expr_2:  97.83    100    100    100      ?  29.06      ?
expr_3:  false  false  false  false   false  false  false
expr_4:  false
```

```
pmie NOTE (Wed Jan  4 09:48:45 1995): high disk i/o
expr_1:    4.3      2      0      0      0    165   15.9
expr_2:  93.02     95      ?      ?      ?  6.545      0
expr_3:  false  false  false  false   false  false  false
expr_4:   true

pmie NOTE (Wed Jan  4 09:48:55 1995): high disk i/o
expr_1:    4.8    1.9      0      0      0  198.3    7.7
expr_2:  89.58    100      ?      ?      ?  3.026      0
expr_3:  false  false  false  false   false  false  false
expr_4:   true

pmie NOTE (Wed Jan  4 09:49:05 1995): high disk i/o
expr_1:    5.9    1.9      0      0      0  212.1    3.3
expr_2:  79.66    100      ?      ?      ?  3.112      0
expr_3:  false  false  false   false  false  false  false
expr_4:   true

^C
```

The *pmie* command returned eight samples before the user pressed the **<Ctrl-C>** key to interrupt the operation. The first sample contains unknowns, since all four expressions depend on computing rates. Also notice that the second expression (*expr_2*) has an undefined value whenever a disk is idle; the denominator of the evaluated expression is 0. From the sixth sample on, the rule condition for the rule expression (*expr_4*) is satisfied and a message is printed.

## pmie Essentials

This section describes the complete *pmie* syntax, as well as macro facilities and the issue of sampling and evaluation frequency.

Complex expressions are built up recursively from simple elements:

1. Performance metric values are obtained from running hosts or archive files.
2. These raw values are refined into computed values using arithmetic operators.
3. These computed values are compared using relational operators.

4. The resulting true/false evaluations are aggregated using Boolean operators, including very powerful quantifiers.
5. The resulting true/false evaluation can initiate a sequence of actions.

### Basic pmie Syntax

The inference engine accepts a sequence of semicolon-terminated expressions (*exprs*):

*exprs* = *expr* **;** [*exprs*]
: Sequence of expressions, each terminated by a semicolon. Any type of expression can appear at the top level:

*expr* = *metric*
: Metric expressions direct *pmie* to fetch the named performance metric value(s).

*expr* = *aexpr*
: Arithmetic expressions map numeric values to other numeric values.

*expr* = *rexpr*
: Relational expressions map numeric values to true/false values.

*expr* = *lexpr*
: Logical expressions map true/false values to other true/false values.

*expr* = *act*
: Action expressions have true/false values, but are used primarily to take the named action, rather than to generate a value.

*expr* = *rule*
: Rule expressions specify the conditional execution of actions.

### pmie Macros

If fully written out, expressions in *pmie* tend to be verbose and repetitious. The use of macros (a shorthand for a piece of syntax) can eliminate repetition and improve readability and modularity. A macro definition is considered a kind of naming action:

*act* = *ident* **=** *string*
: Associates the name *ident* with the given *string* constant.

Any subsequent occurrence of

**$***ident*

is replaced by the *string* most recently associated with *ident*. For example, given the macro definitions

```
pmie -v
servers = ":moomba :larry";
clients = ":splat :wobbly :sandpit";
all = "$servers $clients";
```

you can then use the syntax

```
sum_host irix.kernel.all.cpu.idle $all;
```

to compute the total CPU idle time summed over all five server and client machines specified in the macros.

### Setting Evaluation Frequency

A naming action is also used to set the frequency of expression evaluation.

*act* = *ident* **=** *num* [*units*]

The name *delta* is reserved to denote the interval of time separating two evaluations. You set *delta* as follows:

**delta =** *num* [*units*];

If present, *units* must be one of *sec*, *min* or *hour*. If absent, *units* are assumed to be seconds (*sec*).units. For example

```
delta = 5 min;
```

has the effect that any subsequent expressions (until *delta* is changed again) are scheduled for evaluation at a fixed frequency, once every five minutes.

If not otherwise specified, delta is set to be 10 seconds by default

### pmie Units Syntax

You are encouraged to specify the units for numeric constants. The inference engine converts all numeric values to canonical units (*seconds* for time, *bytes* for space, and *events* for count). If units are specified they are checked against metadata in the performance metrics descriptors in the context of the given expression. The syntax for a *units* specification is:

*units* = *unit*

*units* = *unit*s *unit*

*units* = *units* / *unit*

*unit* = **byte** | **Kbyte** | **Mbyte** | **Gbyte** | **Tbyte**

*unit* = **nsec** | **usec** | **msec** | **sec** | **min** | **hour**

*unit* = **count** | **Kcount** | **Mcount** | **Gcount** | **Tcount**

If you do not specify the units for numeric constants, it is assumed that the constant is in the canonical units of *seconds* for time, *bytes* for space, and *events* for count. In this case it is also assumed that the dimensionality of the constant is correct.

Identifiers (*ident*) and strings (*string*) have the syntax:

*ident* = _ | **.** | **A..Z** | **a..z** (_ | **.** | **A..Z** | **a..z** | **0..9**)*

*ident* = ‘( ? | \?)*’

*string* = “( ? | \?)*”

This is, the usual syntax for identifiers, except that you may use the dot character (.), as used in metric names, and you can make any arbitrary sequence of characters into an identifier by enclosing the sequence in single quotes. A string is an arbitrary sequence of characters enclosed in double quotes. In quoted identifiers and strings, the backslash character (\) escapes any special meaning that the following character may have.

### pmie Comments Syntax

Comments may be embedded anywhere in the source, in the form:

*comment* = **/*** *text* ***/**
C - style comment, with no nesting of comments.

*comment* = **//** *text*
C++ - style comment.

## pmie Metric Expressions

A Performance Metrics Name Space (PMNS) provides a means of referring to a set of performance values sharing common semantics. For instance, the

name *irix.disk.dev.read* refers to a free running counter of disk read operations. Three further parameters: (*host*, *instance* and *sample*) are required to identify a single value for a metric. The expression

```
irix.disk.dev.read :moomba #dks0d1 @0
```

refers to the current value (*sample time: 0*) of the read counter for the disk named *dks0d1* (*instance: dks0d1*) on the machine named *moomba* (*host: moomba*). The domain of host names is provided by the internet naming convention. The instance domains are provided by the Performance Metrics Domain Agents (PMDAs). The sample time domain is defined as the set of natural numbers 0, 1, 2, and so on. A number refers to one of a sequence of sampling events, stretching back from the current sample 0 to its predecessor 1, whose predecessor was 2, and so on. This scheme is illustrated by the time-line figure:

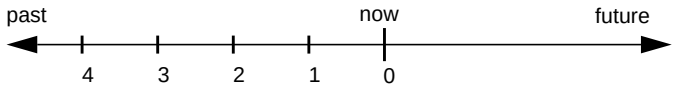


**Figure 2-22** A Sampling Time Line

Each sample point is assumed to be separated from its predecessor by a constant amount of real time, the *delta*. The current sample point is always zero. The value of *delta* may vary from one expression to the next, as you have specified. For more information on deltas, see “Setting Evaluation Frequency.”

We can now visualize the three-dimensional parameter space <*host*, *instance*, *sample*>, as shown in Figure 2-23.

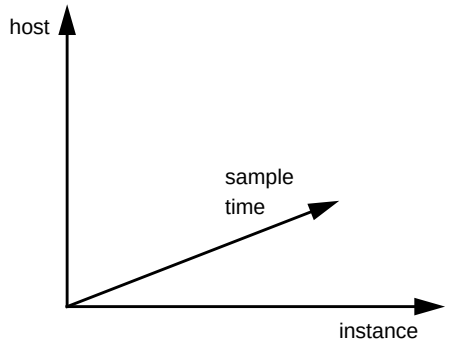


**Figure 2-23** A Three-Dimensional Parameter Space

A single performance metric value is located by its coordinates in this space. A one-, two-, or three-dimensional slice of this space can be referred to when we provide notation for sets of names from the three domains. For example, the syntax

```
irix.disk.dev.read :moomba :gonzo @9..0
```

refers to a 3D aggregate of the values of the last 10 samples of the read counters on all the disks on the two hosts *(moomba* and *gonzo*).

Notice that if no instances are specified explicitly, it is assumed that all available instances are desired.

More formally, the syntax allows optional hosts, metric instances, and sample times specifications to follow a metric name. Thus you may use the syntax:

*metric* = *metric-name* [*hosts*] [*insts*] [*samples*]

The *hosts* specification is a sequence of internet host names, each preceded by a colon.

*hosts* = :*host-name* [*hosts*]

The *instances* specification is a sequence of instance names, each name preceded by a hash. Recall that you can discover the instance names for a particular metric, using the *pminfo* command, see “pmie and the Performance Metrics Collection System” on page 96..

*insts* = #*inst-name* [*insts*]

Either a single point or an interval of samples may be specified. Only the natural numbers 0, 1, 2, 3, and so on are valid here. The end points of the interval may appear in either order.

*samples* = @*number*

*samples* = @*number*..*number*

Where explicit *host*, *instance*, and *sample* times parameters are not given, default values are assumed. The default for host is *localhost*, or more precisely the official hostname for the *localhost* on the current system. When no instance is explicitly mentioned, all available instances are implied. The current value of the metric is commonly required, so this is a good default for the sample time parameter.

## pmie Arithmetic Expressions

Many of the metrics delivered by the PMCS are cumulative counters. Consider for example, the following metric:

```
irix.disk.dev.read
```

A single value for this metric tells you only that a certain number of events have occurred since boot time, and that information may be invalid if the counter has exceeded its 32-bit range and started over. You need at least two values, sampled at known times, to compute the rate at which the I/O operations are being executed. The syntax is:

```
(irix.disk.dev.read @0 - irix.disk.dev.read @1) / delta
```

However, as well as being too verbose, the accuracy of *delta* as a measure of actual inter-sample delay is an issue here. The PMCS delivers time stamps that can solve this problem. For these reasons a built-in, accurate, and implicit rate operator is provided. That is, the expression

```
irix.disk.dev.read
```

already refers to a rate. You may also call the built-in *rate* operator explicitly, as follows:

```
rate irix.disk.dev.read
```

But, this statement returns the rate of rate (acceleration) of disk read operations.

You should recall that our basic expressions do not deliver just single values, but also one-, two-, and three-dimensional representations. We provide aggregation operators to collapse values on a chosen dimension. For example, the statement

```
avg_inst irix.disk.dev.read $Servers @0..4
```

yields a two-dimensional matrix of values (five samples per host), where each value is the average read rate over all disk instances. We can further reduce this to a single value by summing over the remaining *hosts* dimension and then taking the maximum sample:

```
max_sample sum_host avg_inst irix.disk.dev.read $Servers
@0..4
```

Numeric constants are provided for *pmie* as follows:

*aexpr* = *num* [*units*]
: Numeric constants are optionally followed by a *units* specification. Any value that is legal for a C language *double* is legal for a numeric constant expression.

The remaining syntax for arithmetic expressions (*aexpr*) is as follows:

*aexpr* = *metric*
: A metric specification, as detailed in “pmie Metric Expressions” on page 103.

*aexpr* = **(***aexpr***)**
: Parentheses can be used for grouping. The default operator precedence is given by the order of these productions, the operators that bind more tightly being presented first. All operators associate to the left.

*aexpr* = **-***aexpr* | **rate** *aexpr*
: Two unary arithmetic operators, negation and rate.

*aexpr* = *aexpr* ***** *aexpr* | *aexpr* **/** *aexpr*

*aexpr* = *aexpr* **+** *aexpr* | *aexpr* **-** *aexpr*
The usual binary operators, with the usual precedence.

*aexpr* = **sum**_*dom aexpr* | **avg**_*dom aexpr* | **max**_*dom aexpr* | **min**_*dom aexpr*
Aggregation operators, being *sum*, *average*, *maximum* and *minimum*.

The aggregation (and later quantification) operators require the domain (*dom*) of aggregation to be specified. The three domains are denoted by the constants:

*dom* = **host** | **inst** | **sample**
The *host*, *metric instance*, and *sample time* domains.

## pmie Relational Expressions

True / false values are derived from the numerical values previously described. Using relational operators with *pmie* and the numerical values, you can arrive at a boolean result. For instance, you may wish to write

```
irix.disk.dev.read > 50 count/sec
```

to obtain true / false values, indicating which disks are performing more than 50 read operations per second.

All operators are required to take as arguments constant, singleton, and matrix valued expressions. So the **>** operator in this case

```
irix.disk.dev.read :moomba :larry @0..10 > 500 count/sec
```

compares each value in the 3D matrix on its left with the single constant value on its right, producing a 3D matrix of true / false values as the result.

The usual operators are available for relational expressions (*rexpr*). The operators are of equal precedence.

```
rexpr = aexpr == aexp | aexpr != aexpr
rexpr = aexpr > aexpr | aexpr < aexpr
rexpr = aexpr >= aexpr | aexpr <= aexpr
```

## pmie Logical Expressions

Relational operators can deliver one-, two-, or three-dimensional matrices of true / false values. Such a structure of true / false values is collapsed into a smaller dimensional matrix by quantification operators. For example:

```
all_host some_inst irix.disk.dev.read $Servers > 100 count/
sec
```

The result here is a single true / false value indicating whether all the server hosts have at least one disk that is a busy reader.

The syntax for logical expressions (*lexpr*) provides for true / false valued constants and relational expressions:

- *lexpr* = **true** | **false**
- *lexpr* = *rexpr*

Three quantifiers over the three domains (already defined above) are provided:

*lexpr* = **all**_*dom lexpr*
: True only if true for all elements of the named domain.

*lexpr* = **some**_*dom lexpr*
: True only if true for at least one element of the nominated domain.

*lexpr* = *number* **%**_*dom lexpr*
: True only if true for at least the specified percentage of elements of the domain.

The standard boolean operators are supported:

*lexpr* = **!** *lexpr*

*expr* = *lexpr* **&&** *lexpr*

*expr* = *lexpr* **||** *lexpr*

Negation (specified by the prefix ! operator) inverts truth values, while logical AND and logical OR operators function normally.

In addition two special operators are provided:

*lexpr* = **rising** *lexpr*
True when argument value changes from false to true.

*lexpr* = **falling** *lexpr*
True when argument value changes from true to false.

The above logical operators take constant, singleton, and aggregate true / false values as arguments and deliver true / false values as results. Therefore,

```
all_host (
    some_inst irix.disk.dev.read $Servers > 100 &&
    some_inst irix.disk.dev.write $Servers > 100
)
```

tells you whether all server hosts have both a busy reader and a busy writer.

## pmie Action Expressions

Actions are logical (true / false) expressions. An action expression evaluates as *true* if the action is successfully performed, *false* if it is not. But actions are executed for their effect, rather than just their true / false value. The actions supported by the *pmie* inference engine are:

*act* = **shell** [*time*] *string*
The given string is passed to the shell as input. This action is implemented using the *system*(3S) call. The actions fail if the call to *system*(3S) fails.

*act* = **alarm** [*time*] *string*
A notifier containing a time stamp, the given message *string* and a *Cancel* button is posted on the current display screen. Each alarm action first checks if its notifier is already active. If there is an identical active notifier, a duplicate notifier is not posted.

*act* = **syslog** [*time*] [*facility.level*] *string*
The given message *string* is written into the system log.

*act* = **print** [*time*] *string*
A notice containing a time stamp in *ctime*(3C) format and the given *string* is printed out to standard output (stdout).

The above actions take an optional *time* parameter, specifying that once an action is executed, it does not execute again until the given interval of time has passed. The optional *facility.level* parameter for *syslog* is described in the reference page for *logger*(1).

Now we can modify the example rule like this:

```
some_inst irix.disk.dev.total > 100 ->
print 10 min “high disk i/o”;
```

This prevents it from repeating a message until 10 minutes have passed.

Actions may be composed to form more complex actions as follows:

*acts* = *act*

*acts* = *acts* **&** *acts*
: Actions are executed sequentially left to right.

*acts* = *acts* **|** *acts*
: The actions on the left side of the alternate operator (**|**) are executed; if they fail, the actions on the right side are executed.

Note that the **&&** and **||** operators denote logical AND and OR, whereas the **&** and **|** operators denote sequential and alternate execution of actions.

## pmie Rules

Rules have the following syntax:

*rule* = *lexpr* **->** *acts*

The semantics are:

- If the rule condition (logical expression *lexpr*) evaluates true, then perform the action(s) (*acts*) that follow, otherwise do not perform the actions.
- It is required that the rule condition have a singleton truth value; that is, aggregation and quantification operators have been applied to collapse any non-singular domains.

As a further example of a rule, consider:

```
delta = 1 hour;

rising (irix.resource.procovf > 0) ->
syslog “process table overflow, systune suggested”;
```

The sample interval (*delta*) is set to one hour here, to avoid excessive load on the system. A message is written to the system log when the process table overflow counter exceeds 0.

## Caveats and Notes on pmie

### Performance Metric Wrap-Around

Performance metrics that are cumulative counters may occasionally overflow their range and wrap around to 0. When this happens, an unknown value (printed as ?) is returned as the value of the metric for one sample (recall that the value returned is normally a rate).

### pmie Sample Intervals

The sample interval (*delta*) should always be long enough, particularly in the case of rates, to ensure that a meaningful value is computed. Even for instantaneous values, do not oversample, to avoid unnecessary load on the system being monitored.

### pmie Instance Names

When you specify a metric instance name (*#ident*) in a *pmie* expression, this is compared against the instance name supplied by the Performance Metrics Collection System (PMCS) as follows:

- If the given and PMCS name are the same, they are considered to match.
- The first two blank separated tokens are extracted from the PMCS name. If the given name is the same as either of these tokens, they match.

For some metrics, notably the per process (*proc.xxx.xxx*) metrics, the first token in the PMCS instance name is impossible to determine at the time you are writing expressions. The above policy circumvents this problem.

### pmie Error Detection

The parser used in the current implementation of *pmie* is quite fragile in the face of syntax errors. It is suggested that you check any problem expression individually in interactive mode:

```
pmie -i
f
expression
^D
l
```

If the expression was parsed, its internal representation is shown:

```
s
```

The expression is evaluated once and its value printed:

```
q
```

Also beware that it is not always possible to detect semantic errors at parse time. This happens when a performance metric descriptor is not available from the named host at this time. A warning is issued, and the expression is put on a wait list. The wait list is checked periodically (about every five minutes) to see if the metric descriptor has become available. If an error is detected at this time, a message is printed to the standard error stream (stderr), and the offending expression is put aside.

## Changing PCP Metric Values With pmstore

From time to time you may wish to change the value of a particular metric. Some metrics are counters that may need to be reset, and some are simply control variables for agents that collect performance metrics. When you need to change the value of a metric for any reason, the command to use is *pmstore*(1).

The basic syntax of the command is:

**pmstore** *metricname value*

There are also command-line flags to further specify the action. For example, the **-i** option restricts the change to one or more instances of the performance metric.

The *value* may be in one of several forms, according to the following rules.

1. If the metric has an integer type, then *value* should be an optional leading hyphen, followed either by decimal digits or “0x” and some hexadecimal digits. “0X” is also acceptable in lieu of “0x.”
2. If the metric has a floating point type, then *value* should be in the form of an integer (described above) or a fixed point number, or a number in scientific notation.
3. If the metric has a string type, then *value* is interpreted as a literal string of ASCII characters.
4. If the metric has an aggregate type, then an attempt is made to interpret *value* as an integer, or as a floating point number, or as a string. In the first two cases, the minimal word length encoding is used; for example, ‘‘123’’ would be interpreted as a 4-byte aggregate, and ‘‘0x100000000’’ would be interpreted as an 8-byte aggregate.

For complete information on *pmstore* usage and syntax, see the *pmstore* reference page.

*Chapter 3*

# The Performance Metrics Application Programming Interface (PMAPI)

This chapter describes the Performance Metrics Application Programming Interface (PMAPI) provided with the Performance Co-Pilot.

The PMAPI provides performance tool developers with access to all of the distributed services of the Performance Metrics Collection System (PMCS) of the PCP, and is the interface used by the PCP utilities. The PMAPI is also relevant to the application developer who creates customized performance utilities that require access to instantiated performance metrics and the metadata describing those metrics.

The most common use of performance monitoring utilities is a scenario where the PCP tools are executed on a workstation (the monitoring system), while the interesting performance data is collected on a remote system by a number of PMCS processes. These processes execute on both the monitoring workstation and one or more server systems. The server systems are the real object of performance investigations.

In the development of the PMAPI the most important question has been "how easily and quickly will this API enable the user to build new performance tools, or exploit existing tools for newly available performance metrics?" The PMAPI and the standard tools that used the PMAPI have enjoyed a symbiotic evolution throughout the development of the Performance Co-Pilot.

## Naming and Identifying Performance Metrics

Across all of the supported performance metric domains, there are a large number of performance metrics Each metric has its own structure and semantics. The Performance Co-Pilot presents a uniform interface to these

metrics above the PMAPI, independent of the source of the underlying metric data.

The PMCS uses an internal identification scheme that unambiguously associates a single integer with each known performance metric. This integer is known as the Performance Metric Identifier, or PMID. Above the PMAPI, a PMID is defined and manipulated with the typedef *pmID*.

Above the PMAPI, a Performance Metrics Name Space (PMNS) is used to provide a hierarchic classification of external metric names, and a one-to-one mapping of external names to internal PMIDs. A more detailed description of the PMNS can be found in “Performance Metrics Name Space” on page 10 of this guide.

Applications that use the PMAPI may have independent versions of a PMNS, constructed from an initialization file when the application starts. Not all PMIDs need be represented in the PMNS of every application. For example, an application that monitors disk traffic likely uses a name space that references only the PMIDs for I/O statistics. Other applications require a stable PMNS that can be assumed to be the same on all systems. The distributed implementation includes a default PMNS for just this purpose.

## Performance Metric Instances

When performance metric values are returned across the PMAPI to a requesting application, there may be more than one value instance for a particular metric; for example, independent counts for each CPU, or each process, or each disk, or each system call type, and so on. This multiplicity of values is not enumerated in the name space, but rather when performance metrics are delivered across the PMAPI.

Each performance metric is associated with an instance domain. Each instance domain is identified by a unique value, as defined by the following typedef declaration:

```
typedef unsigned long pmInDom
```

The special instance domain *PM_INDOM_NULL* is reserved to indicate that the metric has a single value (no instances).

Each individual instance, within an instance domain, is represented by an internal integer instance identifier. The special instance identifier *PM_IN_NULL* is reserved for the single value in the null instance domain. Performance metric values are delivered across the PMAPI as a set of instance identifier and value pairs.

Internal instance identifiers correspond one to one with external (textual) descriptions of the members of an instance domain. The syntax of the external names is more precisely defined to be an arbitrary sequence of characters, such that the initial sequence of non-space characters serves to uniquely name an instance, and the optional characters following the first space serve as additional descriptive text for the instance.

The difficult issue of transient performance metrics (for example, per-process information, hot-plug replaceable hardware modules, and so on) means that repeated requests for the same PMID may return different numbers of values, or some changes in the particular instance identifiers returned. This means applications need to be aware that metric instantiation is guaranteed to be valid at the time of collection only.

## Current PMAPI Context

When performance metrics are retrieved across the PMAPI, they are delivered in the context of a particular source of metrics, a point in time, and a profile of desired instances. This means that the application making the request has already negotiated across the PMAPI to establish the context in which the request should be executed.

A metrics source may be the current (and recent) performance data from a particular host, or an archive log of performance data collected by *pmlogger* at some distant host or earlier time. The metrics source is specified when the PMAPI contest is created by calling the *pmNewContext* function.

By default, the collection time for a performance metric is the current time of day, but may be set to some point in the past to request retrieval of historical performance metrics values or performance metrics metadata (for example, descriptions of performance metrics, or explanatory text). The collection time can be manipulated by calling the *pmSetMode* function.

The last component of a PMAPI context is an instance profile that may be used to control which particular instances of a performance metric should be retrieved. When a new PMAPI context is created, the initial state expresses an interest in all possible instances, to be collected at the current time. The instance profile can be manipulated using the functions *pmAddProfile* and *pmDelProfile*.

## Performance Metric Descriptions

For each defined performance metric, there is associated metadata, a Performance Metric Description (*pmDesc* structure), that describes the syntax and semantics of the performance metric. The *pmDesc* structure provides all of the information required to describe and manipulate a performance metric through the PMAPI. It has the following declaration:

```
/* Performance Metric Descriptor */
typedef struct {
   pmID    pmid;   /* unique identifier */
   int     type;   /* base data type (see below) */
   pmInDom indom;  /* instance domain */
   int     sem;    /* semantics of value (see below) */
   pmUnits units;  /* dimension and units (see below) */
} pmDesc;
```

```
/* pmDesc.type - data type of metric values */
#define PM_TYPE_NOSUPPORT -1   /* not in this version */
#define PM_TYPE_32        0    /* 32-bit signed integer */
#define PM_TYPE_U32       1    /* 32-bit unsigned integer */
#define PM_TYPE_64        2    /* 64-bit signed integer */
#define PM_TYPE_U64       3    /* 64-bit unsigned integer */
#define PM_TYPE_FLOAT     4    /* 32-bit floating point */
#define PM_TYPE_DOUBLE    5    /* 64-bit floating point */
#define PM_TYPE_STRING    6    /* array of char */
#define PM_TYPE_AGGREGATE 7  /* arbitrary binary data */
```

```
/* pmDesc.sem - semantics of metric values */
#define PM_SEM_COUNTER  1    /* cumulative counter
                                (monotonic increasing) */
#define PM_SEM_INSTANT  3    /* instant. value
                                continuous domain */
#define PM_SEM_DISCRETE 4    /* instant. value
                                discrete domain */
```

The *type* field in the *pmDesc* structure accommodates various encodings of a metric's value. If the value of a performance metric is of type *PM_TYPE_AGGREGATE* (or indeed *PM_TYPE_STRING*), the interpretation of the value is unknown to the PMCS. In these cases, the application using the value and the Performance Metrics Domain Agent (PMDA) providing the value must have some common understanding about how the value is structured and interpreted.

Each value for a performance metric is assumed to be drawn from a set of values that can be described in terms of their dimensionality and scale by a compact encoding, as follows:

1. The dimensionality is defined by a power, or index, in each of three orthogonal dimensions: Space, Time, and Count (dimensionless). For example, I/O throughput is $Space.Time^{-1}$, while the running total of system calls is Count, memory allocation is Space, and average service time is $Time.Count^{-1}$.
2. In each dimension we have defined a number of common scale values that may be used to better encode ranges that might otherwise exhaust the precision of a 32-bit value.

This information is encoded in the *pmUnits* data structure, which is embedded in the *pmDesc* structure.

```
/*
 * Encoding for the units (dimensions and
 * scale) for Performance Metric Values
 *
 * For example, a pmUnits struct of
 * { 1, -1, 0, PM_SPACE_MBYTE, PM_TIME_SEC, 0 }
 * represents Mbytes/sec, while
 * { 0, 1, -1, 0, PM_TIME_HOUR, 6 }
 * represents hours/million-events
 */
typedef struct {
   int dimSpace:4;   /* space dimension */
   int dimTime:4;    /* time dimension */
   int dimCount:4;   /* event dimension */
   int scaleSpace:4; /* one of PM_SPACE_* below */
   int scaleTime:4;  /* one of PM_TIME_* below */
   int scaleCount:4; /* one of PM_COUNT_* below */
} pmUnits;   /* dimensional units and scale of value */
/* pmUnits.scaleSpace */
```

```
#define PM_SPACE_BYTE 0  /* bytes */
#define PM_SPACE_KBYTE 1 /* Kilobytes (1024) */
#define PM_SPACE_MBYTE 2 /* Megabytes (1024^2) */
#define PM_SPACE_GBYTE 3 /* Gigiabytes (1024^3) */
#define PM_SPACE_TBYTE 4 /* Terabytes (1024^4) */

/* pmUnits.scaleTime */
#define PM_TIME_NSEC 0   /* nanoseconds */
#define PM_TIME_USEC 1   /* microseconds */
#define PM_TIME_MSEC 2   /* milliseconds */
#define PM_TIME_SEC 3    /* seconds */
#define PM_TIME_MIN 4    /* minutes */
#define PM_TIME_HOUR 5   /* hours */

/*
 * pmUnits.scaleCount (e.g. count events, syscalls,
 * interrupts, etc.) -- these are simply powers of 10,
 * and not enumerated here.
 * e.g. 6 for 10^6, or -3 for 10^-3
 */
#define PM_COUNT_ONE 0 /* 1 */
```

## Performance Metrics Values

An application may fetch (or store) values for a set of performance metrics, each with a set of associated instances, using a single *pmFetch* (or *pmStore*) function call. To accommodate this, values are delivered across the PMAPI in the form of a tree, rooted at a *pmResult* structure. This encoding is illustrated in Figure 3-1, and uses the following component data structures:

```
typedef struct {
   int len;                 /* length in bytes */
   char vbuf[1];            /* one or more values */
} pmValueBlock;

typedef struct {
   int inst;                /* instance identifier */
   union {
      pmValueBlock *pval;   /* pointer to value-block */
      long lval;            /* long value insitu */
   } value;
} pmValue;

typedef struct {
```

```
   pmID pmid;              /* metric identifier */
   int  numval;            /* number of values */
   int  valfmt;            /* value style, insitu or ptr */
   pmValue vlist[1];       /* set of instances/values */
} pmValueSet;

/* Result returned by pmFetch() */
typedef struct {
   struct timeval timestamp;/* stamped by collector */
   int numpmid;            /* number of PMIDs */
   pmValueSet *vset[1];    /* set of value sets */
} pmResult
```

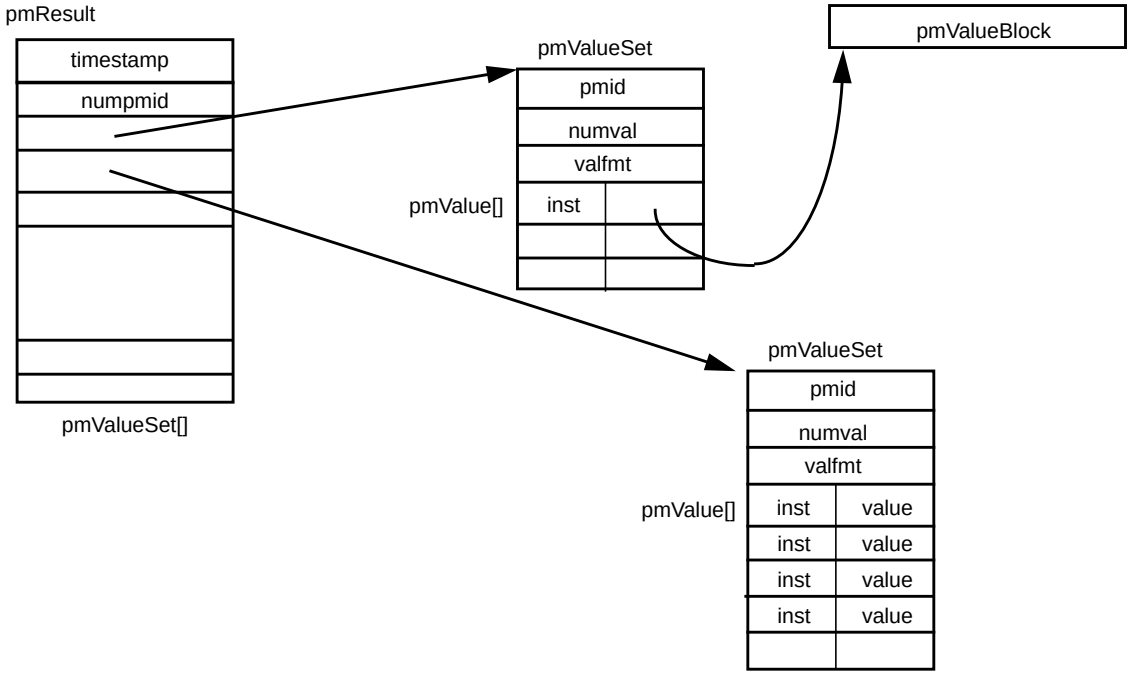


**Figure 3-1** A Structured Result for Performance Metrics From pmFetch

The *pmResult* structure contains a dynamically allocated array of *numpmid* pointers to *pmValueSets*, one *pmValueSet* per PMID. The *pmValueSet* structure in turn contains a dynamically allocated array of *numval pmValues*. Each *pmValue* is an instance identifier and value pair.

The field *valfmt* in the *pmValueSet* structure indicates if the values for this metric are stored in the *lval* field or held in associated *pmValueBlock* structures. The *pmValueBlock* structure can accommodate arbitrary-sized binary data, and is suited for “string-valued” metrics and metrics with aggregated or complex data types.

Along with the metric values, the PMAPI returns a timestamp with each aggregation of values that serves to identify when the performance metric values were collected. It typically is not be long before the metrics are

exported across the PMAPI. There is a question of exactly "when" individual metrics may have been collected, especially given their origin in potentially different Performance Metric Domains, and variability in the metric updating frequency at the lowest level of the Performance Metric Domain. The pragmatic approach is used with the PCP, in which the PMAPI implementation returns all of the metrics with values that are accurate as of the timestamp, as much as possible. The inaccuracy this introduces is small, and the additional burden of accurate individual timestamping for each returned metric value is neither warranted nor practical (from an implementation viewpoint).

The PMAPI provides functions to extract, rescale, and print values from the above structures.

## General Issues of PMAPI Programming Style and Interaction

The following sections specify the programming style used in the PMAPI.

### Variable Length Argument and Results Lists

All arguments and results involving a "list of something" are encoded as an array with an associated argument or function value to identify the number of elements in the array. This encoding scheme avoids both the *varargs* approach and sentinel-terminated lists.

Where the size of a result is known at the time of a call, it is the caller's responsibility to allocate (and possibly free) the storage, and the called function assumes the result argument is of an appropriate size.

Where a result is of variable size and that size cannot be known in advance (for example, *pmGetChildren*, *pmGetInDom*, *pmNameInDom*, *pmNameID*, *pmLookupText* and *pmFetch*), the underlying implementation uses dynamic allocation through *malloc* in the called routine, with the caller responsible for subsequently calling *free* to release the storage when no longer required. In the case of the result from *pmFetch*, there is a routine (*pmFreeResult*) to release the storage, due to the complexity of the data structure and the need to make multiple calls to *free* in the correct sequence. As a general rule, if the called routine returns an error status, then no allocation is done, the pointer to the

variable sized result is undefined, and *free* or *pmFreeResult* should not be called.

### PMAPI Error Handling

Where error conditions may arise, the functions that compose the PMAPI conform to a single, simple-error notification scheme, as follows:

- The function returns an *int*.
- Values greater than or equal to zero indicate no error, and perhaps some positive status: for example, the number of things really processed.
- Values less than zero indicate an error, with a global table of error conditions and error messages.

PMAPI library routines along the lines of *perror* are provided to translate error conditions into error messages.

The error condition is returned as the function value; there is no global error indicator (unlike *errno*). This is in attempt to anticipate and accommodate a programming environment that does not hamper the implementation of multi-threaded performance tools.

## PMAPI Procedural Interface

### PMAPI Name Space Services

#### pmLoadNameSpace

```
int pmLoadNameSpace(char *filename)
```

Before requesting any services involving a Performance Metrics Name Space (PMNS), the application must load the PMNS using *pmLoadNameSpace*.

The *filename* argument designates the PMNS of interest. For applications that do not require a tailored name space, the special value *PM_NS_DEFAULT* may be used for *filename*, to force a default PMNS to be established.

Externally a PMNS may be stored in either an ASCII format or a binary format. The utility *pmnscomp* is used to create the binary format from the ASCII format.

### pmLookupName

```
int pmLookupName(int numpmid, char *namelist[], pmID
pmidlist[])
```

Given a list in *namelist* containing *numpmid* full pathnames for performance metrics from a Performance Metrics Name Space (PMNS), *pmLookupName* returns the list of associated PMIDs through the *pmidlist* facility.

The result from *pmLookupName* is the number of names translated in the absence of errors, or an error indication. Note that argument definition and the error protocol guarantee a one to one relationship between the elements of *namelist* and *pmidlist*, such as if both lists contain exactly *numpmid* elements.

### pmGetChildren

```
int pmGetChildren(char *name, char ***offspring)
```

Given a full pathname to a node in the current Performance Metrics Name Space, as identified by *name*, return through *offspring* a list of the relative names of all of the immediate descendents of *name* in the current PMNS. As a special case, if *name* is an empty string, the immediate descendents of the root node in the PMNS is returned.

Normally, *pmGetChildren* returns the number of descendent names discovered, or a value less than zero for an error. The value zero indicates that the *name* is valid, and associated with a leaf node in the PMNS.

The resulting list of pointers (*offspring*) and the values (relative metric names) that the pointers reference are allocated by *pmGetChildren* with a single call to *malloc*, and it is the responsibility of the caller to issue a *free*(*offspring*) system call to release the space when it is no longer required. When the result of *pmGetChildren* is less than one, *offspring* is undefined (no space is allocated, and so calling *free* is counterproductive).

### pmNameID

```
int pmNameID(pmID pmid, char **name)
```

Given a Performance Metric ID through *pmid*, *pmNameID* determines the corresponding metric name, if any, in the Performance Metrics Name Space, and return this through *name*.

In the absence of errors, *pmNameID* returns zero. The *name* argument is a null-byte terminated string, allocated by *pmNameID* using *malloc*. It is the caller's responsibility to *free* the string when it is no longer required.

### pmTrimNameSpace

```
int pmTrimNameSpace(void)
```

If the current PMAPI context corresponds to an archive log of performance metrics (as collected by *pmlogger*), then the currently loaded Performance Metrics Name Space is trimmed to exclude metrics for which no description can be found in the archive. The PMNS is further trimmed to remove empty subtrees that do not contain any performance metric.

Since the PCP archives usually contain some subset of all metrics named in the default PMNS, *pmTrimNameSpace* effectively trims the application's PMNS to contain only the names of the metrics in the archive.

Prior to any trimming, the PMNS is restored to the state as of the completion of the last *pmLoadNameSpace* operation, so the effects of consecutive calls to *pmTrimNameSpace* with archive contexts are not cumulative.

If the current PMAPI context corresponds to a host, rather than an archive, the PMNS reverts to all names loaded into the PMNS at the completion of the last *pmLoadNameSpace* operation. For example, any trimming is undone.

### pmTraversePMNS

```
int pmTraversePMNS(char *name, void (*dometric)(char *))
```

The routine *pmTraversePMNS* may be used to perform a depth-first traversal of the PMNS.

The traversal starts at the node identified by *name* - if *name* is a null string, the traversal starts at the root of the PMNS. Usually *name* would be the pathname of a non-leaf node in the PMNS.

For each leaf node (an actual performance metric) found in the traversal, the user-supplied routine *dometric* is called with the full pathname of that metric in the PMNS as the single argument. This argument is null-byte terminated, and is constructed from a buffer that is managed internally to *pmTraversePMNS*. Consequently the value is only valid during the call to *dometric* - if the pathname needs to be retained, it should be copied using *strdup*(3C) before returning from *dometric*.

## PMAPI Instance Domain Services

### pmLookupInDom

```
int pmLookupInDom(pmInDom indom, char *name)
```

For the instance domain *indom*, in the current PMAPI context, locate the instance with the external identification given by *name*, and return the internal instance identifier.

Only the leading non-space characters of *name* are used to identify the instance.

### pmNameInDom

```
int pmNameInDom(pmInDom indom, int inst, char **name)
```

For the instance domain *indom*, in the current PMAPI context, locate the instance with the internal instance identifier given by *inst*, and return the full external identification through *name*.

The space for the value of *name* is allocated in *pmNameInDom* with *malloc*, and it is the responsibility of the caller to *free* the space when it is no longer required.

### pmGetInDom

```
int pmGetInDom(pmInDom indom, int **instlist, char
***namelist)
```

In the current PMAPI context, locate the description of the instance domain *indom*, and return through *instlist* the internal instance identifiers for all instances, and through *namelist* the full external identifiers for all instances. The number of instances found is returned as the function value (or less than zero to indicate an error).

The resulting lists of instance identifiers (*instlist* and *namelist*), and the names that the elements of *namelist* point to, are allocated by *pmGetInDom* with two calls to *malloc*, and it is the responsibility of the caller to *free*(*instlist*) and *free*(*namelist*) to release the space when it is no longer required. When the result of *pmGetInDom* is less than one, both *instlist* and *namelist* are undefined (no space is allocated, and so calling *free* is a singularly bad idea).

## PMAPI Description Services

### pmLookupDesc

```
int pmLookupDesc(pmID pmid, pmDesc *desc)
```

Given a Performance Metrics Identifier (PMID) as *pmid*, return the associated *pmDesc*, structure through the parameter *desc*, from the current PMAPI context. See “Performance Metric Descriptions” on page 118.

### pmLookupText

```
int pmLookupText(pmID pmid, int level, char **buffer)
```

Provided the source of metrics from the current PMAPI context is a host, retrieve descriptive text about the performance metric identified by *pmid*.

The argument level should be *PM_TEXT_ONELINE* for a one-line summary, or *PM_TEXT_HELP* for a more verbose description, suited to a help dialog.

The space pointed to by *buffer* is allocated in *pmLookupText* with *malloc*, and it is the responsibility of the caller to *free* the space when it is no longer required.

### pmLookupInDomText

```
int pmLookupInDomText(pmInDom indom, int level, char
**buffer)
```

Provided the source of metrics from the current PMAPI context is a host, retrieve descriptive text about the performance metrics instance domain identified by *indom*.

The argument *level* should be PM_TEXT_ONELINE for a one-line summary, or PM_TEXT_HELP for a more verbose description, suited to a help dialog.

The space pointed to by *buffer* is allocated in *pmLookupInDomText* with *malloc*, and it is the responsibility of the caller to *free* the space when it is no longer required.

## PMAPI Context Services

The following table shows which of the three components of a PMAPI context (metrics source, instance profile and collection time) are relevant for the correct execution of PMAPI functions. Those PMAPI functions not shown in this table either manipulate the PMAPI context directly, or are executed independently of the current PMAPI context.

**Table 3-1** Context Components of PMAPI Functions

| Function Name | Metrics Source | Instance Profile | Collection Time | Notes |
|---|---|---|---|---|
| *pmAddProfile* | | yes | | |
| *pmDelProfile* | | yes | | |
| *pmDupContext* | yes | yes | yes | |
| *pmFetch* | yes | yes | yes | |
| *pmFetchArchive* | yes | | yes | (5) |
| *pmGetArchiveEnd* | yes | | | (5) |
| *pmGetArchiveLabel* | yes | | | (5) |
| *pmGetInDom* | yes | | yes | (1) |

**Table 3-1** Context Components of PMAPI Functions

| Function Name | Metrics Source | Instance Profile | Collection Time | Notes |
|---|---|---|---|---|
| *pmGetInDomArchive* | yes | | | (5) |
| *pmLookupDesc* | yes | | | (2) |
| *pmLookupInDom* | yes | | yes | (1) |
| *pmLookupInDomArchive* | yes | | | (5) |
| *pmLookupInDomText* | yes | | | (4) |
| *pmLookupText* | yes | | | (3) |
| *pmNameInDom* | yes | | yes | (1) |
| *pmNameInDomArchive* | yes | | | (5) |
| *pmSetMode* | | | yes | |
| *pmStore* | yes | | | (6) |
| *pmTrimNameSpace* | yes | | | |

Notes:

1. A specific instance domain is included in the arguments to these routines, and the result is independent of the instance profile for any PMAPI context.
2. The metadata that describes a performance metric is sensitive to the source of the metrics, but independent of any instance profile, and independent of the collection time.
3. The text associated with a metric is assumed to be invariant with time and is definitely insensitive to the current members of the instance domain. In all cases this information is unavailable from an archive context (it is not included in the archive logs), and is directly available from the PMCD in the other cases.
4. This operation is supported only for PMAPI contexts where the source of the metrics is a host.
5. This operation is supported only for PMAPI contexts where the source of the metrics is an archive.

6. This operation is only supported for contexts where the source of the metrics is a host. Further, the instance identifiers are included in the argument to the routine, and the effects are immediate upon the current values of the metrics (retrospective changes are not allowed). Consequently, from the current PMAPI context, neither the instance profile nor the collection time influence the result of this routine.

### pmNewContext

```
int pmNewContext(int type, char *name)
```

The *pmNewContext* function may be used to establish a new PMAPI context. The source of the metrics is identified by *name*, and may be either a host (where *type* is *PM_CONTEXT_HOST*) or an archive file (where *type* is *PM_CONTEXT_ARCHIVE*). The initial instance profile is set up to select all instances in all instance domains, and the initial collection time is the "current" time at the time of each request for a host, or the time at the start of the log for an archive. In the case of an archive, the initial collection time results in the earliest set of metrics being returned from the archive at the first *pmFetch*.

Once established, the association between a PMAPI context and source of metrics is fixed for the life of the context; however, routines are provided to independently manipulate both the instance profile and the collection time components of a context.

The function returns a "handle" that may be used with subsequent calls to *pmUseContext*.

This new PMAPI context stays in effect for all subsequent calls across the PMAPI, until another call to *pmNewContext* is made, or the context is explicitly changed with a call to *pmDupContext* or *pmUseContext*.

### pmDestroyContext

```
int pmDestroyContext(int handle)
```

The PMAPI context identified by *handle* is destroyed. Typically this would imply terminating a connection to a PMCD or closing an archive file, and orderly clean-up.

The PMAPI context must have been previously created using *pmNewContext* or *pmDupContext*.

On success, *pmDestroyContext* returns zero. If *handle* was the current PMAPI context, then the current context becomes undefined. This means the application must explicitly re-establish a valid PMAPI context with *pmUseContext*, or create a new context with *pmNewContext* or *pmDupContext*, before the next PMAPI operation that requires a PMAPI context.

#### pmDupContext

```
int pmDupContext(void)
```

Replicate the current PMAPI context (source, instance profile and collection time). This routine returns a "handle" for the new context, that may be used with subsequent calls to *pmUseContext*.

The newly replicated PMAPI context becomes the current context.

#### pmUseContext

```
int pmUseContext(int handle)
```

Calling *pmUseContext* causes the current PMAPI context to be set to the context identified by *handle*. The value of *handle* must be one returned from an earlier call to *pmNewContext* or *pmDupContext*.

Below the PMAPI, all contexts used by an application are saved in their most recently modified state, so *pmUseContext* restores the context to the state it was in the last time the context was used, not the state of the context when it was established.

#### pmWhichContext

```
int pmWhichContext(void)
```

Returns the "handle" for the current PMAPI context (source, instance profile, and collection time).

#### pmAddProfile

```
int pmAddProfile(pmInDom indom, int numinst, int instlist[])
```

Add new instance specifications to the instance profile of the current PMAPI context. In the simplest variant, the list of instances identified by the *instlist* argument for the *indom* instance domain are added to the instance profile. The list of instance identifiers contains *numinst* values. If *indom* equals *PM_INDOM_NULL*, or *numinst* is zero, then all instance domains are selected. If *instlist* is (*int* *) *0*, then all instances are selected.

To enable all available instances in all domains, use the syntax

```
pmAddProfile(PM_INDOM_NULL, 0, (int *)0).
```

### pmDelProfile

```
int pmDelProfile(pmInDom indom, int numinst, int instlist[])
```

Delete instance specifications from the instance profile of the current PMAPI context. In the simplest variant, the list of instances identified by the *instlist* argument for the *indom* instance domain is removed from the instance profile. The list of instance identifiers contains *numinst* values.

If *indom* equals *PM_INDOM_NULL*, then all instance domains are selected for deletion. If *instlist* is (*int* *) *0*, then all instances in the selected domains are removed from the profile.

To disable all available instances in all domains, use the syntax:

```
pmDelProfile(PM_INDOM_NULL, 0, (int *)0)
```

### pmSetMode

```
int pmSetMode(int mode, struct timeval *when, int delta)
```

This routine is used to define the collection time and mode for accessing performance metrics and meta-data in the current PMAPI context. This mode affects the semantics of subsequent calls to the following PMAPI routines; *pmFetch*, *pmFetchArchive*, *pmLookupDesc*, *pmGetInDom*, *pmLookupInDom* and *pmNameInDom*.

If *mode* is *PM_MODE_LIVE*, then all information is returned from the active pool of performance metrics as of the time that the PMAPI call is made, and the other two parameters to *pmSetMode* are ignored. *PM_MODE_LIVE* is the default mode when a new PMAPI context of type *PM_CONTEXT_HOST* is created.

If the mode is not *PM_MODE_LIVE*, then the *when* parameter defines a time origin, and all requests for meta-data (metrics descriptions and instance identifiers from the instance domains) are processed to reflect the state of the meta-data as of the time origin. For example, we use the last state of this information at, or before, the time origin.

If the *mode* is *PM_MODE_INTERP* then, in the case of *pmFetch*, the underlying code uses an interpolation scheme to compute the values of the metrics from the values recorded for times in the proximity of the time origin. A mode of *PM_MODE_INTERP* may be used with either an archive context, or a host context (in which case the subsequent PMAPI functions are serviced from the log, for example for VCR-replay).

If the *mode* is *PM_MODE_FORW*, then, in the case of *pmFetch*, the collection of recorded metric values are scanned forward, until values for at least one of the requested metrics is located after the time origin. Then all requested metrics stored in the log or archive at that time are returned with the corresponding timestamp. A mode of *PM_MODE_FORW* may be used with either an archive context, or a host context (in which case the subsequent PMAPI functions are serviced from the log, for example for VCR-replay).

If the *mode* is *PM_MODE_BACK*, then the situation is the same as for *PM_MODE_FORW*, except a *pmFetch* are serviced by scanning the collection of recorded metrics backward for metrics before the time origin.

For modes other than *PM_MODE_LIVE*, after each successful *pmFetch*, the time origin is reset to the timestamp returned through the *pmResult*. The *pmSetMode* parameter *delta* defines an additional number of milliseconds that should be used to adjust the time origin (forward or backward), after the new time origin from the *pmResult* has been determined.

Using these mode options, an application can implement replay, playback, fast forward, or reverse for performance metric values held in the PMCS log by alternating calls to *pmSetMode* and *pmFetch*.

For example, the following code fragment may be used to dump only those values recorded in an archive in correct temporal sequence, for a selected set of performance metrics:

```
pmNewContext(PM_CONTEXT_ARCHIVE, "myarchive");
while (pmFetchArchive(npmid, pmidlist, &result) !=
PM_ERR_EOL) {
```

```
    /*
    * process real metric values as of result->timestamp
    */
    pmFreeResult(result);
}
```

Alternatively, to replay interpolated metrics from the log in reverse chronological order, at 10-second intervals (of recorded time), the following code fragment could be used:

```
struct timeval mytime;

mytime.tv_sec = 0x7fffffff;
pmSetMode(PM_MODE_BACK, &mytime, 0);
pmFetchArchive(&result);
mytime = result->timestamp;
pmSetMode(PM_MODE_INTERP, &mytime, -10000);

while (pmFetch(npmid, pmidlist, &result) != PM_ERR_EOL) {
    /*
     * process interpolated metric values as of
     * result->timestamp
     */
    pmFreeResult(result);
}
```

### pmReconnectContext

```
int pmReconnectContext(int handle)
```

As a consequence of network, host, or Performance Metrics Coordination Daemon (PMCD) failures, an application's connection to a PMCD may be established and then lost.

The routine *pmReconnectContext* allows an application to request that the PMAPI context identified by *handle* should be re-established, provided the associated PMCD is accessible.

To avoid flooding the system with reconnect requests, *pmReconnectContext* attempts a reconnection only after a suitable delay from the previous attempt. This imposed restriction on the reconnect re-try time interval uses an exponential back-off so that the initial delay is 5 seconds after the first unsuccessful attempt, then 10 seconds, then 20 seconds, then 40 seconds, and then 80 seconds thereafter.

If the reconnection succeeds, *pmReconnectContext* returns *handle*. Note that even in the case of a successful reconnection, *pmReconnectContext* does not change the current PMAPI context.

## PMAPI Metrics Services

### pmFetch

```
int pmFetch(int numpmid, pmID pmidlist[], pmResult **result)
```

The most common operation is likely to be calls to *pmFetch*, specifying a list of PMIDs (for example, as constructed by *pmLookupName*) through *pmidlist* and *numpmid*. The call to *pmFetch* is executed in the context of a source of metrics, instance profile, and collection time, previously established by calls to the routines described in “PMAPI Context Services” on page 129.

The principal result from *pmFetch* is returned as a tree structured *result*, described in the section “Performance Metrics Values” on page 120.

If one value (for example, associated with a particular instance) for a requested metric is unavailable at the requested time, then there is no associated *pmValue* structure in the result. If there are no available values for a metric, then *numval* is zero and the associated *pmValue*[ ] instance is empty. (*valfmt* is undefined in these circumstances, but *pmid* is correctly set to the PMID of the metric with no values.)

As an extension of this protocol, if the PMCS is able to provide a reason why no values are available for a particular metric, this is encoded as a standard error code in the corresponding *numval*. Since the error codes are all negative, values for a requested metric are unavailable if *numval* is less than or equal to zero.

The argument definition and the result specifications have been constructed to ensure that for each PMID in the requested *pmidlist* there is exactly one *pmValueSet* in the result, and that the PMIDs appear in exactly the same sequence in both *pmidlist* and *result*. This makes the number and order of entries in *result* completely deterministic, and greatly simplifies the application programming logic after the call to *pmFetch*.

The result structure returned by *pmFetch* is dynamically allocated using one or more calls to *malloc* and specialized allocation strategies, and should be released when no longer required by calling *pmFreeResult*. Under no circumstances should *free* be called directly to release this space.

As common error conditions are encoded in the result data structure, we'd expect only cataclysmic events to cause an error value to be returned. Otherwise the value returned by the *pmFetch* function is zero.

#### pmFreeResult

```
void pmFreeResult(pmResult *result)
```

Release the storage previously allocated for a result by *pmFetch*.

#### pmStore

```
int pmStore(pmResult *request)
```

In some special cases it may be helpful to modify the current values of performance metrics in one or more underlying Performance Metric Domains, for example to reset a counter to zero, or to modify a "metric," which is a control variable within a Performance Metric Domain.

The routine *pmStore* is a lightweight inverse of *pmFetch*. The caller must build the *pmResult* data structure (which could have been returned from an earlier *pmFetch* call) and then call *pmStore*.

It is an error to pass a *request* to *pmStore* in which the *numval* field within any of the *pmValueSet* structure has a value less than one.

The current PMAPI context must be one with a host as the source of metrics, and the current value of the nominated metrics is changed. For example, *pmStore* cannot be used to make retrospective changes to information in either the PMCS logs or an archive.

### PMAPI Archive Services

#### pmGetArchiveLabel

```
int pmGetArchiveLabel(int handle, pmLogLabel *lp)
```

The routine *pmGetArchiveLabel* may be used to fetch the label record from an archive log that has already been opened using *pmNewContext* or *pmDupContext*, and thereby associated with the current PMAPI context.

The structure returned through *lp* is as follows:

```
/*
 * Label Record at the start of every log file
 */
typedef struct {
    int ll_magic;          /* PM_LOG_MAGIC or
                           log format version no.*/
    pid_t ll_pid;           /* PID of logger ^/
    struct timeval ll_start; /* start of this log */
    int ll_seq;             /* log sequence no. */
    char ll_hostname[MAXHOSTNAMELEN];
                           /* name of collection host */
    char ll_tz[40];         /* $TZ at collection host */
} pmLogLabel;
```

#### pmGetArchiveEnd

```
int pmGetArchiveEnd(struct timeval *tvp)
```

Assuming the current PMAPI context is associated with an archive log, *pmGetArchiveEnd* attempts to find the logical end of file (after the last complete record in the archive), and return the last recorded timestamp with *tvp*. This timestamp may be passed to *pmSetMode* to reliably position the context at the last valid log record, for example in preparation for subsequent reading in reverse chronological order.

For archive logs that are not concurrently being written, the physical end of file and the logical end of file are co-incident. However if an archive log is being written by *pmlogger* at the same time an application is trying to read the archive, the logical end of file may be before the physical end of file due to write buffering that is not aligned with the logical record boundaries.

#### pmGetInDomArchive

```
int pmGetInDomArchive(pmInDom indom, int **instlist, char
***namelist)
```

Provided the current PMAPI context is associated with an archive log, *pmGetInDomArchive* scans the union of all the instance domain metadata for the instance domain *indom*, and returns through *instlist* the internal instance identifiers for all instances, and through *namelist* the full external identifiers for all instances.

This routine is a specialized version of the more general PMAPI routine *pmGetInDom*.

The number of instances found is returned as the function value (or less than zero to indicate an error).

The resulting lists of instance identifiers (*instlist* and *namelist*), and the names that the elements of *namelist* point to, are allocated by *pmGetInDomArchive* with two calls to *malloc*, and it is the responsibility of the caller to *free*(*instlist*) and *free*(*namelist*) to release the space when it is no longer required.

When the result of *pmGetInDomArchive* is less than one, both *instlist* and *namelist* are undefined (no space is allocated, so calling *free* is a singularly bad idea).

#### pmLookupInDomArchive

```
int pmLookupInDomArchive(pmInDom indom, char *name)
```

Provided the current PMAPI context is associated with an archive log, *pmLookupInDomArchive* scans the union of all the instance domain metadata for the instance domain *indom*, locates the first instance with the external identification given by *name*, and returns the internal instance identifier.

This routine is a specialized version of the more general PMAPI routine *pmLookupInDom*.

Only the leading non-space characters of *name* are used to identify the instance.

The *pmLookupInDomArchive* routine returns a positive instance identifier on success.

#### pmNameInDomArchive

```
int pmNameInDomArchive(pmInDom indom, int inst, char **name)
```

Provided the current PMAPI context is associated with an archive log, *pmNameInDomArchive* scans the union of all the instance domain metadata for the instance domain *indom*, locates the first instance with the internal instance identifier given by *inst*, and returns the full external instance identification through *name*.

This routine is a specialized version of the more general PMAPI routine *pmNameInDom*.

The space for the value of *name* is allocated in *pmNameInDomArchive* with *malloc*, and it is the responsibility of the caller to *free* the space when it is no longer required.

#### pmFetchArchive

```
int pmFetchArchive(pmResult **result)
```

This is a variant of *pmFetch* that may be used only when the current PMAPI context is associated with an archive. The *result* is instantiated with all of the metrics (and instances) from the next archive record; consequently there is no notion of a list of desired metrics, and the instance profile is ignored.

It is expected that *pmFetchArchive* would be used to create utilities that scan archive logs, and the more common access to the archives would be through the *pmFetch* interface.

## PMAPI Ancillary Support Services

The routines described in this section provide services that are complementary to, but not necessarily a part of, the distributed manipulation of performance metrics delivered by the PMCS.

#### pmErrStr

```
char *pmErrStr(int code)
```

This routine translates an error code into a text string, suitable for generating a diagnostic message. By convention, all error codes are negative. The small values are assumed to be negated versions of the UNIX error codes as defined in <*errno.h*>, and the strings returned are as per *strerror*(3C). The

large, negative error codes are PMAPI error conditions, and *pmErrStr* returns an appropriate PMAPI error string, as determined by *code*.

The string value is held in a single static buffer, so the returned value is only valid until the next call to *pmErrStr*.

### pmExtractValue

```
int pmExtractValue(int valfmt, pmValue *ival, int itype,
pmAtomValue *oval, int otype)
```

The *pmValue* structure is embedded within the *pmResult* structure, which is used to return one or more performance metrics; see the description of *pmFetch*.

All performance metric values may be encoded in a *pmAtomValue* union, defined as follows;

```
/* Generic Union for Value-Type conversions */
typedef union {
    _int32_t   l;     /* 32-bit signed */
    _uint32_t  ul;    /* 32-bit unsigned */
    _int64_t   ll;    /* 64-bit signed */
    _uint64_t  ull;   /* 64-bit unsigned */
    float      f;     /* 32-bit floating point */
    double     d;     /* 64-bit floating point */
    char       *cp;   /* char ptr */
    void       *vp;   /* void ptr */
} pmAtomValue;
```

The routine *pmExtractValue* provides a convenient mechanism for extracting values from the *pmValue* part of a *pmResult* structure, optionally converting the data type, and making the result available to the application programmer.

The *itype* argument defines the data type of the input value held in *ival* according to the storage format defined by *valfmt* (see *pmFetch*). The *otype* argument defines the data type of the result to be placed in *oval*. The value for *itype* is typically extracted from a *pmDesc* structure, following a call to *pmLookupDesc* for a particular performance metric.

Table 3-2 defines the various possibilities for the type conversion. The input type (*itype*) is shown vertically, and the output type (*otype*) is shown horizontally. The following rules apply:

- Y means the conversion is always acceptable.
- N means the conversion can never be performed (the function returns *PM_ERR_CONV*).
- P means the conversion may lose accuracy (but no error status is returned).
- T means the result may be subject to high-order truncation (in which case the function returns *PM_ERR_TRUNC*).
- S means the conversion may be impossible due to the sign of the input value (in which case the function returns *PM_ERR_SIGN*).

If an error occurs, *oval* is set to zero (or NULL). Note that some of the conversions involving the types *PM_TYPE_STRING* and *PM_TYPE_AGGREGATE* are indeed possible, but are marked N; the rationale is that *pmExtractValue* should not attempt to duplicate functionality already available in the C library through *sscanf* and *sprintf*.

**Table 3-2** PMAPI Type Conversion

| TYPE | 32 | U32 | 64 | U64 | FLOAT | DBLE | STRING | AGGR |
|---|---|---|---|---|---|---|---|---|
| **32** | Y | S | Y | S | P | P | N | N |
| **U32** | T | Y | Y | Y | P | P | N | N |
| **64** | T | T,S | Y | S | P | P | N | N |
| **u64** | T | T | T | Y | P | P | N | N |
| **FLOAT** | P,T | P,T,S | P,T | P,T,S | Y | Y | N | N |
| **DBLE** | P,T | P,T,S | P,T | P,T,S | P | Y | N | N |
| **STRING** | N | N | N | N | N | N | Y | N |
| **AGGR** | N | N | N | N | N | N | N | Y |

In the cases where multiple conversion errors could occur, the first encountered error is returned, and the order of checking is not defined.

If the output conversion is to one of the pointer types, such as *otype* *PM_TYPE_STRING* or *PM_TYPE_AGGREGATE*, then the value buffer is allocated by *pmExtractValue* using *malloc*, and it is the caller's responsibility to *free* the space when it is no longer required.

Although this function appears rather complex, it has been constructed to assist the development of performance tools that convert values, whose type is only known through the *type* field in a *pmDesc* structure, into a canonical type for local processing.

### pmConvScale

```
int pmConvScale(int type, pmAtomValue *ival, pmUnits *iunit,
pmAtomValue *oval, pmUnits *ounit)
```

Given a performance metric value pointed to by *ival*, multiply it by a scale factor and return the value in *oval*. The scaling takes place from the units defined by *iunit* into the units defined by *ounit*. Both input and output units must have the same dimensionality.

The performance metric type for both input and output values is determined by *type*, the value for which is typically extracted from a *pmDesc* structure, following a call to *pmLookupDesc* for a particular performance metric.

*pmConvScale* is most useful when values returned through *pmFetch* (and possibly extracted using *pmExtractValue*) need to be normalized into some canonical scale and units for the purposes of computation.

### pmUnitsStr

```
char *pmUnitsStr(pmUnits *pu)
```

As an aid to labeling graphs and tables, or for error messages, *pmUnitsStr* takes a dimension and scale specification as per *pu*, and returns the corresponding text string.

For example, if **pu* was *{1, -2, 0, PM_SPACE_MBYTE, PM_TIME_MSEC, 0}*, then the result string would be "Mbyte / sec^2".

The string value is held in a single static buffer, so concurrent calls to *pmUnitsStr* may not produce the desired results.

#### pmIDStr

```
char *pmIDStr(pmID pmid)
```

For use in error and diagnostic messages, return a “human readable” version of the specified PMID, with each of the *domain*, *cluster*, and *item* subfields appearing as decimal numbers, separated by periods.

The string value is held in a single static buffer, so concurrent calls to *pmIDStr* may not produce the desired results.

#### pmInDomStr

```
char *pmInDomStr(pmInDom indom)
```

For use in error and diagnostic messages, return a “human readable” version of the specified instance domain identifier, with each of the *domain* and *serial* subfields appearing as decimal numbers, separated by periods.

The string value is held in a single static buffer, so concurrent calls to *pmInDomStr* may not produce the desired results.

#### pmTypeStr

```
char *pmTypeStr(int type)
```

Given a performance metric type, produce a terse ASCII equivalent, appropriate for use in error and diagnostic messages.

Examples are “32” (for *PM_TYPE_32*), “U64” (for *PM_TYPE_U64*), “AGGREGATE” (for *PM_TYPE_AGGREGATE*), and so on.

The string value is held in a single static buffer, so concurrent calls to *pmTypeStr* may not produce the desired results.

#### pmAtomStr

```
char *pmAtomStr(pmAtomValue *avp, int type)
```

Given the *pmAtomValue* identified by *avp*, and a performance metric *type*, generate the corresponding metric value as a string, suitable for diagnostic or report output.

The string value is held in a single static buffer, so concurrent calls to *pmAtomStr* may not produce the desired results.

#### pmPrintValue

```
void pmPrintValue(FILE *f, int valfmt, int type, pmValue
*val, int minwidth)
```

The value of a single performance metric (as identified by *val*) is printed on the standard I/O stream identified by *f*. The value of the performance metric is interpreted according to the format of *val* as defined by *valfmt* (from a *pmValueSet* within a *pmResult*) and the generic description of the metric's type from a *pmDesc* structure, passed in through *type*.

The output may be padded to be at least *minwidth* characters wide.

#### pmSortInstances

```
void pmSortInstances(pmResult *result)
```

The routine *pmSortInstances* may be used to guarantee that for each performance metric in the result from *pmFetch*, the instances are in ascending instance identifier sequence.

This is most useful when trying to compute rates from two consecutive *pmFetch* results.

## PMAPI Programming Issues and an Example

The following issues and examples are provided to enable you to create better custom performance monitoring tools.

The source code for a sample client (*pmclient*) using the PMAPI, is shipped as part of the *pcp_client.sw.demo* subsystem of the Performance Co-Pilot product. See the *pmclient*(1) reference page, and the source code, located in the directory */usr/demos/PerfCoPilot/pmclient*.

## Symbolic Association Between a Metric's Name and Value

A common problem in building specific performance tools is how to maintain the association between a performance metric's name, its access (instantiation) method, and the application program variable that contains the metric's value. Generally this results in code that is easily broken by bug fixes or changes in the underlying data structures. The PMAPI provides a uniform way of instantiating and accessing the values independent of the underlying implementation, although it does not solve the name-variable association problem. However, it does provide a framework within which a manageable solution may be developed.

Fundamentally, the goal is to be able to name a metric, and reference the metric's value in a manner that is independent of the order of operations on other metrics; for example, to associate the macro *BINGO* with the name "irix.sys.statistic.bingo," and then be able to use *BINGO* to get at the value of the corresponding metric.

The one-to-one association between the ordinal position of the metric names is input to *pmLookupName* and the PMIDs returned by this routine, and the one-to-one association between the PMIDs input to *pmFetch* and the values returned by this routine provide the basis for an automated solution.

The tool *pmgenmap* takes the specification of a list of metric names and symbolic tags, in the order they should be passed to *pmLookupName* and *pmFetch*. For example:

```
# one line comment

mystuff {
    irix.sys.statistic.bingo  BINGO
    oracle.latchstats.lru.miss  MISSED
}
```

The above *pmgenmap* input produces the following C code, suitable for including with the `#include` statement:

```
/*
 * Performance Metrics Name Space Map
 * Built by pmgenmap from the file
 * /usr/people/kenmcd/swa/ptg/src/kstat.pcp/x
 * on Thu Feb 24 20:37:53 EST 1994
 *
```

```
 * Do not edit this file!
 */

/* one line comment */

char *mystuff[] = {
   #define BINGO 0
      “irix.sys.statistic.bingo”,
   #define MISSED 1
      “oracle.latchstats.lru.miss”,
};
```

## Initializing New Metrics

Using the code generated by pmgenmap, we are now able to easily initialize the application’s connection to the PMSC as follows:

```
#define MAX_MID 3

int trip = 0;
int numpmid = sizeof(mystuff)/sizeof(mystuff[0]);
double duration;
pmResult *resp;
pmResult *prev;
pmID pmidlist[MAX_MID];

pmLoadNameSpace(PM_NS_DEFAULT);
pmLookupName(numpmid, mystuff, pmidlist);
```

At this stage, *pmidlist* contains the PMID for the two metrics of interest.

## Iterative Processing of Values

Assuming the tool is required to report values every five seconds, use code similar to the following:

```
while (1) {
   pmFetch(numpmid, pmidlist, &resp);
   if (trip) {
      duration = tv_sub(&resp->timestamp, &prev->timestamp);
   /*
    * irix.sys.boring.bozo is an instantaneous value,
```

```
     * so report the most recent value
     * oracle.latchstats.lru.miss is a free running
     * counter, so report the rate over the last two
     * samples
     */
    printf("%6d %5.2f\n",
        resp->vset[BOZO]->vlist[0].value.lval,
        (resp->vset[MISSED]->vlist[0].value.lval -
        prev->vset[MISSED]->vlist[0].value.lval) /
        duration);
    }

    if (trip >= 1)
        pmFreeResult(prev);
    else
        trip++;

    prev = resp;

    sleep(5);
}
```

## Accommodating Program Evolution

The flexibility provided by the PMAPI and the *pmgenmap* utility is demonstrated by the following example; consider the requirement to report a third metric "irix.sys.boring.new" (an instantaneous value) in the middle of the two already reported.

Add the line

```
irix.sys.boring.new  NEW
```

to the middle of the specification file, regenerate the *#include* file, and amend the *printf* statement as follows:

```
printf("%6d %6d %5.2f\n",
    resp->vlist[BOZO]->vlist[0].value.lval,
    resp->vlist[NEW]->vlist[0].value.lval,
    (resp->vlist[MISSED]->vlist[0].value.lval -
        prev->vlist[MISSED]->vlist[0].value.lval) /
        duration);
```

*Chapter 4*

# Extending and Refining the PCP Toolkit

The Performance Co-Pilot (PCP) has been developed to be fully extensible. The following sections describe various facilities provided to allow you to extend and customize the PCP for your site.

- "PCP Client Development" on page 149 describes the basic libraries and tools available to the PCP developer.
- "PMNS Management" on page 152 describes the rules for creating your own Performance Metrics Name Spaces.
- "PMDA Development" on page 157 describes the rules for creating your own Performance Metrics Domain Agent software.

## PCP Client Development

Application developers are encouraged to create new PCP client applications to monitor or display performance metrics in a manner that is particularly relevant to a specific site, application suite, or processing environment.

These client applications use the routines described in Chapter 3, "The Performance Metrics Application Programming Interface (PMAPI)." The PMAPI provides the following services to the PCP clients:

1. Connection to sources of performance metrics that may be on the local host, or a remote host, or an archive log (previously created with the *pmlogger*(1) utility).
2. Name translation services for performance metrics within a Performance Metrics Name Space (PMNS).
3. Retrieval of metadata describing the performance metrics from the source of those metrics.
4. Fetching arbitrary groups of performance metrics values.

5. Extraction and manipulation of performance metric values.

## PMAPI Compilation Support

The PMAPI is designed for programs written in C.

The header file */usr/include/pcp/pmapi.h* defines the function prototypes and data structures used at the PMAPI, so C code using the PMAPI requires the following statement:

```
#include <pcp/pmapi.h>
```

Occasionally, applications may require access to internal routines that underpin the PMAPI, and in these cases, the header file */usr/include/pcp/impl.h* must also be included after *pmapi.h*.

## The pmgenmap Utility

Given one or more lists of metric names, the *pmgenmap*(1) utility generates C declarations and *cpp*(1) macros suitable for use across the PMAPI. These generated C constructs simplify the programmer effort required to translate metric names and manipulate the corresponding values of the metrics.

The input to *pmgenmap* should consist of one or more lists of metric names of the form:

```
listname {
   metricname1 symbolname1
   metricname2 symbolname2
   ...
}
```

The above input generates C and *cpp* declarations of the form:

```
char *listname[] = {
#define symbolname1 0
   "metricname1",
#define symbolname2 1
   "metricname2",
   ...
};
```

The array declarations produced are suitable as parameters to *pmLookupName*(3) and the *#define*d constants may be used to index the *vsets* in the *pmResult* structure returned by a *pmFetch*(3) call.

For complete documentation of the *pmgenmap* utility, see the *pmgenmap* reference page.

## The PMAPI Library (libpcp)

The PMAPI routines are supplied in the DSO */usr/lib/pcp/libpcp.so*.

Linking an application with this library and locating it correctly when the program executes requires the following command line options to *cc* or *ld*:

```
cc ... -L/usr/lib/pcp -rpath /usr/lib/pcp -lpcp
```

## Example PMAPI Client

Installing the *pcp_client.sw.demo* software package places the source code, *Makefile* and installation scripts for a fully functional demonstration PCP client in the */usr/demos/PerfCoPilot/pmclient* directory.

## The libpcp_lite Library

The library */usr/lib/pcp/libpcp_lite.so* is a special “lightweight” version of */usr/lib/pcp/libpcp.so*. This lightweight library provides efficient access to a restricted subset of the PCP facilities. When the restrictions do not affect the PCP client, it may be linked with *libpcp_lite* instead of *libpcp*.

The restrictions imposed by *libpcp_lite* are as follows:

- No access to remote hosts.
- No access to archive logs.
- No support for VCR-mode replay.
- No help text services.
- Metric values may be fetched, but not modified.

- Only metrics in the *irix* and *proc* domains are supported.

The restrictions are significant; however, they allow an application linked with *libpcp_lite* to execute autonomously, without a Performance Metrics Coordinating Daemon *(*PMCD), and the overhead associated with inter-process communication (IPC) and context switching.

## PMNS Management

This section describes the syntax, semantics, and processing framework for the external specification of a PMNS as it might be loaded by the PMAPI routine *pmNameSpace*(3). The PMNS specification is a simple ASCII source file that can be easily edited. For reasons of efficiency, a binary format is also supported, and the utility *pmnscomp*(1) may be used to translate the ASCII source format into the binary format.

### PMNS Processing Framework

The PMNS specification is initially passed through *cpp*(1). This means the following facilities may be used in the specification:

- C-style comments
- #include directives
- #define directives and macro substitution
- conditional processing via #if, #endif, and so on

When *cpp* is executed, the “standard” include directories are the current directory and */usr/pcp/lib/pmns* (where some standard macros and default specifications may be found).

## PMNS Syntax

The general syntax for a non-leaf node[1] in the PMNS is as follows:

```
pathname {
    name   [pmid]
    ...
}
```

Here *pathname* is the full pathname from the root of the PMNS to this non-leaf node, with each component in the pathname separated by a period (.). The root node for the PMNS must have the special name *root*, but the prefix string "root." must be omitted from all other pathnames.

Each component in the pathname must begin with an alphabetic character, and be followed by zero or more alphanumeric characters or the underscore (_) character. For alphabetic characters in a pathname component, upper and lower case are distinguished.

For example, refer to the PMNS shown in Figure 4-1. The correct pathname for the right-most non-leaf node is *cpu.util*, not *root.cpu.util*.

Non-leaf nodes in the PMNS may be defined in any order desired.

The descendent nodes are defined by the set of *names*, relative to the pathname of their parent non-leaf node. For the descendent nodes, leaf nodes have a *pmid* specification, but non-leaf nodes do not.

The syntax for the *pmid* specification has been chosen to help manage the allocation of Performance Metric IDs (PMIDs) across disjoint and autonomous domains of administration and implementation. Each *pmid* consists of three integer parts, separated by colons; for example, 14:27:11. This is intended to mirror the implementation hierarchy of performance metrics. The first integer identifies the domain in which the performance metric lies. Within a performance metrics domain, related metrics are often grouped into clusters. The second integer identifies the cluster and the third integer  the metric within the cluster.

[1] A PMNS is tree structured.  The leaf nodes are the full performance metric names.

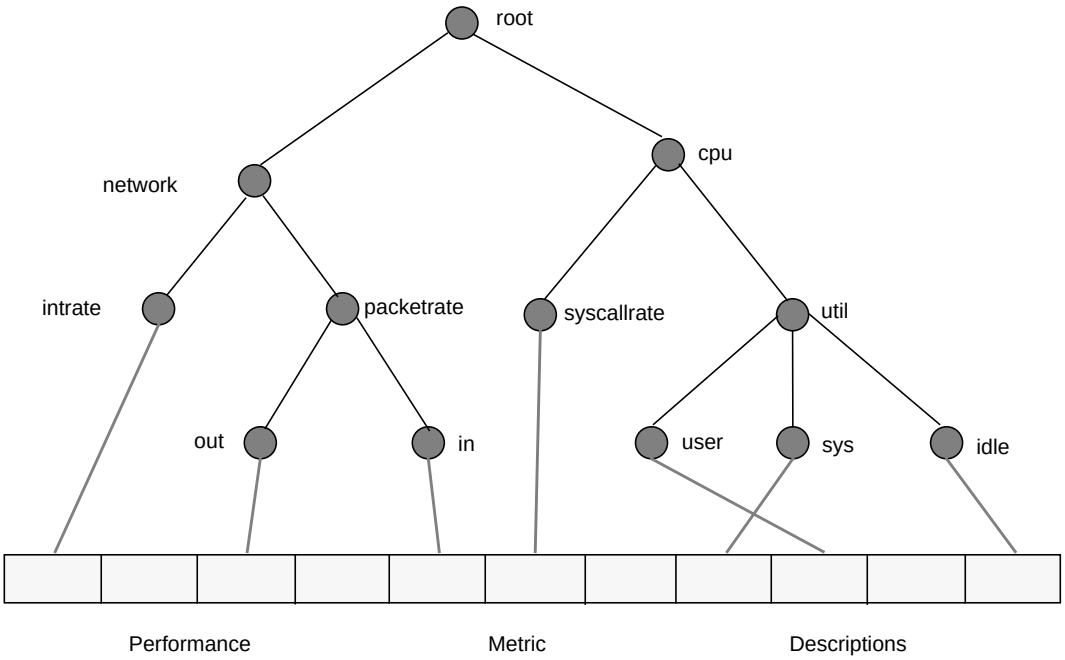


**Figure 4-1** A Small Performance Metrics Name Space (PMNS)

In practice, at least two of these components are likely to be macros in the PMNS specification source, and *cpp* converts the macros to integers. These macros for the initial components of the *pmid* are likely to be defined either in the standard include file <*stdpmid*> or in the current source file.

For complete documentation of the PMNS and associated utilities, see the *pmns(1), pmnscomp(1), pmnsadd(1),* and *pmnsdel(1)* reference pages.

## Example PMNS Specification

The PMNS specification for Figure 4-1 is as follows:

```
/*
 * PMNS Specification for Figure 4-1
 */

#include <stdpmid>

root {
    network
    cpu
}

#define NETWORK 26
network {
    intrate    IRIX:NETWORK:1
    packetrate
}

network.packetrate {
    in     IRIX:NETWORK:35
    out    IRIX:NETWORK:36
}

#define CPU 10
cpu {
    syscallrate    IRIX:CPU:10
    util
}

#define USER 20
#define KERNEL 21
#define IDLE 22
cpu.util {
    user    IRIX:CPU:USER
    sys     IRIX:CPU:KERNEL
    idle    IRIX:CPU:IDLE
}
```

## Using Local Variants of the Name Space (-n Option) in PMNS

All PCP utilities that require a PMNS permit command-line specification of an alternate name space through the option

**-n** *namespace*

The utilities described in this section may be used to create and amend either an alternate name space, or the default PMNS in the directory */usr/lib/pcp/pmns*.

## The pmnscomp Command

The *pmnscomp(1)* command compiles a PMNS in ASCII (plain text) form into a more efficient binary representation. The *pmLoadNameSpace*(3) library function is able to load this binary representation significantly faster than the equivalent ASCII representation.

The basic syntax for this command is:

**pmnscomp** [**-f**] [**-n** *namespace*] *outfile*

By convention, the name of the compiled namespace is that of the root file of the ASCII namespace, with *.bin* appended. For example, the root of the default PMNS is a file named *root* and the compiled version of the entire namespace is *root.bin*. The **-n** option allows you to specify a non-default namespace. The **-f** option forces an overwrite of the *outfile* if the *outfile* already exists.

Complete documentation of the *pmnscomp* command is found in the *pmnscomp(1)* reference page.

## The pmnsadd and pmnsdel Commands

The *pmnsadd*(1) command adds a subtree of new metric names into a PMNS). The new metric names are specified in a file given as an argument to the *pmnsadd* command, and must conform to the syntax for PMNS specifications, as documented in the *pmns*(4) reference page.

The *pmnsdel*(1) command removes a given subtree of names from a PMNS. The metric names to be deleted are specified on the command line with a pathname. All metrics to which the given pathname is a prefix are deleted.

Complete documentation on the *pmnsadd* and *pmnsdel* commands can be found in the *pmnsadd(1)* and *pmnsdel(1)* reference pages.

## PMDA Development

The PCP framework supports modular extension of the range of available performance metrics through an architecture in which a Performance Metrics Domain Agent (PMDA) is responsible for all metrics within its particular application or functional domain. For example, the proc PMDA is responsible for supplying performance metrics relating to Unix processes.

The generic architecture is as shown in Figure 4-2. The components below the Performance Metrics Application Programming Interface (PMAPI) are collectively known as the Performance Metrics Collection Subsystem (PMCS). The PMCS architecture is distributed, in the sense that the performance tools may be executing remotely; however, that is an irrelevant consideration in the context of a PMDA's behaviour:

As shown in Figure 4-2, the Performance Metrics Coordinating Daemon (PMCD) acts in a coordinating role, accepting requests from clients, routing the requests to one or more PMDAs, aggregating the responses from the PMDAs, and responding to the requesting client.

A PMDA is responsible for a set of performance metrics, in the sense that it must respond to requests from the PMCD for information about performance metrics, instance domains, and instantiated values. Requests from the PMCD are generated on behalf of performance tools that make requests to the PMCD.

New performance metrics are incorporated into the PMCS by creating a PMDA, then re-configuring the PMCD to communicate with the new PMDA.

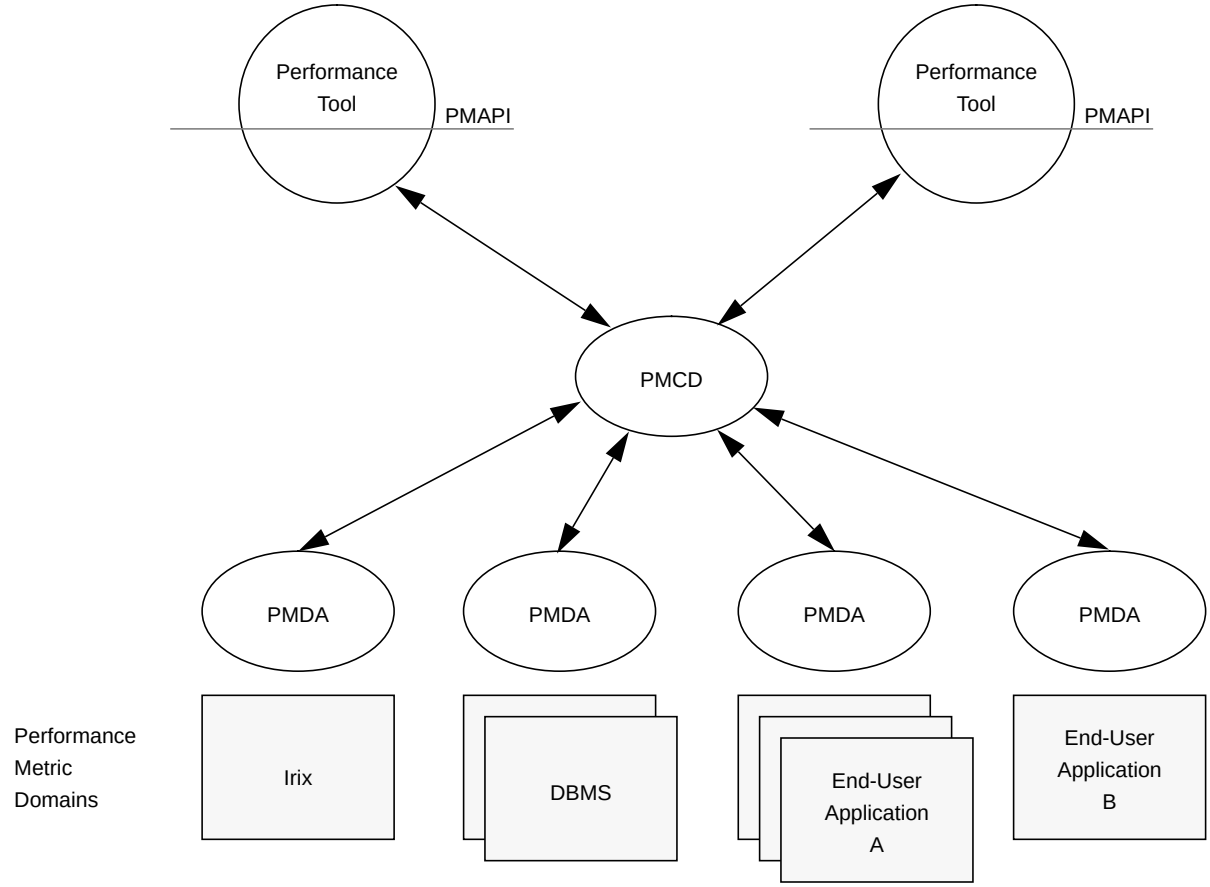


**Figure 4-2** PMDA Global Process Architecture

## Creating a PMDA

The following work is required to create a new PMDA.

- Determine what metrics are provided and how the metrics' values are obtained.

- Allocate data types and instance domains for the metrics.
- Assign *pmid*s to the metrics and add them to the PMNS.
- Create help text for the metrics.
- Write code to supply the metrics and associated information to PMCD.

## Domain Numbering Protocols for PMDA Metrics and Instance Domains

Every performance metric in the PCP framework is assigned a unique PMID. An initial task when adding new performance metrics is to assign PMID values.

The PMID is structured as follows:

```
/*
 * Internally, this is how to decode a PMID!
 */
typedef struct {
       int                pad:2;
       unsigned int       domain:8;
       unsigned int       cluster:12;
       unsigned int       item:10;
} _pmID_int;
```

If the new metrics are an extension of an existing PMDA implementation, then the domain is the same as for other metrics in that PMDA. If a new PMDA is being developed, the domain should be chosen to be unique amongst all PMDAs, using the table below as a guideline.

**Table 4-1** PMDA Domains

| Domain | Use |
|---|---|
| 0 | Reserved for internal use |
| 1 | IRIX kernel metrics |
| 2 | PMCD instrumentation and control |
| 3 | /proc metrics |

**Table 4-1** PMDA Domains

| Domain | Use |
|---|---|
| 4 | Challenge™/Onyx™ environmental monitor metrics |
| 5 | Cisco® router statistics |
| 6 to 31 | Reserved for other Silicon Graphics metrics |
| 32 to 39 | ORACLE instances |
| 40 to 47 | Sybase® instances |
| 48 to 55 | Informix® instances |
| 66 to 127 | ISV performance metrics |
| 128 to 254 | End-user application metrics |
| 255 | Reserved for internal use |

The remaining components of the PMID are used to guarantee uniqueness within a PMDA, and the division between cluster and item is entirely arbitrary. The set of assigned PMIDs within a PMDA may be as sparse or as dense as required; however, it is an implicit design rule that once assigned, a PMID is never re-assigned to another metric with different semantics. It is acceptable for the metric associated with a PMID to be no longer supported in a later PMDA release.

Refer to the control file */etc/pmcd.conf* for a list of the domain values for the PMDAs currently attached to the PMCD at the local host. The reference page for *pmcd(1)* describes the format of this file.

Once the PMIDs have been chosen for the new metrics, they need to be given external names so that end-user tools can identify the metrics symbolically. The PCP supports the notion of a hierarchic name space, used to name all metrics. There is a global (default) name space (the Performance Metrics Name Space, or PMNS), which spans all possible metrics over the site or sites of interest. Use the *pminfo* command to dump the current default name space, and choose appropriate names for the new metrics.

For new PMDAs, this may (but does not necessarily) require the addition of new non-terminal nodes in the name space; these may be added arbitrarily

as desired. For example, see Figure 4-3, where the new performance metrics are *network.packetrate.out*, *network.packetrate.in* and *cpu.interface_interrupts*.

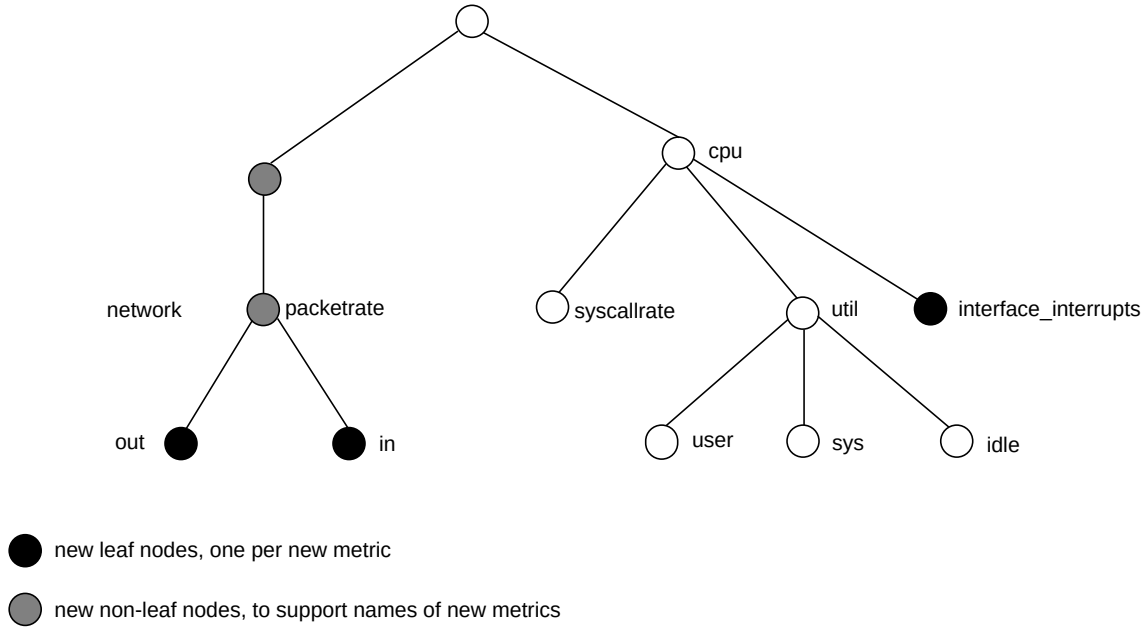


**Figure 4-3** Changes in a Small Section of the Performance Metrics Name Space

To permanently incorporate the association between the PMID and the default name in the PMNS, perform the following steps:

1. Consult “PMNS Management” on page 152, and then update the file or files required in the directory */usr/lib/pcp/pmns*.
2. Compile the name space with the commands:

```
cd /usr/lib/pcp/pmns
rm -f root.bin
../pmnscomp -f root
```

These changes to the PMNS need to be propagated (and perhaps merged with the existing PMNS) to each server where the new metrics are collected. The changes are also required on each workstation where PCP tools are

going to be run to monitor the new metrics. This includes workstations that examine archive logs containing the new metrics.

## Defining the Metadata That Describes the Performance Metrics

Correct processing of values for a performance metric in the PCP framework requires the person implementing of a performance metric (the PMDA) to export metadata that describes the structure and semantics of the metric. The relevant data structure is *pmDesc*. For complete information on *pmDesc*, see the file */usr/include/pcp/pmapi.h*, or "Performance Metric Descriptions" in Chapter 3.

Specifically, the PMDA must be able to provide the following for each metric it supports:

- The PMID.
- The basic (atomic) data type including format and size. For example, 32-bit integer, or 64-bit unsigned integer, or an aggregate of binary data with known length but unknown (within the PMCS) interpretation.
- The value should have a single instance. If not, what is the associated instance domain? (See "Creating and Maintaining Instance Domains" immediately below this list.)
- The following value semantics:
  - The most recent value for a free-running event counter (possibly with overflow).
  - An instantaneous value. (You cannot determine a "rate" from an instantaneous value, but interpolation may make sense.)
  - A discrete value that is instantaneous, and cannot be interpolated.
- Units and scale. For example, kilobytes per sec, or microseconds per event, or megabytes.

## Creating and Maintaining Instance Domains

Instance domains are sets of identifiers that are used to differentiate between multiple instances of the same metric. Some metrics are associated with the

special instance domain *PM_INDOM_NULL* and always have a single instance. All other instance domains are managed by a single PMDA. Multiple metrics may be associated with a single instance domain.

The PMDA is responsible for:

- Assigning the domain identifier for any instance domain it requires. This must be constructed to have the PMDA's domain identifier in the high-order (domain) part of the instance domain, but the low-order (serial) part is arbitrary, provided uniqueness is ensured. The PMDA may even assign different instance domain identifiers to the same instance domain over time, provided the assignment is fixed for any single execution of the PMDA.
- Determining which instances exist within the instance domain. For some instance domains this can be done at PMDA startup, and remains static thereafter. Other instance domains may require enumeration of the current instances each time the PMDA is requested to provide information about the instance domain.
- Assigning a unique internal value (an integer) and a corresponding external label (a string) that identifies the instance for each instance within an instance domain. The internal instance identifier *PM_IN_NULL* (defined in */usr/include/pcp/pmapi.h*) is reserved and may not be used.

The interface between the PMDA and the other elements of the PMCS requires the PMDA to export information about instance domains in response to specific requests; for example, to enumerate all internal and external identifiers, to map from an internal identifier to an external identifier, and vice versa.

## PMDA Help Text

For each metric defined within a PMDA, the PMDA developer is strongly encouraged to provide both one-line and extended help text to describe the metric, and perhaps provide hints about the expected value ranges.

The help text is prepared as ASCII files, processed with the *newhelp*(1) utility, and typically installed in the directory */usr/lib/pcp/help* on the system(s) where the PMDA executes.

Further details may be found in the *newhelp* reference page, and by consulting the *Makefile* and the file *help* in the example PMDA directory, */usr/demos/PerfCoPilot/pmdas/simple*.

## Building a PMDA

A PMDA interacts with PMCD across one of several well-defined interfaces and protocol mechanisms. These implementation options are described in "Performance Metrics Collection System" in Chapter 1.

It is strongly recommended that code for a new PMDA should be based on the source of one of the demonstration PMDAs in the */usr/demos/PerfCoPilot/pmdas* directory.

### The DSO Method

This method of building a PMDA uses a Dynamic Shared Object that is attached by PMCD, using *dlopen*, at initialization time. This is the highest performance option (there is no context switching and no IPC between the PMCD and the PMDA), but is operationally intractable in some situations. (For example, where special access permissions are required to read the instrumentation behind the performance metrics, or where the performance metrics are provided by an existing process with a different protocol interface.) The PMDA effectively executes as part of PMCD, so care is required when crafting a PMDA this way.

### The Daemon Process Method

Functionally, this method may be thought of as a DSO implementation with a standard wrapper to convert distributed PMCS messages into procedure calls. (See the file */usr/demos/PerfCoPilot/pmdas/sample/pmda.c*.)

### The Shell Process Method

This method offers the least performance, but may be well-suited for rapid prototyping of performance metrics, or for diagnostic metrics that are not going into production.

Implementation of the ASCII protocols is rather lengthy. The suggested approach is to take the */usr/demos/PerfCoPilot/pmdas/news/news.agent* PMDA as an illustrative example, and adapt it for the particular metrics of interest.

## New PMDA Integration With the PMCD

Once the PMDA has been implemented, it should be installed in */usr/lib/pcp* and its existence made known to PMCD by adding a new line to the *options* file */etc/pmcd.conf*. Restarting PMCD makes the metrics in the new PMDA immediately available. The reference page for *pmcd(1)* describes how to do this.

## Management of Evolution Within a PMDA

Natural evolution of PMDAs, or more particularly the underlying instrumentation that they provide access to, results in new metrics appearing and old metrics disappearing. This creates potential problems for both new and former versions of the PMDA.

The following guidelines are designed to reduce the complexity of PMDA implementation in the face of evolutionary change, while maintaining as much semantic coherence and ‘‘the law of least surprise’’ for tools using the PMAPI and for end-users of those tools.

- Try to support as full a range of metrics as possible in every version of the PMDA. In this context, “support” means to respond sensibly to requests, even if the underlying instrumentation is not available.
- If a metric is not supported in a particular version of the underlying instrumentation, the PMDA should respond to *pmLookupDesc* requests with a dummy *pmDesc* structure in which the type is the special value *PM_TYPE_NOSUPPORT*. The values of the fields other than *pmid* and *type* are immaterial, but the following example is typically benign:

```
pmDesc dummy = {
    0,                  /* pmid, fill this in */
    PM_TYPE_NOSUPPORT,  /* this is the important part */
    PM_INDOM_NULL,      /* singular,causes no problems */
    0,                  /* no semantics */
    { 0, 0, 0, 0, 0, 0 }  /* no units */
};
```

- If a metric is not supported in a particular version of the underlying instrumentation, the PMDA should respond to *pmFetch* requests with a *pmResult* in which no values are present for the unsupported metric. This is marginally friendlier than the other semantically acceptable option: “illegal PMID” or *PM_ERR_PMID*.
- The help text should be updated with annotation to describe the versions of the underlying product, or product configuration option, for which the metric is available. In this way *pmLookupText* always responds correctly.
- The *pmStore* command should fail with *PM_ERR_GENERIC* if one tries to amend an unsupported metric.
- The value extraction, conversion, and printing routines (*pmExtractValue*, *pmConvScale*, *pmAtomStr*, *pmTypeStr*, and *pmPrintValue*) returns errors or appropriate “Not Supported” strings if an attempt is made to operate on a value of type *PM_TYPE_NOSUPPORT*. If performance tools take note of the type in the *pmDesc* structure, they should not try to manipulate values for unsupported metrics. Even if the tools ignore the type in the metric’s description, following these development guidelines should ensure that no value is ever returned, so there is no reason to call the extraction, conversion, and printing routines.

## PMDA Samples

Samples of working PMDA software can be found in the */usr/demos/PerfCoPilot/pmdas* directory. The simple subdirectory contains a relatively simple example of a PMDA with an instance domain.

## PMDA Library Routines

The PCP framework provides run-time library support for many routines common to the implementation of the PMDAs. Use the source code of the PMDA samples to understand the functionality of these routines, and how they may be used.

*Chapter 5*

# Troubleshooting the Performance Co-Pilot

This chapter outlines the basic troubleshooting strategies for Performance Co-Pilot. Assorted issues with installation and operation of the Performance Co-Pilot pieces are presented. For each issue, there are sections describing symptoms, possible explanations for each symptom, and likely resolutions.

## Performance Metrics Application Programming Interface (PMAPI) Issues

The *PMAPI* (3) reference page documents many of the environmental, protocol, and style issues common to many of the PMAPI routines. The reference pages for the specific PMAPI routines are also a primary resource for troubleshooting.

Further information may be found in the sample client programs in */usr/demos/PerfCoPilot*.

### Slow PMCD Service

Symptom: A long pause sometimes occurs when attempting to connect to PMCD on another machine.

Cause: Some PMAPI routines that attempt to connect to a remote PMCD on a machine that is booting may block until the remote machine finishes its initialization.

Resolution: If PMCD_CONNECT_TIMEOUT is set in the environment to a real number of seconds and if the connection has not been established after the specified interval has elapsed, these routines abort and return an error.

PMCD_CONNECT_TIMEOUT may also be required to connect to a PMCD over a slow network connection, as discussed below.

## Performance Metrics Coordinating Daemon (PMCD) Issues

The following problems are related to the Performance Metrics Coordinating Daemon.

### PMCD isn't reconfiguring after a SIGHUP

Symptom The PMCD does not reconfigure itself after receiving the SIGHUP signal.

Cause: If there is an error in */etc/pmcd.conf*, PMCD does not use the contents of the file. This can lead to situations in which the configuration file and PMCD's internal state do not agree.

Resolution: Always monitor PMCD's log. For example, use

```
tail -f /var/tmp/pmcd.log
```

in another window when reconfiguring PMCD, so that any errors that occur during the process are visible.

### PMCD Does Not Start

Symptom: If the following messages appear in the system log (*/usr/adm/SYSLOG*), consider the cause and resolution below:

```
pcp[27020] Error: OpenRequestSocket(4321) bind:
Address already in use
```

```
pcp[27020] Error: pmcd is already running
```

```
pcp[27020] Error: pmcd not started due to
errors!
```

Cause: PMCD is already running or was terminated before it could clean up properly. The error occurs because the socket it advertises for clients to connect to is already being used or has not been cleared out by the kernel.

Resolution: Start PMCD as **root** (superuser) by typing:

```
/etc/init.d/pcp start
```

Any existing PMCD is shut down and a new one is started in such a way that the symptomatic message should not appear.

If you are starting PMCD this way and the symptomatic message appears, there has been a problem with the connection to one of the deceased PMCD's clients. This may happen when the network connection to a remote client's machine is lost and PMCD is subsequently terminated. The system may attempt to keep the socket open for a while to allow the remote client a chance to re-establish the connection and read any outstanding data. The only solution in these circumstances is to wait until the kernel times out the socket and deletes it. The command

```
netstat -a | grep 4321
```

displays the status of the socket and any connections. If the socket is in the FIN_WAIT or TIME_WAIT states, then you must wait for it to be deleted. Once the command above produces no output, PMCD may be restarted.

Less commonly, you may have another program running on your system that uses the same internet port number (4321) that PMCD uses. In the current version of the Performance Co-Pilot (PCP) software, this port number is hard-wired. You must either reconfigure the other program to use another port, or not use the program, if you want to use the PCP suite.

## Performance Metrics Name Space (PMNS) Issues

The following issues have to do with the matrix of performance metrics.

#### Performance Metrics Are Unknown

Symptom: Performance metrics are defined in the ASCII format of a Performance Metrics Name Space (PMNS), but attempts to use these metrics in the utilities (such as *pmval*, *pminfo*, and *pmchart*) produce errors notifying you of:

```
Unknown metric name
```

Cause: If you can, always use the binary format of the PMNS rather than the ASCII format. Changes made to the ASCII format that are not promulgated to the binary format remain "invisible."

Resolution: Recompile the PMNS. Try these commands:

```
cd <directory where PMNS is defined>
/usr/lib/pcp/pmnscomp -f -n root root.bin
```

## Missing and Incomplete Values for Performance Metrics

The following issues have to do with the information returned for various performance metrics.

### Metric Values Not Available

Symptom: Values for some or all of the instances of a performance metric are not available.

Cause: This can occur as a consequence of changes in the installation of modules (for example, a DBMS or an applications package) that provide the performance instrumentation underpinning the PMDAs. Changes in the selection of modules that are installed or operational, along with changes in the version of this modules, may make metrics appear and disappear over time.

For archive logs, the collection of metrics to be logged is a subset of the metrics available, so utilities playing from a log may not have access to all of the metrics available from a "live" (PMCD) source.

Resolution: Make sure the underlying instrumentation is available and the module is active. Ensure the PMDA is running on the host to be monitored. If necessary, create a new archive log with a wider range of metrics to be logged.

## Archive Logging Issues

The following issues have to do with the creation of logs using *pmlogger*.

### pmlogger Can’t Write Log

Symptom: The *pmlogger* utility does not start, and complains with a message of the form:

```
_pmLogNewFile: ‘‘foo.index'' already exists,
not over-written
```

Cause: Archive logs are considered sufficiently precious that *pmlogger* does not empty or overwrite an existing set of archive log files. The log “named" *foo* actually consists of the physical file *foo.index*, *foo.meta* and at least one file *foo.N*, where *N* is in the range 0, 1, 2, 3, ...

A message similar to the one above is produced when a new *pmlogger* instance encounters the first of these files already in existence.

Resolution: If you are sure, remove all of the parts of the archive log. For example, use the command:

```
rm -f foo.*
```

Then re-run *pmlogger*.

### Can’t Find Log

Symptom: The *pmdumplog* utility, or any tool that can read an archive log, complains with a message of the form:

```
Cannot open archive mylog: No such file or
directory
```

Cause: An archive consists of at least three physical files. If the base name for the archive is *mylog*, then the archive actually consists of the physical files *mylog.index*, *mylog.meta*, and at least one file *mylog.N*, where *N* is in the range 0, 1, 2, 3, ...

The above message is produced if one or more of the files is missing.

Resolution: Check which files the utility is trying to open, with the command:

```
ls mylog.*
```

Turn on the internal debug flag DBG_TRACE_LOG(**-D** 128) to see which files are being inspected by the routine *_pmOpenLog*.

Locate the missing files and move them all to the same directory, or remove all of the files that are part of the archive, and recreate the archive log.

## pmlogger Can't Start

Symptom: The primary *pmlogger* cannot be started. A message like the following appears:

```
pmlogger: there is already a primary pmlogger
running
```

Cause: There is either a primary *pmlogger* already running, or the previous primary *pmlogger* was terminated unexpectedly before it could perform its cleanup operations.

Resolution: If there is already a primary *pmlogger* running and you wish to replace it with a new *pmlogger*, use the *show* command in *pmlc(1)* to determine the process id of the primary *pmlogger*. The process id of the primary *pmlogger* appears in parentheses after the word "primary". Send an INT signal to the process to shut it down (use the *kill*(1) command). If the process does not exist, proceed to the manual cleanup described in the paragraph below. If the process did exist, it should now be possible to start the new *pmlogger*.

If *pmlc*'s *show* command displays a process id for a process that does not exist, a *pmlogger* process was terminated before it could clean up. If it was the primary *pmlogger*, the corresponding control files must be removed before one can start a new primary *pmlogger*. It is a good idea to clean up any spurious control files even if they aren't for the primary *pmlogger*. The control files are kept in */usr/tmp/pmlogger*. A control file with the process id of the *pmlogger* as its name is created when the *pmlogger* is started. In addition the primary *pmlogger* creates a symbolic link named *primary* to its control file. For the primary *pmlogger*, remove both the symbolic link and the file (corresponding to its process id) to which the link points. For other *pmlogger*s, remove just the process id file. Do not remove any other files in the directory. If the control file for an active *pmlogger* is removed, *pmlc* is not able to contact it.

## IRIX Metrics and PMCD

The following issues have to do with the interaction between IRIX and the PMCD.

### No IRIX Metrics Available

Symptom: Some of the IRIX metrics are unavailable.

Cause: PMCD and (therefore the IRIX PMDA) does not have permission to read */dev/kmem* or the kernel currently running is not the same as the kernel in */unix*.

Resolution: Check */usr/tmp/pmcd.log*. If there is an error message of the form:

```
kmeminit: cannot open "/dev/kmem": ...
```

PMCD cannot access */dev/kmem*. Ensure that */dev/kmem* is readable by group *sys* and that */usr/lib/pcp/pmcd* is installed setgid *sys*. Restart PMCD and the problem should be solved.

If the running kernel is not the same as the kernel in */unix*, the IRIX PMDA cannot access raw data in the kernel. A message of the form:

```
kmeminit: "/unix" is not the namelist for the
running kernel...
```

appears in */usr/tmp/pmcd.log*. The only resolution to this is to make the running kernel the same as the one in */unix*.

If the running kernel was booted from the file system, then renaming files to make */unix* the booted kernel and restarting PMCD will resolve the problem.

If the running kernel was booted over the network, then PMCD cannot access the kernel's symbol table and hence the metrics extracted by reading */dev/kmem* directly are not available.

## ORACLE Metrics and the ORACLE PMDA

Prior to installing any ORACLE PMDAs, read */usr/demos/PerfCoPilot/pmdas/oracle7/README*. It may prove necessary to run the *Install* or *Remove* scripts in that directory to fix problems, so it is a good idea to review the *README* again to refresh your memory before continuing.

There is one complete subtree of ORACLE metrics in the namespace for each ORACLE database instance for which you have installed an ORACLE PMDA. One of the first things you should find out is which numeric instance domain has been assigned to the ORACLE PMDA that is causing trouble. If the PMDA causing trouble is the one for the *xyz* database, list the first few metric identifiers for that PMDA by using the command:

```
pminfo -m oracle.xyz | head
```

A number of lines like the following appear:

```
oracle.xyz.all.logons PMID: 32.1.0
```

The first number in the dotted triplet is the domain number assigned to the PMDA. In this case it is 32.

There should be a line to start the PMDA in */etc/pmcd.conf*. If the ORACLE instance is called *xyz* and its domain number is 32, the first two things on the line are:

```
ora_xyz    32
```

If there is no line like that in the file, see the troubleshooting symptom immediately below. The domain number should also appear immediately after the **-d** option for the PMDA.

The log files for ORACLE PMDAs are */usr/tmp/oracle7-*.log* where the wildcard is replaced by the domain number of the PMDA. Thus the log file for an ORACLE PMDA with domain 32 would be */usr/tmp/oracle7-32.log*. Take care to check the date and time in the log files to ensure that you are not using an old log file to diagnose problems.

## PMDA Can't Connect to ORACLE

Symptom: The */usr/adm/SYSLOG* file contains the entry:

```
PMDA unable to connect to ORACLE (invalid
username/password; logon denied)
```

Cause: The ORACLE database user account has not been created for the PMDA or the user.

Resolution: When the script */usr/demos/PerfCoPilot/pmdas/oracle7/Install* was run to configure the PMDA, a file was created that contains the SQL statements that allow the PMDA to connect to the database and access the performance data. The file is in the same directory as the *Install* script. For example, if the ORACLE database instance was called *xyz*, the file is named *setup.xyz.sql*.

Use *sqldba* to connect to the same ORACLE database instance that the PMDA uses, and run the SQL commands in the file. Send PMCD a reconfiguration request like this:

```
killall -HUP pmcd
```

Check the log file for the PMDA again.

If the file creates an ops$... user, make sure that a corresponding UNIX user exists and that */usr/lib/pcp/pmdaoracle7* exists, is owned by that UNIX user, and has the setuid bit set. Become that UNIX user, set ORACLE_HOME and ORACLE_SID in the environment, and attempt a default login to the database:

```
sqlplus /
```

If instead the file created a normal ORACLE user with a password (for example, user *pcp* with password *meter*), log in to the database by typing:

```
sqlplus pcp/meter
```

In either case, ensure that you use the same ORACLE_HOME and ORACLE_SID as those specified for the PMDA in the log file. If either is wrong in the log file, you should alter them in the command line for the PMDA in */etc/pmcd.conf*.

## ORACLE Connection Errors

Symptom: */usr/adm/SYSLOG* file has errors of form:

```
Error connecting to ORACLE.  ORACLE not
available. smsgsg: shmget() failed...
```

Cause: This can be caused by a number of things, ranging from ORACLE being unavailable or misconfigured to incorrect parameters being specified when using the *Install* script to configure the PMDA.

Resolution: Make sure that the ORACLE database instance that the problematic PMDA utilizes is available. Connect to it using the command

```
sqlplus
```

and try to fetch some data using a database user other than that of the PMDA. A demonstration user like *scott* / *tiger* is ideal here.

If that works, connect to the database using the same ORACLE user that the PMDA does. Before doing so, you should make absolutely certain that the values of ORACLE_HOME and ORACLE_SID that appear in the PMDA's log file are correct, then set both in the environment. The ORACLE user and password for the PMDA appear on the command line for the PMDA in */etc/pmcd.conf* as the argument to the -c option. For example, if the flag **-c pcp/pcp** appears on the command line, you should enter the command:

```
sqlplus pcp/pcp
```

If there is no **-c** option, the PMDA is using an ops$... logon and */usr/lib/pcp/pmdaoracle7* will be setuid. In this case you should become the UNIX user that owns the *pmdaoracle7* file (using *su* or an equivalent command) and use a default login to sqlplus:

```
sqlplus /
```

Remember to check that ORACLE_HOME and ORACLE_SID are correct prior to doing this. If *sqlplus* says that the login is not permitted because the username or password is invalid, see the section titled “PMDA Can’t Connect to ORACLE”.

If *sqlplus* doesn't let you in, there may be a problem with your ORACLE database configuration. In particular, ORACLE does not handle NFS-mounted ORACLE_HOME directories well, because write permission is required to update control and or log files in $ORACLE_HOME/dbs.

Once you are in *sqlplus*, type the command:

```
describe v$sysstat
```

A description of the v$sysstat view should appear. If it does not you should re-run the script to grant the PMDA access to the performance data from the database (see the section titled “PMDA Can’t Connect to ORACLE”).

Restart PMCD by becoming **root** and typing:

```
/etc/init.d/pcp start
```

Inspect the log file for the PMDA. If it started successfully this time, you may need to alter your database startup routine. See the section below regarding ORACLE metrics being unavailable when the database or machine is restarted.

## Can’t Find ORACLE Metrics

Symptom: After the machine is rebooted, or after the ORACLE database is shut down and then restarted, it is not possible to get any ORACLE metrics.

Cause: If a PMDA is unable to connect to its ORACLE database instance, it exits. If the PMDA has a connection to the database and the database is shut down, the PMDA does not notice until the next request for ORACLE metrics arrives. This is because a PMDA accesses the ORACLE database only when a request is made. Thus, if a PMDA is

idle while its database is shut down and then brought back up again, the next request to it fails because it is trying to use an old connection to the database.

Resolution: As part of the startup procedure for a database, once the database has started you should use *pminfo* **-f** to fetch a single ORACLE metric from the database that has restarted. Then, as **root**, enter this command:

```
killall -HUP pmcd
```

This tells PMCD to restart any deceased PMDAs. The fetch ensures that the ORACLE PMDA for the database tries to exercise its now-defunct connection and terminates as a result; this makes PMCD restart with a fresh connection to the database.

## General Utilities Issues

The following issues are more general.

### Can't Connect to Remote PMCD

Symptom: A PCP client tool (such as *pmchart*, *dkvis*, or *pmlogger*) complains that it is unable to connect to a remote PMCD (or establish a PMAPI context), but you are sure that PMCD is active on the remote host.

Cause: To avoid hanging applications for the duration of TCP timeouts, the PMAPI library implements its own timeout when trying to establish a connection to a PMCD. If the connection to the host is over a slow network, then successful establishment of the connection may not be possible before the timeout, and the attempt is abandoned.

Resolution: Establish that the PMCD on *far-away-host* is really alive, by connecting to its control port (TCP port number 4321):

**telnet** *far-away-host* **4321**

If the response is

```
Unable to connect to remote host: Connection
refused
```

then PMCD is not running and needs to be restarted on that host. Enter the command:

```
/etc/init.d/pcp start
```

If the response is

```
Connected to far-away-host
```

then PMCD is alive and well. Interrupt the *telnet* session. Increase the PMAPI timeout by setting the environment variable PMCD_CONNECT_TIMEOUT to some large number of seconds (for example, 60) and try the PCP tool again.

## Changing pmchart Colors

Symptom: When using the Indy Presenter™, or making presentations to a large group, the default pastel color scheme used by *pmchart* may be inappropriate.

Cause: These are the default colors for *pmchart*.

Resolution: Override the defaults using the X11 resources that *pmchart* honors. For example, create or add the following entries in the file *$HOME/.xrdb*:

```
PmChart*xrtForegroundColor: "green"
PmChart*xrtBackgroundColor: "black"

PmChart*xrtGraphForegroundColor: "rgb:00/b0/00"
PmChart*xrtGraphBackgroundColor: "black"

PmChart*xrtHeaderForegroundColor: "green"
PmChart*xrtHeaderBackgroundColor: "black"

PmChart*pmDefaultColors: rgb:ff/ff/00 rgb:00/ff/
00 rgb:00/00/ff \
                         rgb:ff/ff/00 rgb:00/ff/
ff rgb:ff/00/ff
```

Now use the command:

```
xrdb -merge $HOME/.xrdb
```

This changes the default color scheme for *pmchart* to one with bright primary colors on a black background.

*Chapter 6*

# Glossary of Acronyms

This chapter provides a glossary of the acronyms used in the Performance Co-Pilot documentation, help cards, reference pages, and user interface.

**Table 6-1** Performance Co-Pilot Acronyms and Their Meanings

| Acronym | Meaning |
|---|---|
| DSO | Dynamic Shared Object |
| IP | Internet Protocol |
| PCP | Performance Co-Pilot |
| PDU | Protocol Data Unit |
| PMAPI | Performance Metrics Application Programming Interface |
| PMCD | Performance Metrics Coordination Daemon |
| PMCS | Performance Metrics Collection Subsystem |
| PMD | Performance Metrics Domain |
| PMDA | Performance Metrics Domain Agent |
| PMID | Performance Metric Identifier |
| PMNS | Performance Metrics Name Space |
| TCP | Internet Transmission Control Protocol |

# Index

## R



## S



## T



## U



## V



## W