

CALIFORNIA POLYTECHNIC STATE UNIVERSITY
San Luis Obispo

CAMP: A Common API for Measuring Performance

A Thesis submitted in partial satisfaction of the
requirements for the degree

Master of Science

in

Computer Science

by

Mark Gabel

Committee in charge:

Professor Michael Haungs, Chair

Professor Aaron Keen

Professor Mei-Ling Liu

June 2006

CAMP: A Common API for Measuring Performance

Copyright © 2006

by

Mark Gabel

I grant permission for the reproduction of this thesis in its entirety or any of its parts, without further authorization from me.

The thesis of Mark Gabel,
“CAMP: A Common API for Measuring Performance”,
is approved.

Professor Aaron Keen

Professor Mei-Ling Liu

Professor Michael Haungs, Committee Chair

March 2006

Abstract

CAMP: A Common API for Measuring Performance

by

Mark Gabel

Accurate performance testing of heterogeneous distributed systems, such as those created using GRID technology, requires a consistent method for retrieving system performance data from multiple platforms. This thesis presents CAMP: a low-level platform independent performance data API designed for use with distributed testing frameworks.

CAMP is not necessarily tied to the distributed testing task: it provides a simple, low-level interface into operating system performance data that can be used to build complex performance measurement applications. This thesis discusses CAMP's functionality and implementation in detail. It also contains a detailed analysis of the API's correctness, performance, and overhead.

Acknowledgements

I would like to thank my committee members for showing a genuine interest in my work and accommodating my atypical schedule.

Special thanks goes to Dr. Michael Haungs, who endured countless hours of the results of my tendency to think out loud. His feedback, ideas, and incredible responsiveness to my scheduling needs over the past four months made this work possible.

Contents

Contents	vi
List of Tables	viii
List of Figures	ix
1 Introduction	1
2 Intended Applications	5
2.1 Derived Functions	5
2.2 System Monitoring	6
2.3 Distributed System Testing	7
2.4 Integration With GRID Frameworks	8
3 Design and Architecture	9
3.1 Completeness and Correctness	9
3.2 Platform Independence	11
3.3 Generalizing the Solution	13
3.4 Performance	15
4 Implementation	17
4.1 Python	17
4.2 Windows	19
4.3 Linux	20
4.4 Solaris	22

4.5	Implementation Issues	23
4.5.1	Raw Data on Windows	24
4.5.2	Precision and Consistency	25
4.5.3	Determining CPU Usage	26
4.5.4	Inquiry Functions	27
4.5.5	Process Addressing	28
5	Results and Analysis	31
5.1	Verification	31
5.1.1	Unit Testing	32
5.1.2	Result Validation	33
5.1.3	Informal Verification	41
5.2	Performance Analysis	42
5.2.1	Timing and Profiling	43
5.2.2	Measured Overhead, or CAMP on CAMP	47
5.2.3	Impact on an Operational System	48
6	Related Work	55
6.1	PAPI and Derivatives	55
6.2	Distributed System Monitoring	57
6.3	Distributed Performance Testing	60
6.4	Benchmarking	65
6.5	Summary	66
7	Future Work	67
7.1	Performance Optimizations	67
7.2	Continued Development	69
8	Conclusions	71
	Bibliography	73

List of Tables

3.1	The final CAMP API.	12
3.2	The CAMP system architecture.	14
5.1	Memory allocation results.	37
5.2	Network test results.	39
5.3	Thread count test results.	40
5.4	Practical maximum function call rates.	48

List of Figures

2.1	A derived function that measures network throughput.	6
2.2	A CAMP-based monitoring system.	7
5.1	The CPU load generation code.	35
5.2	CPU time verification results.	36
5.3	Execution time results.	44
5.4	Relative execution time results.	45
5.5	Operational impact at increasing sample rates.	49
5.6	Results of the operational system impact test.	51
5.7	Results of the system impact test at a critical system load.	53

Chapter 1

Introduction

Performance testing is a critical part of the distributed system development cycle, and there is a clear need for robust, automated, and reusable testing mechanisms. This task is made difficult by several factors, and each must be individually addressed and resolved for a system to be generally applicable to an appreciable portion of the set of distributed systems.

Test beds often consist of heterogeneous systems composed of different platforms and capabilities. For purpose-built systems, this can be intentional. However, for some system developers, this type of test bed may be the only resource available. For GRID-based systems, this is inevitable.

With the increasing popularity of Java and other virtual machine or interpreter driven languages, application code is often made “incidentally portable;” that is, the final system does not necessarily need cross-platform capabilities, but it gains that ability through the features of the host language. This provides a unique opportunity for distributed system developers to test code on expanded sets of test beds, including the previously mentioned heterogeneous

systems. For developers to exploit this advantage, a performance testing framework must be able to operate seamlessly and transparently on several platforms, and it must not provide a burden to developers who are developing otherwise platform-independent code.

Aside from platform independence, a testing framework should be able to measure data that is both relevant and meaningful. Unfortunately, distributed metrics are difficult to generalize. For example, developers of distributed agent systems might be concerned with end-to-end processing time of requests, while a developer of a distributed computation system might be interested with the efficient and total utilization of available network resources. However, certain metrics do lend themselves to standardization: system performance counters on individual nodes. A platform-independent method of accessing performance data on individual nodes could serve to supplement or even build larger, domain-specific metrics.

However, the source of system performance data varies. Some solutions make use of highly accurate hardware performance counters, which are dependent on the underlying architecture of the tested systems. In addition, the interfaces to these hardware systems usually require extension of the host operating system, making the framework doubly dependent on both the operating system and the underlying architecture. While this provides very accurate data for specific systems on a system-wide level, it is difficult to extrapolate isolated data for a single process.

At a higher level, modern operating systems provide interfaces to performance data, both on the system-wide and per-process levels. These interfaces can take the form of a well-defined API, a high-level system of performance counting, or simply exposed kernel structures that contain performance information. Operat-

ing system performance interfaces often make use of hardware performance counters as well, further strengthening their accuracy. In addition, several operating system-dependent metrics are of use to the distributed system developer. These can include virtual memory usage, page fault counting, and network interface statistics.

This thesis introduces CAMP: a cross-platform low-level API for measuring system performance, designed for use with distributed testing frameworks. CAMP's research is grounded on a single proposition: modern operating systems function similarly, and they keep track of their performance data in some externally-accessible way. CAMP standardizes access to this data through a cross-platform interface, fully encapsulating the available operating system performance reporting services. It is implemented across three major platforms in Python, a platform-independent interpreted language. The API and its respective semantics are identical across each platform. While not a testing framework in itself, CAMP makes a valuable contribution to the task by solving the system data independence problem at the lowest level.

CAMP provides a common entry point and retrieval format for system-wide and per-process performance data. It is implemented completely statically, holding no persistent state. It attempts to provide raw, unprocessed data wherever possible. The implementation is substantial, production-ready, and uses the highest-performing, lowest-overhead method available on each platform to provide data. CAMP provides novel solutions to several implementation-specific problems. These include addressing a process consistently across multiple operating systems, encapsulating time-averaged data in a single, stateless function call, and addressing locally-named devices in a platform independent manner.

Chapter 2 provides a more global view of CAMP through several possible

usage scenarios. Chapter 3 describes CAMP's global and specific design goals, system architecture, and realized design. Chapter 4 discusses the technical implementation across three platforms. It also provides an overview of general implementation issues and their respective solutions. Chapter 5 contains a detailed analysis of CAMP in the areas of correctness, consistency, performance, and overhead. Chapter 6 compares CAMP to other solutions in the performance testing field, and Chapter 7 details a plan for continuing research.

Chapter 2

Intended Applications

CAMP is a versatile interface: it aims to be the foundation of a performance measurement system and can be used as is or easily extended. Its interface is capable of being used in a variety of different scenarios.

2.1 Derived Functions

The data provided by CAMP is raw in nature; that is, it has no meaningful calculation performed on it. For example, network connections are measured by total numbers of packets sent and received rather than a rate. CAMP's ability to provide this low-level data across multiple platforms allows developers to create platform independent secondary functions.

For example, a developer needing rate or throughput data for network connections is free to implement the function in any way: her or she can choose how the state is stored, what statistical functions will be used to determine the rate, and how often the system will be polled. The developer is guaranteed reliable

```

def BulkByteTransferRate(adapter):
    initial = GetNetBytesSent(adapter)
    initial += GetNetBytesRecvd(adapter)
    time.sleep(SAMPLE_INTERVAL)
    final = GetNetBytesSent(adapter)
    final += GetNetBytesRecvd(adapter)
    return (final - initial)/SAMPLE_INTERVAL

```

Figure 2.1: A derived function that measures network throughput.

and consistent raw data from the CAMP interface. This extension would not be tied to a specific platform. By selectively determining what extra processing and state are used, the developer minimizes the overhead of the performance measuring system. A sample rate function appears in figure 2.1.

2.2 System Monitoring

CAMP could also be used in a wide scale client-server monitoring system. The API's versatility and consistency would allow implementation of a variety of cross-platform monitoring systems. System analysts might be interested in passive performance logging systems, where CAMP's data could be used to make decisions about expanding capacity or more efficient allocation of resources. System administrators could find derived functionality like active throughput and capacity meters useful: spikes or drops in measured data could quickly indicate problems.

Using CAMP, all of these systems could be built with little knowledge of the inner workings of the tested systems' kernels. The simplistic model and implementation in a clear, scripting-like language allows non-developers to intuitively collect desired performance data. A sample architecture for a CAMP-based monitoring system appears in figure 2.2.

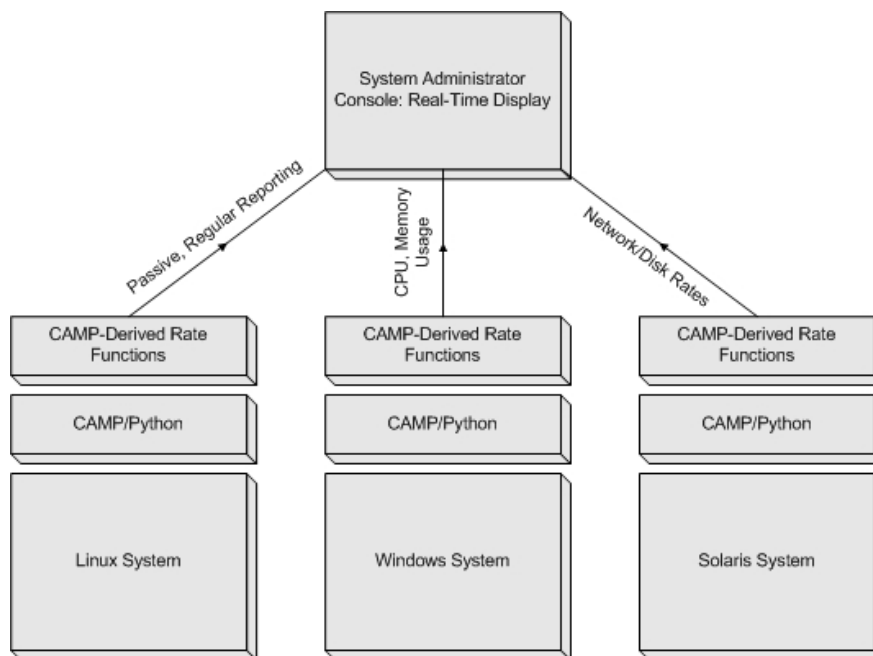


Figure 2.2: A CAMP-based monitoring system. A platform independent agent on each node passively reports raw and derived data to a system administrator’s console.

2.3 Distributed System Testing

Accurate operating system performance data would be a valuable addition to an existing distributed testing framework, and CAMP would be able to provide this functionality with minimal overhead. Many existing testing frameworks rely on the distribution of timestamped, domain-specific performance data (see chapter 6). In this scenario, timestamped data from CAMP or CAMP-derived functions could be collected along with the existing data and correlated by time. This could provide powerful insight for the diagnosis of performance problems observed on a global level: developers could precisely quantify their software’s effect on various operating system services during periods of poor performance.

Other systems concern themselves with treating distributed systems as black

boxes. These systems test only externally measurable values like global latency and throughput of provided services. CAMP provides a low overhead, non-intrusive way of supplementing this domain-specific data with system performance data. The API allows developers to maintain this black box model by utilizing the available operating systems to probe processes externally rather than instrumenting the running code.

2.4 Integration With GRID Frameworks

GRID frameworks must handle the allocation of resources in large pools of heterogeneous systems. This software, in essence, performs operating system services on a much larger scale. CAMP's universal view of fine-grained operating system performance data could aid in this process. GRID software could probe the current load and capacity of both idle and utilized systems.

The gathered data would serve as a basis for decision making. Examples of allocation-related decisions include the initial allocation of loads to nodes in the GRID, the migration of loads away from heavily loaded nodes, and the selective utilization of nodes based on their individual abilities to cope with domain-specific requirements, like high network throughput or CPU capacity.

Chapter 3

Design and Architecture

Chapter 1 presented an overall outline of CAMP’s design goals. This chapter expands on that outline by discussing specific design objectives and their respective design alternatives. Each specific section presents:

- A description of the design goal and a justification for its inclusion in CAMP.
- A set of realized design alternatives.
- CAMP’s realized design, with justification as to how it best meets the design goal.

This chapter also presents the current API in its entirety.

3.1 Completeness and Correctness

For CAMP to be complete, it should be able to provide substantial coverage of each implemented platform’s performance statistics. At the minimum, CAMP

should be able to retrieve global resource usage statistics for the measured system’s CPU(s), memory, network interfaces, and disk IO subsystem. Additionally, the API should minimally provide CPU and memory usage for individual processes. This presents several challenges: some platforms provide more information than others, while some platforms simply provide different information or different levels of granularity.

Three design alternatives exist that could counter these problems. The most straightforward approach is to take the intersection of the functionality available on each platform. This approach, described in [8], is taken by the Java Abstract Windowing Toolkit (AWT) [40]. The Java AWT takes the lowest common denominator set of natively-available GUI components on several platforms and generalizes them under a single interface. While effective, this method does limit the functionality to that of the least comprehensive platform. However, it is guaranteed to be fully consistent in any implementation.

A second alternative involves widening the set of available features to that of one of the more endowed platforms and *emulating* the missing functionality on the others. This method allows the core functionality to remain identical and allows a best-effort implementation of other functionality, but it may induce frustration: functions may not behave correctly or return consistently correct data. It does, however, provide an expanded feature set while maintaining a consistent interface.

The final alternative is employed by Python’s own “operating system services” (`os`) package. Each platform’s full feature set is exposed, and standardization is done on a best-effort basis. For example, the `os` package attempts to standardize UNIX commands like `chmod` and `stat` by providing Windows implementations, but importing the package on a Windows platform causes all functions relating

to symbolic links to disappear. This design provides much more functionality than the previous two, but any use of the optional services renders the code non-portable.

CAMP makes use of the lowest common denominator design. Consistency and transparency of the API is one of CAMP's most important design goals, and using the intersection of available functionality guarantees that this goal is met. The second alternative, emulated functionality, may be a candidate for future work, but at present it appears that the deficient platforms do not have any obvious ways to emulate missing data. In addition, the emulation model implies a loss of precision or correctness, which does not fit well with the CAMP model. CAMP functions are designed to be the building blocks of higher-order systems, and it is probable that even minor inconsistencies would be amplified in derived functions. The third alternative, while providing the greatest level of completeness, contradicts CAMP's efforts to be common.

While the intersection design has the potential to severely limit functionality, CAMP is still able to provide a comprehensive API. As discussed in chapter 1, CAMP's research is based on the postulation that modern operating systems function similarly and record similar data. While this data may be difficult to access, the common set of accessible data was more than enough to provide a usable API. The final API appears in table 3.1.

3.2 Platform Independence

CAMP's functions should not be tied to any one platform. Platform independence may seem implied and basal, but two factors provide a multitude of design alternatives.

Global Functions	
<code>GetPercentProcessorTime(cpu)</code>	Global CPU usage. CPU parameter may be omitted.
<code>GetFreePhysicalMemory()</code>	Global free physical memory
Network Functions	
<code>GetNetBytesSent(interface)</code>	Total bytes sent on interface
<code>GetNetPacketsSent(interface)</code>	Total packets sent on interface
<code>GetNetBytesRecv(interface)</code>	Total bytes received on interface
<code>GetNetPacketsRecv(interface)</code>	Total packets received on interface
Disk IO	
<code>GetNumReads_Disk(disk)</code>	Number of read operations on the given disk
<code>GetNumWrites_Disk(disk)</code>	Number of write operations on the given disk
<code>GetNumReads_Partition(partition)</code>	Number of read operations on the given partition
<code>GetNumWrites_Partition(partition)</code>	Number of write operations on the given partition
Per-process Functions	
<code>GetNumPageFaults(process)</code>	Number of major and minor page faults
<code>GetCPUTime_Total(process)</code>	Process CPU utilization
<code>GetCPUTime_User(process)</code>	Process user mode CPU utilization
<code>GetCPUTime_Kernel(process)</code>	Process privileged mode CPU utilization
<code>GetWorkingSetKb(process)</code>	Size of the process working set in KB
<code>GetVMSizeKb(process)</code>	Size of the used virtual address space in KB
<code>GetThreadCount(process)</code>	Number of threads contained in this process
Inquiry Functions	
<code>EnumDiskPartitions()</code>	Enumerates the available disk partitions
<code>EnumPhysicalDisks()</code>	Enumerates the available physical disks
<code>EnumNetworkInterfaces()</code>	Enumerates the valid inputs to the network functions
<code>GetCPUCount()</code>	Returns the number of CPUs in the system
<code>GetProcessIdentifier(pid)</code>	Returns a “process identifier” for the given pid
<code>GetProcessIdentifiers(name)</code>	Returns a “process identifier” for each running process launched from an executable of the given name
<i>All CPU-time functions have an optional second parameter: the sample interval.</i>	

Table 3.1: The final CAMP API.

Platform independence is provided through features of the host language, and it can manifest itself at either compile-time or runtime. Source code that compiles properly on multiple platforms is often referred to as “portable,” while compiled code that runs correctly on multiple platforms through a virtual host is referred to as “Write-Once, Run Anywhere (WORA).” As discussed in chapter 1, CAMP’s target audience includes developers of “incidentally portable” systems. Developers of such systems need not have knowledge of native programming on the target platforms. Because of this, CAMP is built on a WORA platform:

Python, an interpreted language. This allows rapid development of tests with a minimized learning curve as well as easing deployment: once distributed, CAMP-based code can be immediately run without first being compiled.

The second factor involves the modification of the underlying platforms. The addition of constantly-running user mode services would allow additional data collection, and the augmentation of existing kernels allows nearly limitless possibilities for the addition of statistics. However, for CAMP to be truly common, it must not require the modification of the measured host in any way. Required modification of the kernel is next to impossible on Windows, and it breaks the ease-of-use afforded by a stateless, common interface that would otherwise require no setup¹.

3.3 Generalizing the Solution

The solution to cross-platform performance measurement should be generally applicable to many problem domains. It should not necessarily be tied to the distributed computing field. CAMP's abstraction level could have been implemented at several levels: a standardized call/response architecture for performance data, similar to SNMP [5], or several lower level, non-network models. One model, following from Microsoft's design, involves an object-oriented approach. The user registers a list of relevant performance statistics with the operating system and actively requests that data be collected and returned. This approach is unnecessarily complicated and involves a constant memory footprint for data that may only be collected occasionally.

Another model involves the standardization of BSD-style process account-

¹In other words, required kernel modification would alienate most of CAMP's user base.

Performance measuring applications, testing frameworks			
Possible CAMP extensions: derived functions, stateful counters			
CAMP			
Low-level, stateless performance functions			
Linux(proc)	Win32(pdh)	Solaris(kstat)	Other

Table 3.2: The CAMP system architecture.

ing. In this model, the operating system collects specific and fine-grained process performance data and makes it available as a summary report after a process terminates. This design has the advantage that it is already implemented on several UNIX platforms, but it removes an important attribute from the performance data: the temporal variable. Performance testing involves more than just summarizing performance data—It comprises the linking of performance data to specific events, correlated by time. This solution also rules out CAMP’s use in any active-monitoring scenario.

CAMP’s design uses a simple, low-level stateless interface. To the user, all performance data can be thought of as existing in a set of permanent counters, and the API merely provides simple accessors to the data. This simplistic view of the data allows the architecture shown in table 3.2, which is described in the previous chapter. The API is a set of solid, well-tested building blocks that form the foundation of a platform-independent performance monitoring application.

The design is largely inspired by declarative languages like SQL. The data is assumed to exist in a changing but directly inaccessible state, and the methods for accessing the data are thread-safe and consistent for a single point in time. CAMP focuses on providing the simplest, most intuitive interface without prematurely optimizing it by forcing it into the mold that best suits the performance data

access patterns of a particular platform. As discussed in chapter 5, this does take a toll on performance in some cases. The simple, atomic request-response architecture allows a focus on correctness and usability: CAMP defines distinct function contracts, and its proper use is not defined by a set of unnecessarily complex access patterns.

3.4 Performance

CAMP functions should incur as little intrinsic overhead as possible. This involves setting a specific performance goal and using that as a benchmark for optimizations.

The performance goal is as follows:

The implemented solution should be able to deliver most of its information at a one second quantum without causing a significant amount of overhead.

To clarify, *most of its information* is a general term that describes the hypothetical set of information that interests a particular distributed test. As discussed later in chapter 5, CAMP’s test plan uses a set of nine performance statistics to compose that set. This test plan uses the hypothetical “set” of performance data to accomplish two goals: it tests that CAMP is usable in a typical monitoring scenario, and more importantly, it tests that the stateless design discussed in the previous section does not introduce a substantial amount of overhead.

Designing CAMP to be “as fast as possible” involves the elimination of some design alternatives—namely, the ability to leverage preexisting native utilities to shorten CAMP’s development time. Most² of CAMP’s functionality can be

²But definitely not all.

collected from the output of a native, command line driven utility on the host. However, this is a potentially costly layer of indirection: the forking of a process to satisfy a single function call, which may be called several times per second, puts an unnecessary strain on operating system resources and is very time consuming.

This overhead may be acceptable for simple system monitoring tasks, as demonstrated by Eddie [27], but this level of resource consumption is not satisfactory for a high-performance API that intends to form a foundation for performance measuring applications. Instead, CAMP makes use of the fastest, most direct native interfaces into the kernel performance data for each implemented platform. It does not make use of any other utilities or services.

Chapter 4

Implementation

CAMP is implemented in the Python language. It currently supports the Win32 (Windows NT/2000/XP), Linux 2.6, and Solaris (SunOS kernel 5.x) platforms. The Win32 implementation interfaces to Microsoft’s performance data handler (PDH) interface. In the Linux implementation, CAMP interfaces with the `proc` filesystem. On Solaris, CAMP uses both the `proc` filesystem and the kernel statistics chain (`kstat`).

This chapter discusses each platform’s implementation in detail, and it also chronicles several global issues that affected all platforms.

4.1 Python

Python is an open-source, interpreted language that is available for nearly every modern platform. It is praised for its unusually clear syntax and strong set of libraries. Python lacks end-of-statement delimiters, forced declaration of variables, and explicit types. It also uses the concept of “significant white

space,” with indentation actually indicating the nest level of a particular statement. These features lend to Python code’s readability. Python combines the simplistic syntax and excellent text processing capabilities of a scripting language with the robustness and maturity of a full, heavyweight programming language.

CAMP uses Python for a number of reasons. Python has a fully wrapped interface to the Win32 interface using the `pywin32` package. This included a wrapper around Microsoft’s performance data handler interface, which was able to provide nearly all performance measuring functionality in the Win32 implementation. Python is also ideal for text processing tasks. Containing a full regular expression matching and replacement implementation, Python lessened the difficulty of parsing the cryptic output of the `proc` filesystem.

Python’s language structure eased the development of a platform-aware `import` statement. In Python, code not contained within a class is executed at the time of import. In the main CAMP interface file, the current platform is determined using the `sys` built-in package. Upon successful discovery, CAMP assigns into common variable names the appropriately implemented functions. These variables then become the publicly accessible CAMP functions. In a traditional programming language, this could have been a relatively complex task. For example, an object-oriented language could have leveraged dynamic dispatch from a common base class, but this would have involved a layer of indirection on each function call. In Python, functions are indistinguishable from any other type of variable, so the entire encapsulation of the API was accomplished with a conditional and a series of simple assignments.

As an object-oriented language, Python fully supports exceptions. If a developer attempts to import the CAMP package on an unsupported platform, the import is rolled back and an appropriate exception is raised. In addition, descrip-

tive exceptions represent error conditions in the various performance functions.

4.2 Windows

Microsoft does provide a low-level interface to performance information: the `HKEY_PERFORMANCE_DATA` registry branch. However, it is not visible to the user and requires direct programmatic interaction with the registry hive. When using the registry directly, access is not error-checked or thread safe. Incorrect use may provide false data rather than an exception or error. Instead of accessing this data directly, Microsoft recommends the use of the performance data handler interface.

The Win32 performance data handler interface is a high-level encapsulation around the concept of performance data gathering. It revolves around the concept of a “counter,” which is an object that is attached to particular performance “concept” and can be called to periodically gather data about that concept. To retrieve data, the user calls either a raw or formatted data retrieval function on the counter. The Windows “Performance” control panel administrative tool demonstrates the direct use of this interface.

Windows provides counters for performance “objects.” Examples of objects include “Memory,” “Processor,” “Paging File,” and “Physical Disk.” Microsoft allows outside applications to add their own performance objects and allows them to be accessed through the same performance data handler interface.

Each object is associated with one or more “counters.” For example, the “Memory” performance object contains around twenty counters, including instances such as “Pages/sec,” “Available Bytes,” and “% Committed Bytes in

Use.” Some counters are associated with “formatted” data, which involves a computation on raw data. “Pages/sec” is an example of one of these: the counter itself contains a raw count of the number of pages written to and read from disk, but retrieving the counter value causes an average rate to be calculated. One can bypass this calculation by retrieving the “raw” performance data from a counter. Other, scalar counters like “Available Bytes” return the same value for both formatted and raw outputs. CAMP almost exclusively uses the raw data, which may have its ultimate source in the NT kernel (in the case of process information) or in an individual device driver (in the case of network or disk IO).

A counter can optionally be associated with an “instance.” For the counters in the object “Process,” the set of instances consists of the set of running processes. For the counter “Network Interface,” the set of instances consists of the set of installed network adapters. Microsoft provides a function to enumerate the set of instances for a given counter, which CAMP uses to implement each of the enumeration functions.

The Microsoft performance data handler interface is designed for higher-level usage than that provided by CAMP, but there appears to be little overhead in using its clear, relatively simple interface. A more thorough analysis appears in chapter 5.

4.3 Linux

On the Linux platform, CAMP collects operating system performance data through the `proc` pseudo-filesystem. `proc` is a file-like interface to kernel data structures that show system information, which includes performance data. At the top level, the filesystem contains three types of entries: status files, kernel-

specific directories, and process directories¹.

The `proc` filesystem is called a pseudo-filesystem because the “files” presented are actually file-like interfaces into kernel data structures which reside completely in memory. Access to the filesystem is exceptionally fast and is designed for frequent use. Several files are redundant: human-readable versions are supplemented by simple, one-line white space delimited versions that can be parsed more quickly.

The status files contain useful information about the system configuration: information about the CPU, mounted filesystem, attached devices, and memory. It also contains an accurate uptime count, which lists the current system uptime and how much of that time has been spent in the idle process.

The kernel-specific directories are groupings of status files. For example, the `net` directory contains information about each protocol in use as well as raw performance data for each network adapter. In addition, when granted root-level access, several of these files are writable. Modifying one of these “files” causes the modification of the related kernel data structure.

CAMP uses these kernel-specific files for implementing all global functions. In many cases, the data provided by `proc` is already in a sufficient format. In some cases, however, the data requires manipulation. For example, most memory measurements in Linux are stored as page counts. CAMP converts these to a raw size by adjusting the value based on the results of the POSIX `getpagesize` call.

The process directories contain useful information about each running process. They are named by the process identification number (pid) of the related process. A process directory contains a symbolic link to the executable that spawned the

¹Note that this breaks from the traditional UNIX model of only keeping process information in `proc`. This greatly aided CAMP’s development on Linux.

running process, information about memory and CPU usage, a list of open file descriptors, and varied information about the process's environment.

As a safety precaution, CAMP reads the entire contents of the relevant files in a single unbuffered read. This does not cause any performance penalty: the data already exists in main memory. CAMP does this because `proc` only guarantees read consistency on a single read call—not over the entire duration that the file is open. Otherwise, CAMP could deal with very subtle race conditions. For example, most data in the `proc` filesystem is encoded as ASCII text, with each value comprised of several unique digits. In a pathological case, CAMP could potentially report each digit in a two-digit number from two different points in time, which would lead to incorrect and inconsistent data.

The information provided by `proc` fit well into the CAMP interface. It is relatively low-level, accurate, and can be accessed with little overhead.

4.4 Solaris

Performance data on Solaris comes from two sources: the `proc` filesystem and the kernel statistics chain (`kstat`). The Solaris `proc` filesystem differs greatly from the Linux implementation, following a more standard UNIX pattern. It exports performance data *only* for processes, not system-wide statistics, and the files contain binary data that must be read within a C-compatible language and cast to the appropriate struct type. This effectively precluded direct access from Python. To solve this problem, CAMP uses its own C to Python shared library that handles all data retrieval, marshaling, and error checking. Once again, reads on the `proc` files are atomic and unbuffered to prevent data inconsistency.

Global system data comes from a large data structure within the SunOS kernel called the `kstat` chain. The data structure consists of a linked list of performance counters, each of which can be one of several types: a set of name/value pairs, a binary C structure, or an undefined “raw” type. The data is categorized hierarchically by “class,” “module,” instance number, and name, but this organization is not reflected structurally; that is, no matter what the search criteria, finding an appropriate `kstat` instance is always an $O(n)$ operation.

A research-quality implementation of a Python to `kstat` bridge already existed [2], but it was not compatible with the current version of the SunOS kernel and it was incapable of retrieving many of the `kstat` instances that CAMP needed. Building on this system, CAMP generalized the solution to all instances and introduced compatibility with modern kernels. CAMP’s version of the library also includes a faster search algorithm: the original author made several passes of the chain when one would suffice. This library also includes several bug and data type conversion fixes, and it also adds the ability to produce enumerations of all statistics structures of a certain class or module.

With these enhancements, all global and enumeration functions were able to be implemented with a relatively simple interface. As in the Linux implementation, memory data in page units had to be converted to a byte count.

4.5 Implementation Issues

This section presents a selection of the major implementation issues encountered and their respective solutions.

4.5.1 Raw Data on Windows

The `pywin32` package that encapsulated the Windows performance data handler interface was incomplete. The existence of the “raw” retrieval function was not implemented in the Python wrapper DLL, and its existence was not acknowledged or documented. This left three alternatives: raise the abstraction level of CAMP to that of the formatted counters, leave out functions that were only available via formatted counters, or re-implement Python’s Win32 interface to the Microsoft interface.

Raising the abstraction level of CAMP to that of a formatted performance counter was unreasonable. It would involve the introduction of a state and would require simulation of the periodic updates provided by Windows using threads in Linux. The correct functioning of the system would be increasingly difficult to verify as the number of variables increased.

Leaving out functions that were only available via formatted data was also unacceptable. This would involve the dropping of all network-related functions as well as per-process page fault counting. The former would violate the initial assumptions about the system, rendering it a failure.

As a result, CAMP’s solution was to extend the `pywin32` interface. After setting up a full Win32 build environment, the wrapper function was added to the C++ `pywin32` source. The resultant C “raw data” structure consisted of two 64-bit wide fields that could store several types of data. In addition, it also has a “multiplier” field that keeps a count of how many times a given field has overflowed. CAMP uses Microsoft’s documentation and header files as a strict contract for converting these values into a Python tuple, taking special care to

preserve unsigned numbers² and account for the multiplier field.

4.5.2 Precision and Consistency

CAMP functions should be expected to provide consistent results across all instances of a supported platform, and precision should be the same across all platforms.

Fortunately, precision was not much of an issue with CAMP. Most functions are integral in nature, involving only counts. Python's data handling features handled this in stride: the built in function that converts a string into an integer is capable of detecting whether or not a long object is necessary, and returns it accordingly. The only case where a slight inconsistency in precision existed was in the reporting of CPU usage. Microsoft reports CPU usage as an integral percentage, while CPU usage on Linux is calculated directly by CAMP. To solve this, the Linux functions simply round to the nearest whole number.

The Microsoft performance data handlers are very consistent across the various operating systems that use a Win32 kernel. The core set of counters on Windows 2000, Windows XP, Windows XP x64 edition, and Windows Server 2003 were all identically named and functioned the same. Microsoft only provides one set of performance counter documentation for all Windows platforms.

On the Linux side, the `proc` filesystem is very implementation dependent. In theory, this might render CAMP useless, but in practice, the major performance data in the `proc` filesystem is constant between all major kernel releases; only the debug information varies widely. CAMP functions make every effort to use the

²This actually had to be done using a yet-to-be documented feature of Python 2.4: the ability to convert a 64-bit unsigned integer into a Python Long object.

standardized, machine-readable versions of the process information files rather than the locale-dependent human readable versions. CAMP currently only supports kernel releases in the 2.6.x range, but the necessary data does exist in the 2.4 series.

A more detailed discussion of the verification of CAMP's consistency appears in chapter 5.

4.5.3 Determining CPU Usage

Determining CPU usage from a single function call posed a unique problem. At any instant in time, the usage of a CPU in a time-shared operating system is either zero or one hundred percent. Introducing a persistent polling thread into the system was not a viable option; it would violate the design goals of the interface.

Microsoft provides a false sense of encapsulation with its CPU performance counters. A CPU counter appears to be able to be queried for relevant data at any instant, but a careful look into its implementation reveals that the CPU utilization is calculated as a difference in uptimes, thus requiring at least two samples to provide a meaningful result.

To solve this problem, the CPU functions create a CPU counter and query it immediately, block for a specified interval, and query a second time and return. This causes no additional performance overhead: the task can be unscheduled while blocked, and there is no thread-spawning overhead. In addition, the functions take advantage of Python's "optional parameters" feature. The sample rate is configurable at call time by applying an optional second parameter to the call indicating the block interval. This value is set at 100ms by default, and has

proved to be very accurate while keeping the function responsive (see chapter 5).

The Linux and Solaris functions are implemented in much the same way, but the calculations are performed manually. To determine system-wide CPU utilization, the current uptime and idle time are recorded. The function then blocks for a specified value and records the new values. The utilization is calculated as:

$$\left(1 - \frac{final_idle - initial_idle}{final_uptime - initial_uptime}\right) \times 100$$

or, more simply, the percentage of time not spent in the idle process. The per-process CPU usage functions are implemented similarly but with a finer granularity, as both kernel time and user time are reported separately.

Another issue arose with multi-CPU systems. CAMP should provide the ability to distinguish load on a single CPU from global system load, so the global CPU measuring function allows an optional CPU parameter: a 0-based index that requests the load for a specific CPU. Omission of the parameter causes the total system load to be returned. CAMP also provides a CPU count function to provide information when the count is not known.

4.5.4 Inquiry Functions

The networking and disk functions must operate on a per-interface and per-disk or partition level to be useful. However, device names are certainly not consistent across systems, and indexes, while consistent, are arbitrary and not meaningful.

One solution would be to use common device names to reference the devices. The logical candidate for this would be to map Linux's standard `eth*` or `hd/sd` naming scheme into Windows. However, this would provide an unnecessary bur-

den and layer of confusion for Windows developers. In addition, the mapping of English device names to a universal “code” could prove to be nondeterministic, forcing developers to use empirical tests to determine what “code” referenced the relevant network adapter or disk.

CAMP’s solution is to use an “inquiry function,” a function that enumerates a list of current values. In this particular context, the inquiry functions enumerate the list of installed network adapters and the list of installed disks. However, the semantics of the function are stronger than they appear on the surface. Every output of the inquiry function is guaranteed to be a valid input to the performance measurement functions; this means that the output will not only contain the correct name, but it will also be in the correct format with the correct amount of surrounding white space and capitalization required by the underlying platform.

On the Windows side, this was implemented using Microsoft’s PDH object enumeration function. In Linux, this was implemented by parsing a file in the `proc` filesystem and building a tuple of the results. On Solaris, this data is gathered over one traversal of the `kstat` chain.

4.5.5 Process Addressing

The standard method of addressing a function in an operating system is the process identifier, or pid. However, Windows does not have an $O(1)$ method for programmatically accessing a process’s performance counter based on its pid; one must enumerate all processes and look for a match. This would be an unreasonable solution for CAMP: because the API does not maintain a state, each call to a per-process function would take $O(\textit{number of processes})$ on a Windows machine.

On Linux and Solaris, the opposite is true. Getting process information by pid is achievable in $O(1)$ time, but finding a set of pids from a name is an $O(\textit{number of processes})$ operation.

CAMP’s solution to this is to introduce an abstract concept: the “Process Identifier.” The developer can retrieve a process identifier through a CAMP function that takes a pid as its input. Because this function ideally only runs once per session of use, the $O(\textit{number of processes})$ running time is acceptable.

On the Windows side, this function is implemented by enumerating all current processes and finding the matching process. CAMP then creates a process identifier, which, on this platform is a string, according to Microsoft’s naming conventions. The identifier consists of the name of the executable binary with its extension truncated, and an increasing numeral appended for processes spawned from the same binary. CAMP returns an error value if the process is not found.

The Linux and Solaris functions are much simpler: they merely attempt to open the process’s status file from its `proc` directory. If it fails for any reason (existence of the process or lack of permissions), an error value is returned.

CAMP does not attempt to hide the value of a process identifier in an abstract class. If a developer is aware of the identifier type for the current platform, he or she can bypass the process identifier step.

For convenience, CAMP also provides a function that returns a list of process identifiers given an executable file name. This function returns a list because multiple processes can be running from the same executable—which may often be the case in a distributed system. For example, the `prefork` variant of the Apache web server creates multiple processes to handle child requests. A developer can simply run `GetProcessIdentifiers('httpd')` to get an enumerable list of all running

Apache processes.

Chapter 5

Results and Analysis

Analysis of CAMP comprised two main areas: functional verification and quantitative performance analysis. The verification phase consisted of evaluating the API's runtime behavior across all supported platforms, and the performance analysis phase involved measuring each function's overhead and quantifying its impact on operational systems. Testing concluded with an analysis of CAMP's effects on a running application: a heavily loaded Apache web server. All tests demonstrated that CAMP meets or exceeds each of its design goals.

5.1 Verification

The following sections describe the verification phase of CAMP's testing. First, CAMP was run against a set of tests that ensure consistency in its functional behavior. Second, CAMP measured several testing programs, each of which was engineered to use a specific resource in a measurable amount. Collectively, these tests verified that API's implementation was both correct and consistent.

5.1.1 Unit Testing

CAMP-based applications should be fully platform independent. Moreover, they should not have to have any awareness of the underlying platform and its idiosyncrasies. To satisfy these goals, performance measuring functions must provide consistent behavior for all input, valid or invalid.

As a dynamic language, Python performs semantic checks only as code is executed—there is no compilation phase¹. Because of this, Python cannot statically ensure consistent exit paths or return types across all functions. A Python function’s execution can halt in one of three ways: returning without a return value, returning with a return value, or by propagating an uncaught exception. This presents a problem for CAMP: developers using this API must be guaranteed a strict and consistent interface and should not have to litter their code with type-deducing conditional statements and overly-broad exception handlers.

CAMP solves this problem with a shallow but broad unit test suite, implemented with the `unittest` framework. This framework mirrors the widely used JUnit [25] test suite for the Java programming language. A `unittest` test case is conceptually simple: a test consists of a “fixture,” the test code, and zero or more interspersed assertions. The “fixture” is a procedure that brings the system in to a state before a test is run. As CAMP is stateless, this is empty. The assertions amount to simple boolean comparisons that must hold true for the test to complete in a passed state.

The first round of tests ran the enumeration functions. The test code asserted that no exceptions were thrown and that the return type was of type `List`².

¹Python does actually compile its code to an intermediate bytecode, but it does this at runtime and does not perform many semantic checks.

²When an enumeration function returns no values, it still returns an empty list.

The second group of tests operated on all performance measuring functions and the processor count retrieval function. The test code called all parameter-requiring functions with valid data, e.g. an existent disk, process, or network interface. Assertions verified that no errors were thrown and that the return type of each function was a nonnegative integer.

The third round of tests ran the parameter-requiring functions again with invalid inputs, which consist of either wrongly-typed parameters or references to nonexistent performance objects. The assertions in this test verified that each function exited with a single CAMP-specific exception.

The final group of tests verified the process identifier functions. Under valid inputs, the tests asserted that the two functions, one taking a numeric PID and the other taking an executable name, each return a platform-dependent identifier and a List of identifiers, respectively. Under invalid inputs, defined in the previous test, the functions raise a CAMP-specific exception and no other errors.

This test suite exposed a few inconsistencies in CAMP's original implementation. Each platform-specific implementation was corrected to properly encapsulate the varied exceptions thrown by the native wrapping code. CAMP's interface is now wholly consistent, returning predictable types and errors on all supported platforms.

5.1.2 Result Validation

The objective of the result validation phase was to ensure that each performance function reports reasonable and predictable behavior under the presence of controlled conditions. This could have been implemented in at least two different ways.

The first possible method amounts to simply comparing CAMP’s results directly to preexisting user-mode utilities on each platform. On the win32 platform, this would include the “performance” control panel and the task manager. On Solaris and Linux, `ps` and `top` provide human-readable performance data. However, the use of this method requires the assumption that each utility is implemented correctly and provides continuously updated data that is not cached or delayed.

The second method, which is employed by CAMP’s test plan, involves running the performance monitoring functions against small programs that serve a single purpose: to use a specific resource in a measurable amount. True, correct implementations of these programs would require kernel augmentation on all three platforms, as the operating system is inherently control of the distribution of resources. However, CAMP’s test plan was able to provide a reasonable approximation for CPU, memory, network, and thread usage. This verification plan provided a much more powerful analysis: aside from the confirmation of CAMP’s functionality, this analysis method proves that operating system performance data *in general* is able to be factored out consistently across multiple platforms.

The test bed consisted of a high-end PC³ running VMWare workstation. Each test ran on three virtual machines: Microsoft Windows 2000 Professional (kernel 5.0), SuSe Linux 9.3 (kernel 2.6.11), and Sun Solaris 10 (kernel 5.10). Each operating system ran only its essential services to minimize interference with tests. The virtual machine configuration limited each to 256MB of RAM and a 32-bit virtual processor, and each contained an installation of the following software:

³The host PC has a 64-bit Athlon processor, 2048 MB of main memory, and two 10,000 RPM SATA disks.

```

def waste(amount):
    on_time = _INTERVAL * (amount/100.0)
    off_time = _INTERVAL - on_time
    ctime = clock()
    while True:
        if clock() - ctime >= on_time:
            sleep(off_time)
            ctime = clock()

```

Figure 5.1: The CPU load generation code.

- Python 2.4
- Sun Java Runtime Environment 1.5
- CAMP
- CAMP test cases

CPU Time Verification

To test the various processor time functions, CAMP was set to monitor a simple Python program that generates a fixed load on a single CPU. Put simply, this program divides a discrete interval into an “on time” and an “off time” based on the desired load, and alternates between busywaiting during the “on time” and sleeping during the “off time.” The code for this function appears in figure 5.1.

The global CPU time function (`GetPercentProcessorTime`) and the per-process analogues (`GetCPUtime_*`) were first individually tested against idle, 50% and full system loads. Next, both functions were tested against two processes, each consuming approximately 30% of the CPU. The optional “sample rate” parameter was omitted, leaving a default sampling period of 100ms.

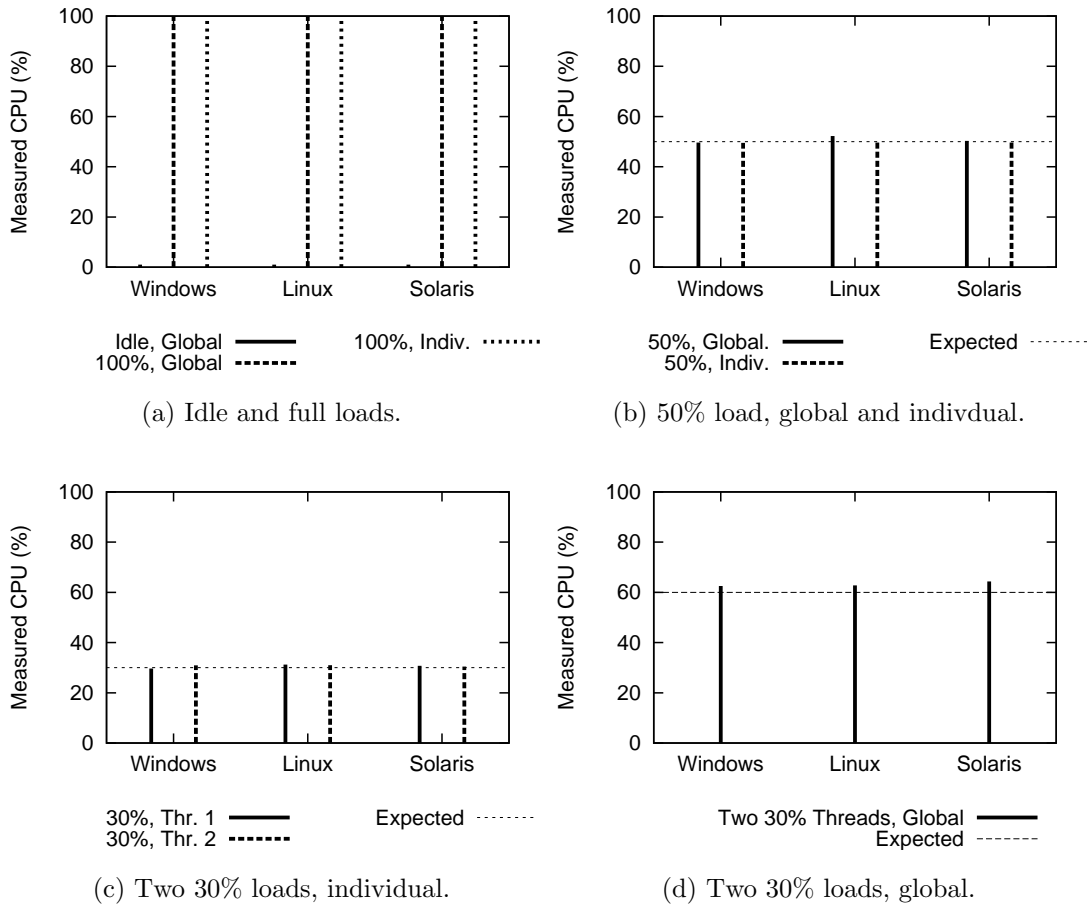


Figure 5.2: CPU time verification results.

All reported values are averages of 200 samples taken at a 0.5 second interval. Each average is the center of a 95% confidence interval on the mean, given that the overall sample distribution is normal. The algorithm used is described in [20]. The range of each confidence interval never exceeded 0.5% and does not cause a visual difference on the scaled graphs.

The results of these tests on appear in figure 5.2. All platforms reported the idle and full loads without error, and each platform was able to provide accurate results for all other intermediate loads. The global measurement of the two-process test read *slightly* above the expected value; this is most likely due to

	Before Alloc.	After Alloc.	Difference
Work. Set Size	9132	10120	988
Phys. Free	416244	415272	(972)

(a) Windows

	Before Alloc.	After Alloc.	Difference
Work. Set Size	14128	15112	984
Phys. Free	10196	9324	(872)

(b) Linux

	Before Alloc.	After Alloc.	Difference
Work. Set Size	13096	14084	988
Phys. Free	387440	386472	(968)

(c) Solaris

Table 5.1: Memory allocation results.

scheduling overhead.

Memory Usage Verification

The memory usage tests operated on the functions `GetWorkingSetKb` and `GetFreePhysicalMemory`. The test program is a simple, two-stage Java program. It first launches and immediately allocates a 1,000,000 byte (976 KB) array on the heap and waits. Upon continuing, it allocates another 976KB array and waits until terminated. Table 5.1 lists the results of the two CAMP functions before and after the allocation on all three platforms.

CAMP reported consistent working set results for the program on all three platforms, albeit with a 4KB disparity on Linux. The Linux virtual machine's page size is 4KB, so this represents a reasonable difference of just a single page.

However, while the working set function reported *consistent* results, it did not report the 976KB allocation *precisely*. Because Java programs run in a virtual

machine that handles allocations on behalf of the running code, precise changes in memory usage, from a program perspective, are not necessarily exactly reflected at the physical level. In practice, however, the difference between the expected and actual values was just over 1% when using these short-lived test programs. This was sufficient for verification.

The free physical memory validation was somewhat more informal. A precise test for this function is impossible to execute in user space, as the full working sets of all tested kernels seldom converge on a completely steady state. This is due in part to each operating system dynamically controlling paging and various caches through periodically-running background threads. Instead of a precise test, this measurement was a verification that this function reasonably reflected the expected changes in free physical pages. In all cases, CAMP's free memory function reported a value within 10% of the expected result.

Network Utilization Verification

CAMP's network functions report the cumulative bytes and packets sent and received on a single interface. To verify these values, the functions were run against a Java program that sent and received a fixed number of packets and terminated. Each test packet consisted of a plain, addressed UDP packet with a 32 byte payload. UDP traffic was ideal for this test because it allowed a precise prediction of the number of packets sent and received; there was no extraneous ACK or connection setup overhead to skew the results. Testing occurred on a private network consisting of only the CAMP-tested virtual machines and their host.

Table 5.2 shows the results of this test. All platforms reported identical

	Before Send	After Send	Difference
Packets Sent	37412	47412	10000
Bytes Sent	2175911	2915911	740000
Packets Received	58299	68299	10000
Bytes Received	13001375	13601375	600000

(a) Windows

	Before Send	After Send	Difference
Packets Sent	60968	70968	10000
Bytes Sent	4157951	4897951	740000
Packets Received	112454	122454	10000
Bytes Received	154652907	155392907	740000

(b) Linux

	Before Send	After Send	Difference
Packets Sent	1762	11762	10000
Bytes Sent	160145	900145	740000
Packets Received	3379	13379	10000
Bytes Received	386700	1126700	740000

(c) Solaris

Table 5.2: Network test results.

packet counts, but Windows reported a different bytes received count. The offending byte count was 600,000, which amounts to 60 bytes per packet. All other byte counts were 740,000, or 74 bytes per packet, which gives a difference of 14 bytes per packet. Several more tests with increased packet sizes showed that this discrepancy always remained at exactly 14 bytes per packet.

The composition of each 74 byte packet is likely 14 bytes of Ethernet header, 28 bytes of UDP overhead, and the 32 bytes of payload. A likely explanation for the 14 byte difference is that the network driver for the virtual machine’s network adapter is not adding the Ethernet header to its incoming byte counter. CAMP’s network functions reflect this discrepancy because the byte counts come directly from the respective network drivers. Informal tests on non-virtual Windows ma-

	Before launch	After launch	Difference
Windows	9	10	1
Linux	9	10	1
Solaris	9	10	1

Table 5.3: Thread count test results.

chines did not show this same asymmetry.

Despite this difference, the key metrics for evaluation were the packet count and the verification that the byte counts were incremented by at least 320,000 bytes ($payload \times packetcount$). In addition, the test programs verified that the payload was delivered in full.

Thread Count Correctness

Verifying the thread count function was a straightforward task. The function `GetThreadCount` recorded the current thread count of a Java program running one extra thread. After a fixed amount of time, the program spawned a second thread. The thread count was recorded once again. The results of this test appear in table 5.3. CAMP recorded the proper increase in thread count on each platform.

This test could have been potentially incorrect. The Java language specification does not guarantee that each conceptual thread is backed with a native thread. However, in practice, the Sun-provided HotSpot virtual machine implementation behaves in this manner on all three platforms. Surprisingly, each platform showed the exact same thread count—which would suggest that the virtual machines are structurally similar.

5.1.3 Informal Verification

The previous sections contain strictly quantified test results that confirmed CAMP’s functionality. However, they were not comprehensive—they had no reference to the enumeration functions, the disk IO functions, or the page fault count. Due to the difficulty in generating predictable results, these functions were part of an informal test plan.

The disk IO functions are difficult to test consistently. The disk IO data provided by CAMP comes directly from the respective disk drivers, so this data does not reflect an operating system-level layer of abstraction. There is no direct relationship between a user-level read or write and actual physical disk action. For this reason, the disk functions were verified informally. On all platforms, the counters were integral, increasing, and strictly monotonic. They also increased their respective rates of increase during periods of high disk activity, making them suitable for a second tier rate function.

The page fault function was also difficult to test. CAMP defines the page fault count as *the number of major and minor page faults experienced by the given process*. However, the precise definition of “major” and “minor” depends on the underlying platform, and these numbers can vary widely. Instead of a direct analysis, CAMP’s output was verified to match that of a comparable native utility on each platform.

On Windows, Microsoft provides an optional utility in the Windows Resource Kit called `pstat`, which provides a crude version of UNIX `ps`. This utility was able to provide a page fault count for comparison. On Linux, the standard `ps` command is able to provide a page fault count with the switch `-O minflt,majflt`. Sun does not provide a utility with Solaris 10 to access the page fault count of a

single process. However, a comparable utility, `pio` [18], was able to provide the needed data to verify CAMP’s function.

The values returned by the enumeration functions are by definition platform and system-dependent, making them impossible to test quantitatively. Instead, the functions were tested informally on an expanded test bed, which included a 64-bit Linux production server, two quad-processor 64-bit Solaris servers, a Windows XP x64 desktop, and the three original virtual machines.

Testing the enumeration functions involved manually collecting a list of required results in a platform-dependent manner. On all six test systems, the CPU count was known beforehand and simple to verify. On Windows, correct values for the network, disk, and partition enumeration functions are available in the “Performance” administrative tool.

On Solaris and Linux, the command `netstat -i` provided a correct list of network interfaces suitable for comparison. Disk and partition lists were available as symbolic links in the device node directories. This information was available in `/dev/disk/by-id/` and `/dev/dsk` on Linux and Solaris, respectively.

5.2 Performance Analysis

As discussed in chapters 3 and 4, high performance and minimal impact were two of CAMP’s principal design goals. This section details the tests used to measure the API’s execution speed and overhead. The first set of tests precisely measured execution speed and CPU usage distribution throughout the implementation components. The second test measured the overhead of a simple CAMP-based monitoring program on a running system. The final test measured

the impact of the same monitoring program on a production web server using externally-derived statistics.

5.2.1 Timing and Profiling

This section contains a low-level, direct measurement of CAMP’s performance overhead. Python provides a timing package, `timeit`, which takes a small snippet of code as an input and aggregates several iterations of it into a single execution unit. `timeit` then runs that larger unit several times, timing it using the highest resolution clock available on the host platform.

Timing runs consisted of a total of 30,000 runs per function: 3 separate execution runs of 10,000 invocations. To measure the overhead of the CPU functions in their purest form, the sample rate was set to zero to eliminate the normal blocking.

For brevity, each “execution unit” comprised groups of similarly implemented functions rather than individual calls. For a function to be “similarly implemented,” it must access the same class of performance data. A trial run on all individual functions confirmed that the grouped functions had near-identical execution times.

Figure 5.3 displays the results of these timing runs. All CAMP functions executed quickly, with the slowest function still allowing for over 100 invocations per second, which is far in excess of the target rate described in chapter 3.

The Linux functions all executed at an order of magnitude faster than the Solaris or Windows counterparts. The Windows functions all displayed relatively significant overhead, but the disk counters were surprisingly fast to access. The Solaris functions behaved as expected: The disk and network functions both

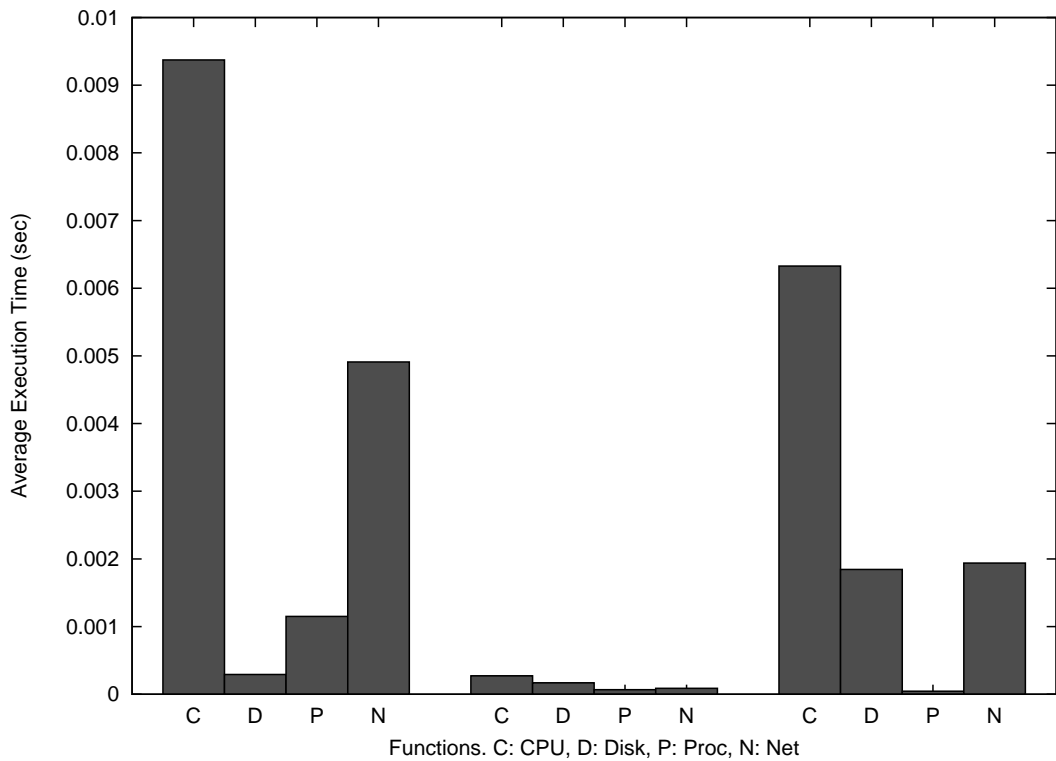


Figure 5.3: Execution time results. Functions are grouped by similar implementations and averaged. From left to right: Windows, Linux, and Solaris.

executed in around the same amount of time as they both traverse the same chain for their target data, while the CPU functions must traverse the chain twice and perform some calculations. Solaris per-process functions make use of the `proc` filesystem, so execution time is comparable to that of Linux. An analysis of the cause the performance disparity appears in the following section.

To put these execution times in perspective, figure 5.4 contains the same information as the figure 5.3, but the graph has been scaled against the execution time of running a native process within Python. As discussed in chapter 3, this is an approach taken by other utilities and was an implementation alternative for CAMP. On Linux, the timing includes forking a second process, executing the command `ps -f`, and collecting its output. On Solaris, the timing performed

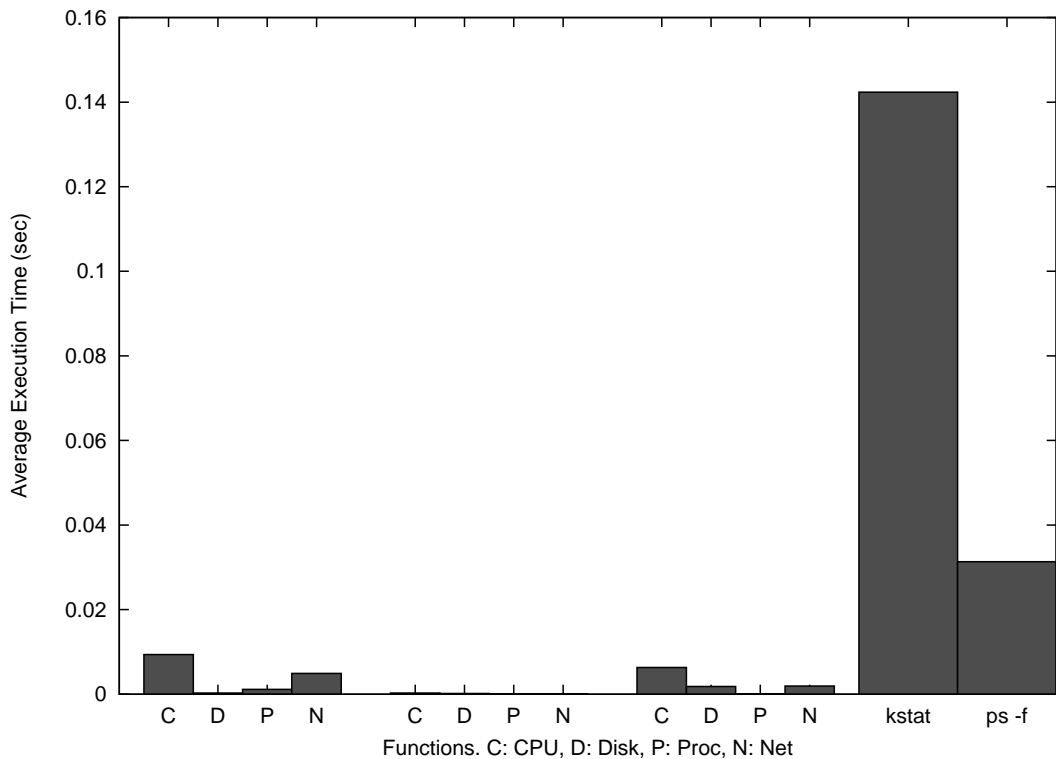


Figure 5.4: Relative execution time results.

a similar action using the `kstat` command line utility. No such command line utility is built in to Windows⁴. CAMP’s design decision to access performance data at the lowest level rather than reusing native utilities clearly allowed a much higher level of performance.

A Closer Look at Windows and Solaris Performance

While CAMP functions execute in reasonable amounts of time on all three platforms, the Linux implementation is an order of magnitude faster than its Windows or Solaris counterparts. The Linux implementation enjoys several ad-

⁴The aforementioned `pstat` tool from the Microsoft Resource Kit is parameterless, which forces the full enumeration of every process and *all* of its performance attributes. Its execution time is on the order of seconds.

vantages: all data is available through direct, $O(1)$ pure-memory access through the `proc` filesystem, and it lacks any expensive native code interfacing. This section contains a brief look at the relatively poor performance on Solaris and Windows.

The Python `profile` package provides the ability to dynamically latch on to preexisting profiling hooks in Python code and determine the relative processing costs of the program's various components. This deterministic form of profiling is ideal for this investigative effort: it adds a significant, constant overhead to execution, but it theoretically provides more accurate relative results than a comparable implementation that uses statistical sampling. A simple Python script aggregated the results of running the profiler against CAMP's functions on Solaris and Linux. Runs consisted of batches of ten runs.

The Windows profiling results showed a significant portion of CAMP's CPU usage taking place in the native function `CollectQueryData`, which retrieves data from the system performance counters and copies it into the local address space. The profiler output also suggested that CAMP spends a considerable amount of time in the overhead required to maintain the "set up, collect, tear down" pattern that is required to keep the performance functions stateless.

The Solaris results confirmed that the per-process functions are nearly as fast on Solaris as they are on Linux. However, anything that required traversing the `kstat` chain incurred a significant "set up/tear down" overhead as in the Windows implementation, and that the traversal of the chain itself involves an expensive Python-to-native call at every node. The chain length varies, but the minimal test machine maintained a length of around 400 nodes, and the quad-processor production server maintained a length of around 800.

Chapter 7 contains potential performance optimizations for both platforms based on these findings. The profiling data proved to be invaluable in developing these optimizations.

5.2.2 Measured Overhead, or CAMP on CAMP

While microbenchmarks provide a solid foundation for predicting a system's speed, practical tests are still necessary when attempting to predict performance in an operational domain. This section details the use of CAMP to measure the overhead of a CAMP-derived monitoring program. This test involved two components: a tunable CAMP-based program to measure and some method for reliably and consistently taking measurements.

The first task involved creating a tunable monitoring program, `probe`, that both represented CAMP's intended use and did not incur any artificial overhead. This program used CAMP as its foundation, but called the CPU-measuring functions with a zero sample rate to eliminate blocking. It called nine CAMP functions at a time, with the entire nine-function block being executed at a specific sample interval. To implement the sampling frequency as accurately as possible, this monitoring program times the execution of the nine-function block at startup and adjusts the sleep interval accordingly.

The first test consisted of measuring the maximum throughput of the monitoring program. This was done by letting `probe` run at its maximum rate and measuring the average number of calls per second. The results of this test are displayed in table 5.4.

CAMP itself met the requirements for the second component of this test. First, CAMP was used to create a simple monitoring program, `monitor`, that

	Max 9-Function Rate	Effective Max Rate
Windows	40	360
Linux	1096	9864
Solaris	100	900

Table 5.4: Practical maximum function call rates.

dumps the output of most of its functions to the terminal at a 0.5 second quantum. Next, this program was set to monitor itself to check for any obvious performance problems. The program reported no significant measurable resource usage, so it became the recording/reporting method for the following tests.

The subsequent tests consisted of running `probe` at increasing sample rates and measuring the impact on the system at each level using `monitor`. While the results of every CAMP function were included in the measurements, the only significant differences manifested themselves in the global and per-process CPU functions. However, this exercise did confirm that CAMP is indeed stateless and without memory or thread leaks on all implemented platforms. The results of this test on the three implemented platforms is shown in figure 5.5. CPU values are an average of 20 samples taken over a 10 second period.

These results were in line with the values recorded by the timing functions. On Solaris and Windows, CAMP produces little overhead up to 10 function blocks/second (90 functions/second). On Linux, CAMP produces little overhead at all measured sample rates.

5.2.3 Impact on an Operational System

The final performance test consisted of measuring CAMP's overhead from an externally accessible statistic. This test is important for several reasons. First,

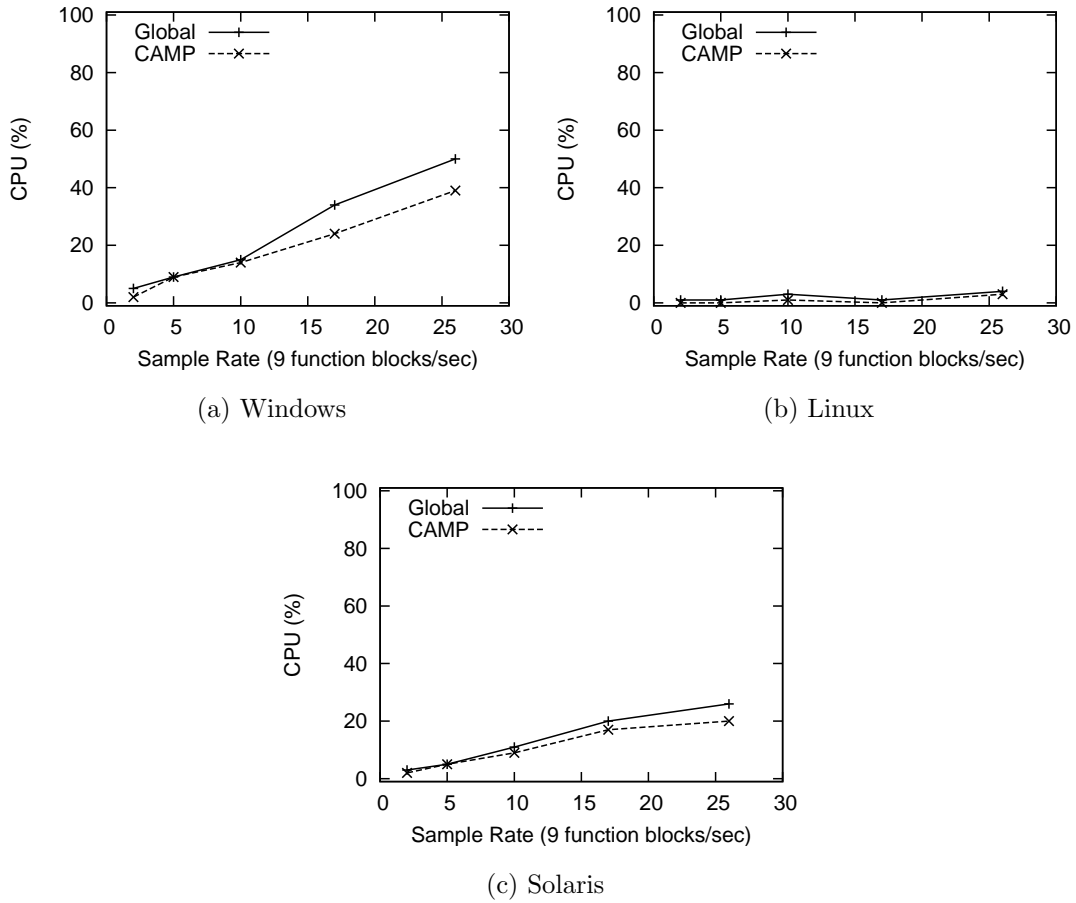


Figure 5.5: Operational impact at increasing sample rates.

the results of the previous test could have been partially skewed by using CAMP to test itself—caches could have been kept artificially hot. Second, the concept of “CPU utilization” is a somewhat coarse statistic that does not always *directly* map to actual or perceived performance. Third, CAMP may cause performance impacts in the kernels of the respective platforms in a non-obvious way that may skew results. Lastly, it allows a complete distrust of the operating system-provided performance data.

For this experiment, the target application was a web server. Each virtual machine was configured with Apache 2.0.53. The Linux and Unix machines used

the `prefork` multiprocessing module (MPM), and the Windows machine used the default Windows NT MPM. Each Apache installation used the default MPM configuration parameters, including initial process and thread count.

This testing phase consisted of two experiments, each at a different load level: one “normal,” and one critically high. Each test involved running the `probe` program from the previous test at the same sample intervals, but instead of recording system performance data on the server, Hewlett Packard `httperf` [30] collected http client response statistics from an external machine.

To generate a normal load, `httperf` created a total of 1400 connections to each web server at a rate of 40 connections/second. This rate produced an approximate 15% load on the Linux and Solaris machines and a 30% load on the Windows server. The results of this test are presented in figure 5.6.

CAMP only produced a measurable impact on the Windows server. It appeared to take a progressively increasing hit as `probe`'s sample rate increased. CAMP did not affect the performance of the Solaris and Linux servers at this sample rate. This was somewhat unexpected, given that CAMP's performance on Solaris was comparable to its Windows performance. A likely explanation lies in the implementation of the Apache MPM on each platform. The Windows version of Apache handles all client connections from within a single, multi-hundred-threaded process. In terms of process scheduling, this lone process has the same weight and priority as `probe`. This causes CAMP's impact to be more pronounced, even at relatively low sample rates. On Solaris and Linux, Apache `prefork` responds to the increased load by forking additional processes, which causes the Apache system as a whole to be scheduled more often than `probe`. On the whole, however, CAMP did not cause significant overhead until well above the original design goal described in chapter 3.

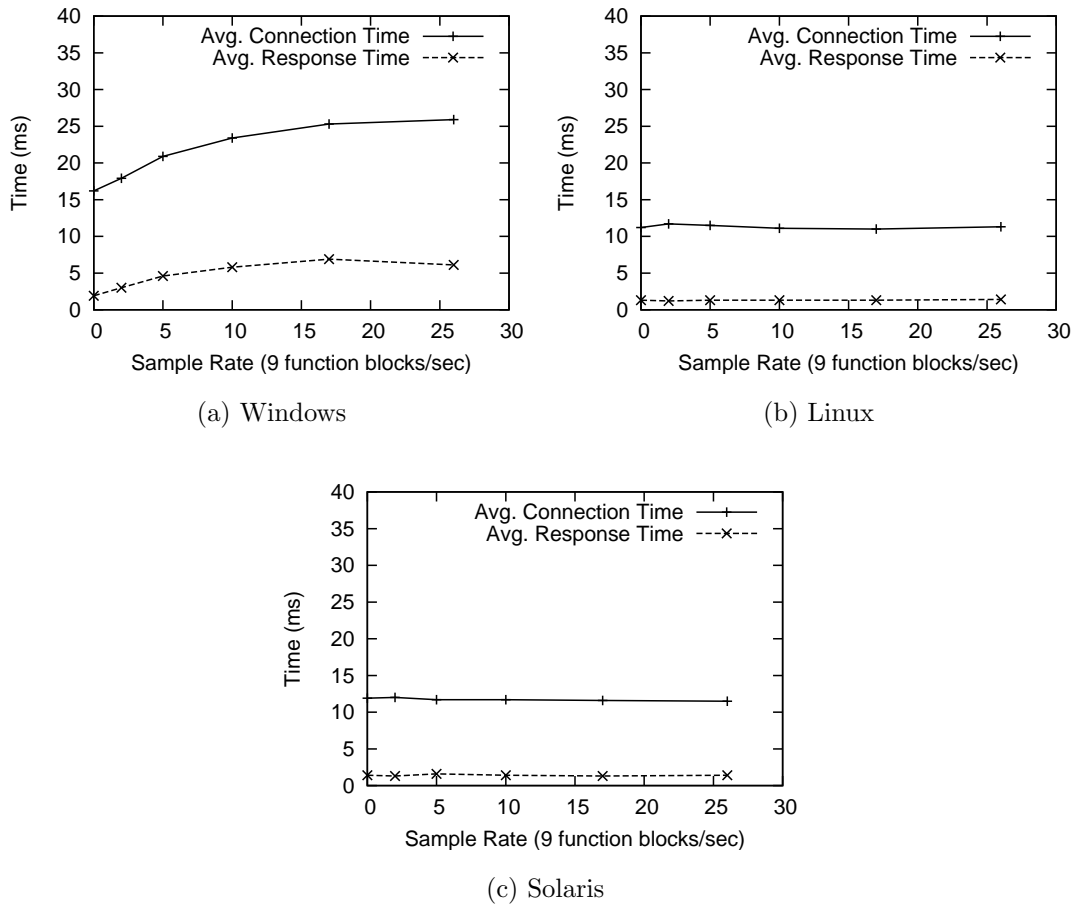


Figure 5.6: Results of the operational system impact test.

Following from the first test, the second experiment involved testing the more robust servers, Linux and Solaris, at critical, breaking-point loads. The original test setup was incapable of generating a load that would drive either of these servers to failure, so the virtual machines were transplanted on to a system with far fewer resources⁵. In addition, the virtual machines were restricted to 256 MB of main memory, half as much as originally allocated. `httperf` ran on the high-end machine. With the increased client power and decreased server resources, `httperf` was able to push both the Linux and Solaris servers to failure. This test

⁵The “slow” test system has a 1333 MHz Athlon processor, 512MB of main memory, and a 7200 RPM disk

involved the following steps:

1. Determine the range of sample rates that marked the end of normal behavior for the web server. This is the point at which the response time graph breaks from its flat trend and curves sharply upward.
2. Starting from this point, work backward to find the first point at which CAMP (through `probe`) causes a measurable impact on the system.
3. Record the response times at the same sample rates used in section 5.2.2, and observe the characteristics of CAMP's impact.

Figure 5.7 contains the results of this test. The graphs on the upper row are the results of step 1: the diagnosis of each server's limits⁶. On Linux, the server begins a sharp increase in response time at 95 cps, and the system becomes exponentially slower and begins dropping connections at around 100 cps. The Solaris server exhibited an increase in response time at 79 cps, and it became unusable by the time the rate reached 83 cps.

The graphs on the lower row contain the data collected in steps 2 and 3. CAMP produced such a minimal impact on Linux that it would not affect response time in a measurable amount unless the request rate was set within the beginning of the failure zone—95 cps. On Solaris, CAMP produced a measurable impact all the way down to 78 cps. These respective `httperf` sample rates were used to produce the data in the graphs on the lower row.

This data reflects important properties of CAMP. First, despite the fact that the systems were operating at absolute limits, CAMP's impact remained linear.

⁶The unseen portions of the connection graphs (from zero to the minimum value shown) are flat and minimal.

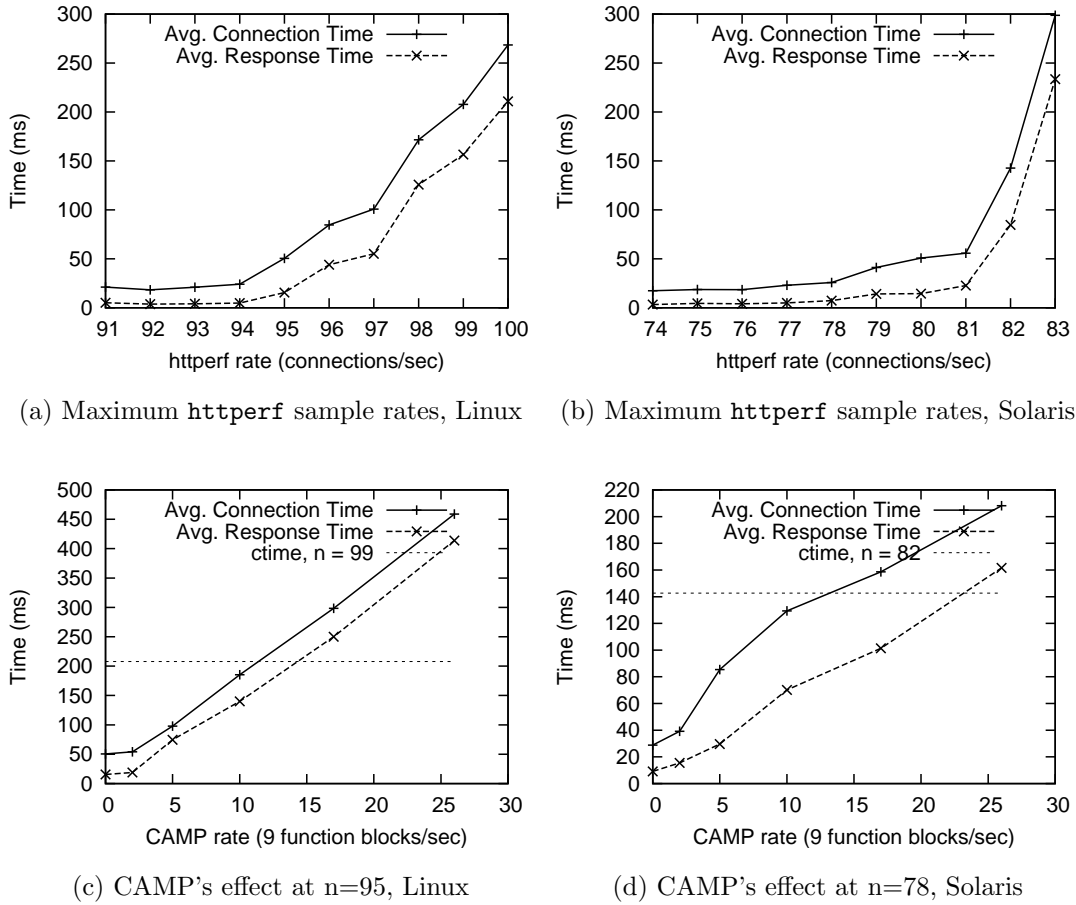


Figure 5.7: Results of the system impact test at a critical system load.

This suggests that the API does not affect the system in non-obvious ways, like taking an excessive share of kernel resources or scaling exponentially. Second, the graphs show that CAMP's impact is on the order of just a few percent of system capacity. This is evidenced by the horizontal lines on each of the lower graphs, which denote the average connection time when `httpperf` is run at ($measured_cps + 5$) without CAMP. The x-value of the intersection of this line with the "average connection time" plot correlates the upper and lower graphs. This point describes the CAMP rate that produces the same impact as 5 additional connections per second at this load level. In both cases, this number is between 10 and 15

samples/second, which is much higher than the sample rate of a normal use case.

Collectively, all tests demonstrated that CAMP meets or exceeds each of its design goals. While there is some room for performance improvement, the current implementation is stable, correct, and causes little overhead on all implemented platforms.

Chapter 6

Related Work

CAMP's design and implementation share many qualities with other systems. Because the API is designed for use with distributed system testing, the bulk of this chapter focuses on distributed testing methods and frameworks. This chapter also discusses distributed monitoring and benchmarking systems that share characteristics with CAMP.

6.1 PAPI and Derivatives

PAPI [4] is a system that shares several design goals with CAMP, but it accomplishes a slightly different task: it provides a platform-independent interface into *hardware* performance counters rather than operating system counters. On most desktop architectures, the underlying architecture is capable of providing controllable metrics about processor and low-level cache events. Each architecture has a different set of assembly instructions for accessing these counters, and each set has different semantics. PAPI provides a platform independent interface to these counters through a standard C and FORTRAN interface, but it requires

operating system kernel augmentation on most common platforms. PAPI does provide accurate metrics: its hardware counter interface has been used to validate other benchmarks [32].

Unlike CAMP, performance data provided by PAPI is difficult to correlate with a specific process. Utilities that allow this behavior either ignore the effects of scheduling or instrument the operating system kernels to report scheduling events [31].

PAPI does provide a portable interface that does not require operating system modification. However, it is not implemented on Windows, and its Linux and UNIX implementations simply call `getrusage(2)`. This module of PAPI appears to be deprecated, as it no longer appears in the current PAPI version 3 documentation.

The developers of PAPI encountered problems while factoring out functionality that were similar to those encountered during CAMP's design phase: newer processors like the Pentium 4 have a more complete set of counters than older processors [39]. Instead of going with the intersection of available features, PAPI's developers chose to break consistency and to provide more functionality on the more full-featured architectures. To provide a common interface, PAPI's designers took a different approach to performance data querying. Counters are named entities that are accessed with a relatively small set of functions that take the entity names as parameters. This contrasts with CAMP's design, in which each function returns a single performance data that is directly related to the function name.

Like CAMP, PAPI also strives to be the low-level foundation for other performance tools. PerfSuite [21] implements a platform-independent profiler using

PAPI that requires no modification of executable code. Similarly, the SvPablo [37, 36] system uses PAPI's platform independence to add parallel application profiling to its suite of performance analysis tools.

PAPI and CAMP provide two different sets of data, and each strives to be a simple, correct set of building blocks for other tools. Chapter 7 discusses a possible future relationship between the two projects.

6.2 Distributed System Monitoring

System performance data is often associated with both performance testing and overall system monitoring, and CAMP's functions provide a set of performance data that is of use in both areas. This section details other efforts to monitor heterogeneous distributed systems.

Early research in distributed systems monitoring involved two main challenges: the selection of relevant performance data and the aggregation and display of the data. The performance data reported in TMP [46] is similar to that provided by CAMP, and it uses the same layered architecture. This work was tied to both a single platform and a single architecture. Early exploratory research works ([7] and [3]) discuss the relevance of various types of performance data to distributed system developers, and they provide a good demonstration that system-level performance data, like the data provided by CAMP, can be of definite use to the distributed developer.

Many system monitoring solutions are implemented using the SNMP protocol [5]. It provides a simple request/response protocol for system monitoring and management. SNMP is an application level protocol, implemented using UDP,

that allows clients to host a set of named data. Examples of managed data include host name, uptime, and load. Because the protocol is at the application level, the local client implementation (the SNMP agent) is in full control of what data is reported and how it is obtained. The latest version of the protocol provides bulk request/response functionality that scales more reliably with large distributed systems [41]. When used with common object names, SNMP is capable of providing a common interface into operating system performance data by means of a common network packet. However, the SNMP agent that runs locally on the monitored machine and services requests must still be able to actually *collect* the system performance data, which in most cases must be accomplished with native code. In this scenario, CAMP can provide a simple method for implementing a platform-independent SNMP agent that provides performance data for a set of managed hosts.

Hybrid system monitoring solutions [15] take a different approach to monitoring. These systems rely on specific hardware that can collect and serve performance data with little overhead. Applications can selectively write performance data into this hardware using an auxiliary interface, and independent hardware appliances aggregate the data and correlate the recorded events with high-resolution timers. Unlike CAMP, this method is tightly coupled to a specific architecture.

The Eddie system for Python [27, 26] is a CAMP-like framework designed for system administrators. The tool allows the user to set up rules and alarms that are triggered by various performance events. The most common examples involve notifying an administrator when the current state of a service or host changes.

Eddie can provide much of CAMP's data on a variety of platforms, but it suffers from several implementation issues. The tool provides a set of common

“directives” for accessing system and performance data, but these directives more aptly describe classes of performance data rather than actual specific values. Examples include `SYS` for system performance data, `DISK` for IO statistics, and `PROC` for process data. Eddie’s main flaw is that it lacks any sense of consistency: the contents of these directive classes are completely OS-dependent. For example, Eddie provides a “rule” that allows the triggering of an event when the sustained load average of the system exceeds a certain amount. However, this feature simply does not exist on Windows.

Eddie also suffers from performance and correctness problems. On all UNIX-like platforms, Eddie implements each API call by forking a second process and parsing the output of a native utility like `ps`. As discussed in chapter 5, this approach yields much higher overhead than CAMP’s purely programmatic design. Additionally, Eddie is tied to network reporting; the operating system reporting modules are not cleanly separated from the rest of the program. A survey of Eddie’s source code yields several errors:

- Eddie does not report raw data on Windows properly like CAMP does—it actually incorrectly reports rate-calculated data as if it was a raw count.
- Several statistics are missing from many of the implemented platforms.
- Eddie only supports bulk calls; that is, one cannot actively poll for the CPU usage of a single process with receiving every performance statistic for that process.

Eddie-related publications make no mentions of any efforts to formally test correctness or overhead.

The addition of CAMP to the Eddie project would greatly strengthen the

system’s reporting accuracy. It could be used to either supplement or replace the existing functionality.

6.3 Distributed Performance Testing

CAMP’s primary intended use case lies in distributed performance testing. This section compares CAMP to several distributed performance testing frameworks and tools, and it discusses how CAMP’s functionality compares to the existing facilities and how it might be used to supplement the existing measurement methods.

The automatic distributed performance modeling tool described in [34] provides a performance modeling framework for distributed systems. The prototype implementation is tied to a specific platform, but it does provide many of the same statistics as CAMP, which suggests that CAMP could provide this system with platform independence. Interestingly, this system specifically addresses the problem of coarse-grained system statistics being too general for application performance measurement: the authors uses statistical regression techniques to artificially sharpen the granularity of process-level statistics. Techniques like this could be used to expand CAMP’s functionality as described in chapter 3.

DiPerf [10] is an automated distributed testing framework. It has no functional overlap with CAMP: its metrics are user defined (and thus specialized to a specific domain). For example, user-defined variables might include requests per second and system response time. DiPerf does, however, provide a strong layer for collection and aggregation of data, which makes it an excellent candidate to layer on CAMP.

The Simple [17] system, while somewhat historical at this point, is an early example of a system like DiPerf. It involved research in the area of collecting domain-dependent performance data from a large parallel system. It encapsulates all data in a common, named packet structure. As in DiPerf, CAMP would fit well in this system, augmenting existing functionality with system performance data.

Developed at the Lawrence Berkeley National Lab, NetLogger [13, 12, 42, 14, 23] is an actively-developed solution for low-impact distributed system monitoring. However, it does not tie itself explicitly to the performance data domain—the high-performance logging framework can be used for a variety of applications. NetLogger’s developers have focused on solving the problems of compacting and aggregating voluminous performance data. The system also provides a generalized visualization framework. To use NetLogger, developers must hand-instrument their code with calls to the general NetLogger API and report the desired information manually, which may be cumbersome for some applications. This suite of tools would be an excellent host for CAMP: NetLogger would provide a solid reporting, distribution, and visualization mechanism for CAMP’s performance data. Netlogger is capable of handling on the order of thousands of events per second, which is in excess of CAMP’s capacity on most platforms.

The Black Box system [1] is similar to NetLogger, but it requires no instrumentation of source code. It deals solely in unobtrusive measures like system response time and throughput. This is a design choice made by the authors; they feel that this greatly simplifies performance testing and encourages it at earlier stages in development. They believe that the loss of precision by treating a distributed system as a system of black boxes is far outweighed by the ease of use afforded by this design choice. This tool currently does not make use of system performance

data, but it is an ideal candidate to integrate with CAMP: the authors are most concerned with being unobtrusive and only measuring data that can be probed externally. CAMP’s measurements are unobtrusive, platform-independent, and able to be queried at a frequency that yields statistically significant results.

The Gloperf system [22] is part of the Globus GRID computing toolkit [11]. By using the GRID framework, it is able to measure selected performance statistics in a platform independent manner. The client side of this system runs as a daemon, and it actively collects its own statistics rather than broadcasting operating system or network protocol statistics. It uses a sensor/collection model in which the user installs a “sensor” on the client to be probed and periodically collects data. The “system status suite” described in [24] follows a similar model. CAMP follows a completely different model: it only reports data that is already available on the host. Despite this, CAMP’s platform independence would allow straightforward integration with the Globus GRID toolkit.

Astrolabe [35] is a general large-scale distributed monitoring system that can be used for performance monitoring. Its main draw is its strong claims of scalability: it uses peer to peer (P2P) technology to efficiently collect and deliver performance data from large distributed networks. On the clients, Astrolabe measures data using an “Astrolabe agent,” which is similar to an SNMP agent. CAMP could leverage this technology to report system usage statistics for very large networks.

CoMon [33] is a monitoring layer for the PlanetLab testbed [6]. It collects data from individual PlanetLab distributed nodes, and its concept of a “node-centric daemon” can deliver nearly all of the performance data that CAMP would provide. However, this tool only works with the PlanetLab operating system (a modified version of Fedora Core Linux), which makes the research less useful for

systems outside of that controlled domain.

The Compass framework [29] accomplishes platform independence of performance data in a different way—by using the Java Management Extensions (JMX) framework. It uses dynamic instrumentation techniques to interleave user code with calls to a performance reporting API that report user-selected data. However, all performance data is limited to that which can be reached within the Java Virtual Machine. This is useful for measuring application-specific metrics, but nearly all of CAMP’s performance statistics require native code to access and are not available through this system.

The Tau [28] set of tools is a high performance computing testing framework that has been recently updated to collect full statistics from a Java Virtual Machine [38]. This has been used to create a Java profiler that can selectively instrument and measure parallel and distributed Java applications. Unlike the Compass framework, which is designed for business applications, this framework is designed to diagnose performance problems in massively parallel systems. The tool’s feature set is comprehensive, and its standard interface allows it to be used on a number of platforms. However, all measurement is confined to within the virtual machine—Tau cannot measure system-level statistics.

The Cougar distributed agent framework has a strong performance analysis component [16]. Agents built and deployed using the Cougar framework are able to use system performance data as criteria for migrating between hosts and choosing whether or not to begin an expensive computation. The performance measuring component of the framework focuses on multiple levels of metrics: system level, distributed network level, and application level. As a Java-based framework that does not rely on native code, Cougar runs on a variety of platforms. It purports to be able to report network, CPU, and memory usage from

within Java—which, at first glance, would make CAMP appear to be a duplication of its effort.

However, an inspection of the latest release of Cougaar’s source code (version SE 12.0, January 2005) reveals substantial limitations:

- All metrics are limited to the currently running process.
- Memory utilization is limited to reports of the usage within the Java Virtual Machine—not the agent’s actual working set.
- Network statistics are not provided by the network driver. They are collected by keeping a set of stateful counters within an alternate Socket implementation.
- The CPU usage collection, which requires a native code bridge, is only implemented on Linux and Windows and will not compile on any other operating system.

Cougaar’s literature makes no reference to attempts to verify the correctness of these metrics.

Cougaar would be an ideal candidate for integration with CAMP. The performance data provided by CAMP (especially its CPU load reporting) is verified to be accurate at both the system and process level, and Cougaar-based agents rely on accurate CPU data as a basis for decision making. Integration with CAMP could allow agents to make decisions based on other factors as well: high frequencies of page faults, network utilization, or disk load.

6.4 Benchmarking

CAMP's metrics are not benchmarks in and of themselves, but they are useful in supplementing distributed benchmarks.

GridBench [44, 43, 9, 45] is a system that attempts to measure heterogeneous distributed system performance potential by translating and running microbenchmarks on the GRID. The authors of this system focus on high-performance computing benchmarks like epWhetstone and hplinpac. While the data provided by the various microbenchmarks might overlap with CAMP's functionality, the benchmarks themselves are not platform independent. Users must still have platform-specific binaries for each desired benchmark, and only the interface by which they are launched is made platform independent by the GridBench Description Language.

The Xampler benchmarking suite [19] provides a large suite of benchmarks for CORBA implementations. By design, CORBA is platform independent, and Xampler properly reports its benchmarks on both the Linux and Windows platforms. Xampler's benchmarks run along side the object broker systems, so their platform independence is implemented through portable code rather than a write-once, run anywhere system. Xampler's primary metrics are mostly composed of throughput and response time of the CORBA implementation, but it also includes statistics like average load and system memory usage over time. These metrics are similar to those provided by CAMP.

These finer grained, non-specific benchmarks duplicate some of CAMP's functionality, but they are limited to global free memory and CPU usage on Windows and Linux only. Xampler does not actually attempt to create a platform-independent performance interface: it merely exports its C-implemented library

to Java using the Java Native Interface (JNI) for use within that language. Like the Eddie system, the implementation only supports the bulk collection of performance data, which would hurt performance in a high-rate sampling scenario.

6.5 Summary

CAMP’s work is related to several research topics—several of which reside under the distributed systems area. Both distributed monitoring applications and benchmarking suites could use CAMP to either 1) replace their existing, broken system performance data component or 2) augment their existing functionality. Existing distributed performance frameworks, with their focus on efficient performance data distribution, would serve as an ideal platform for the deployment of CAMP-derived data.

As mentioned in the previous sections, some projects do have components that are capable of retrieving partial operating system performance data, and in some cases, the data is retrieved in a somewhat consistent way. However, these components lack CAMP’s completeness, consistency, correctness, and its robust implementation across three platforms. CAMP provides a novel solution to the problem of retrieving mixed-platform operating system performance data: unlike existing one-off solutions, CAMP is a strictly-defined, well-tested interface.

Chapter 7

Future Work

CAMP is in a stable state, but it does provide several opportunities for future work. The most immediate opportunity involves working to increase the performance on Windows and Solaris. Other tasks include the expansion of CAMP's functionality and building useful applications using the API.

7.1 Performance Optimizations

As discussed in chapter 5, performance on Windows and Solaris is an order of magnitude slower than on Linux. It is unlikely that the slower platforms can be optimized to the level of Linux without kernel modification, but there are a few areas where speed can be improved.

In all Windows functions and the global Solaris functions, CAMP uses a similar code pattern:

1. Allocate and open some system resource corresponding to the metric to measure.

2. Instruct the operating system to collect or retrieve performance data on that metric.
3. Retrieve the metric's value.
4. Deallocate and close the system resource.

The profiling results indicate that this constant allocation and deallocation of resources is contributing to CAMP's extra overhead. A possible solution to this problem would be to introduce a cache for these resources.

Unlike a traditional object pool, this cache would contain references to objects that have *open connections* to operating system resources. Thus, CAMP would be both caching the allocated resources and pooling the open connections. When a function is called twice in a row, CAMP would reuse the existing connection and re-collect and retrieve the metric's data without going through the set up/tear down pattern. This cache would be populated after a cache miss, and a background thread would selectively time out and close open connections as needed. This automatic timing out would enforce CAMP's stateless behavior and make the interface appear identical. This pooling of open resources most closely models SQL connection pooling.

There is another straightforward optimization for the Solaris global functions. In the current implementation, CAMP's traversal of the `kstat` chain causes a constant exchange of data from the native interface into Python. When searching for a specific statistic, the C library advances the chain forward by one. Then, the Python library itself does all comparisons and checking on the current node, retrieving the metric if necessary. This forces data to be exchanged after every step, which can cause on the order of hundreds of expensive native calls per function execution. Implementing the full linked list walk in native C would

reduce the total number of native calls to one, and the native traversal of the linked list has the potential to be intrinsically much faster, regardless of the unnecessary data exchanging.

7.2 Continued Development

CAMP’s most important future work item is the implementation of the API on more platforms. Other UNIX and UNIX-like operating systems, like *BSD, Mac OSX, and HP-UX are all good candidates for future expansion.

Expansion to new platforms could be expedited by an automated compliance test suite. CAMP’s current automated tests, described in chapter 5, verify only the consistency and stability of the API—not its actual correctness. This test suite could be extended to include *all* tests in chapter 5, fully automating the testing of the accuracy of the various functions. The test suite could control the execution of the programs that generate known loads, and it could provide throughput, timing, and profiling information in a single report. Unfortunately, it would be difficult to automate the testing of the enumeration functions. This test also has the ability to act as an extremely strong regression test, guarding CAMP against potentially destructive but subtle changes to future versions of supported kernels. This level of automated testing might convince the Python community to add CAMP’s functionality to the optional `os` package in the Python standard library.

CAMP’s functionality could also be improved. As considered in chapter 3 and discussed in chapter 6, the API could be extended with a set of “emulated” functions. For example, Solaris only provides a bulk count of all input and output bytes added together on a per-process basis. Solaris does, however, provide counts

of disk blocks read and written per-process. CAMP could estimate the specific number of bytes read and written by adjusting the bulk count by the ratio of disk blocks read and disk blocks written to total blocks written, respectively.

Finally, CAMP could benefit from a standard set of platform independent secondary functions. The API has the potential to be more widely used if it is distributed as more than a just simple, low-level API. Examples of standard library functions could include rate or throughput functions, periodic logging functions, and bulk performance status functions. The bulk functions could provide a single entry point for a performance “dump,” and they could include data available from other Python packages, like the current operating system, kernel version, and Python version.

Chapter 8

Conclusions

CAMP succeeds in providing a common interface into operating system performance data. Its research is novel, and it provides a substantial, production ready implementation. CAMP met each of its design goals:

- CAMP is complete and correct. It provides good coverage of all major operating system services, and it has been demonstrated to be accurate.
- The API provides proper functionality on three very diverse platforms. This in and of itself is a testament to the ability of operating systems to provide similar performance data, and it suggests that implementation on further platforms will be possible.
- The solution is general and low level, making it applicable to a variety of scenarios.
- CAMP performs in far excess of its original performance goal. Its functions can be sampled at high frequency rates with very little overhead.
- CAMP has a great potential for actual use in several cases: distributed

system testing and research, system monitoring, and desktop application debugging.

Finally, CAMP is a solid platform for future work. There are countless possibilities for derived functions and applications, and there are interesting opportunities for enhancement to CAMP itself.

Bibliography

- [1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM Press, pp. 74–89.
- [2] ANNIS, W. pykstat: kstat api for python. World wide web, 2001. <http://www.biostat.wisc.edu/annnis/creations/pykstat.html>.
- [3] BRADDOCK, R. L., CLAUNCH, M. R., AND RAINBOLT, J. W. Operational performance metrics in a distributed system. part ii.: Metrics and interpretation. In *SAC '92: Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing* (New York, NY, USA, 1992), ACM Press, pp. 873–882.
- [4] BROWNE, S., DONGARRA, J., GARNER, N., LONDON, K., AND MUCCI, P. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 2000), IEEE Computer Society, p. 42.
- [5] CASE, J. D., FEDOR, M., SCHOFFSTALL, M. L., AND DAVIN, C. RFC 1157: Simple network management protocol (SNMP), May 1990.

- [6] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.* 33, 3 (2003), 3–12.
- [7] CORWIN, B. N., AND BRADDOCK, R. L. Operational performance metrics in a distributed system. part i.: Strategy. In *SAC '92: Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing* (New York, NY, USA, 1992), ACM Press, pp. 867–872.
- [8] CUSUMANO, M. A., AND YOFFIE, D. B. What Netscape learned from cross-platform software development. *Commun. ACM* 42, 10 (1999), 72–78.
- [9] DIKAIAKOS, M. D. Grid benchmarking: Vision, challenges and current status. Tech. Rep. TR-2005-11, Department of Computer Science, University of Cyprus, May 2005.
- [10] DUMITRESCU, C., RAICU, I., RIPEANU, M., AND FOSTER, I. Diperf: an automated distributed performance testing framework. In *Proceedings. Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)* (Nov 2004), pp. 289–296.
- [11] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.* 15, 3 (2001), 200–222.
- [12] GUNTER, D., TIERNEY, B., CROWLEY, B., HOLDING, M., AND LEE, J. Netlogger: A toolkit for distributed system performance analysis. In *MASCOTS '00: Proceedings of the 8th International Symposium on Model-*

- ing, Analysis and Simulation of Computer and Telecommunication Systems* (Washington, DC, USA, 2000), IEEE Computer Society, p. 267.
- [13] GUNTER, D., TIERNEY, B., JACKSON, K., LEE, J., AND STOUFER, M. Dynamic monitoring of high-performance distributed applications. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 163.
- [14] GUNTER, D., TIERNEY, B. L., TULL, C. E., AND VIRMANI, V. On-demand grid application tuning and debugging with the netlogger activation service. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 76.
- [15] HARDEN, J. C., REESE, D. S., EVANS, M. B., KADAMBI, S., HENLEY, G. J., HUDNALL, C. E., AND ALEXANDER, C. In search of a standards-based approach to hybrid performance monitoring. *IEEE Parallel Distrib. Technol.* 3, 4 (1995), 61–71.
- [16] HELSINGER, A., LAZARUS, R., WRIGHT, W., AND ZINKY, J. Tools and techniques for performance measurement of large distributed multiagent systems. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (New York, NY, USA, 2003), ACM Press, pp. 843–850.
- [17] HOFMANN, R., KLAR, R., MOHR, B., QUICK, A., AND SIEGLE, M. Distributed performance monitoring: methods, tools, and applications. *IEEE Transactions on Parallel and Distributed Systems* 5, 6 (Jun 1994), 585–598.

- [18] HUANG, Y. pio: Solaris process I/O. World wide web, 2004. <http://www.stormloader.com/yonghuang/freeware/pio.html>.
- [19] KALIBERA, T. Regression benchmarking environment. In *Proceedings of WDS'04* (Prague, Czech Republic, 2004), MatfyzPress, Charles University, pp. 174–178.
- [20] KNUTH, D. *The Art of Computer Programming*, third ed., vol. 2. Addison-Wesley Professional, 1997, p. 232.
- [21] KUFRIN, R. Perfsuite: An accessible, open source performance analysis environment for linux. In *6th International Conference on Linux Clusters: The HPC Revolution 2005* (Chapel Hill, NC, USA, Apr 2005).
- [22] LEE, C. A., STEPANEK, J., WOLSKI, R., KESSELMAN, C., AND FOSTER, I. A network performance tool for grid environments. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 1999), ACM Press, p. 4.
- [23] LEE, J., GUNTER, D., STOUFER, M., AND TIERNEY, B. Monitoring data archives for grid environments. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 1–10.
- [24] LOGAN, M. J. Monitoring and state transparency of distributed systems. In *ERLANG '04: Proceedings of the 2004 ACM SIGPLAN workshop on Erlang* (New York, NY, USA, 2004), ACM Press, pp. 7–10.
- [25] MASSOL, V., AND HUSTED, T. *JUnit in Action*. Manning Publications, 2003.

- [26] MILES, C. System monitoring, messaging, and notification. In *Proceedings of SAGE-AU'99* (Jul 1999), The System Administrators Guild of Australia.
- [27] MILES, C., AND TELFORD, R. Eddie (essential distributed diagnostic and information engine). In *Proceedings of SAGE-AU'98* (Jul 1998), The System Administrators Guild of Australia.
- [28] MOHR, B., BROWN, D., AND MALONY, A. D. Tau: A portable parallel program analysis environment for pc++. In *CONPAR 94 - VAPP VI: Proceedings of the Third Joint International Conference on Vector and Parallel Processing* (London, UK, 1994), Springer-Verlag, pp. 29–40.
- [29] MOS, A., AND MURPHY, J. A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance* (New York, NY, USA, 2002), ACM Press, pp. 235–236.
- [30] MOSBERGER, D., AND JIN, T. httpperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.* 26, 3 (1998), 31–37.
- [31] MUCCI, P. J. *papiex: transparently measure hardware performance events of an application with PAPI*. Innovative Computing Laboratory, University of Tennessee, 2005. <http://icl.cs.utk.edu/mucci/papiex/>.
- [32] NAJAFZADEH, H., AND CHAIKEN, S. Towards a framework for source code instrumentation measurement validation. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance* (New York, NY, USA, 2005), ACM Press, pp. 123–130.
- [33] PARK, K., AND PAI, V. S. Comon: a mostly-scalable monitoring system for planetlab. *SIGOPS Oper. Syst. Rev.* 40, 1 (2006), 65–74.

- [34] QIN, M., LEE, R., RAYESS, A. E., VETLAND, V., AND ROLIA, J. Automatic generation of performance models for distributed application systems. In *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research* (1996), IBM Press, p. 33.
- [35] RENESSE, R. V., BIRMAN, K. P., AND VOGELS, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.* 21, 2 (2003), 164–206.
- [36] ROSE, L. D., AND REED, D. A. SvPablo: A multi-language architecture-independent performance analysis system. In *International Conference on Parallel Processing* (1999), pp. 311–318.
- [37] ROSE, L. D., ZHANG, Y., AND REED, D. A. SvPablo: A multi-language performance analysis system. *Lecture Notes in Computer Science 1469* (1998), 352–??
- [38] SHENDE, S., AND MALONY, A. D. Integration and applications of the tau performance system in parallel java environments. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande* (New York, NY, USA, 2001), ACM Press, pp. 87–96.
- [39] SPRUNT, B. Pentium 4 performance-monitoring features. *IEEE Micro* 22, 4 (Jul 2002), 72–82.
- [40] SUN MICROSYSTEMS. AWT: The Java abstract windowing toolkit. GUI development component of the Java 2, Standard Edition platform, 1999. <http://java.sun.com/products/jdk/awt>.
- [41] TEEGAN, H. Distributed performance monitoring using SNMP V2. *IEEE Network Operations and Management Symposium 2* (Apr 1996), 616–619.

- [42] TIERNEY, B., JOHNSTON, W., CROWLEY, B., HOO, G., BROOKS, C., AND GUNTER, D. The netlogger methodology for high performance distributed systems performance analysis. In *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 1998), IEEE Computer Society, p. 260.
- [43] TSOULOUPAS, G., AND DIKAIAKOS, M. Design and implementation of gridbench. *Lecture Notes in Computer Science* (2005).
- [44] TSOULOUPAS, G., AND DIKAIAKOS, M. D. Gridbench: A tool for benchmarking grids. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 60.
- [45] TSOULOUPAS, G., AND DIKAIAKOS, M. D. Characterization of computational grid resources using low-level measurements. Tech. Rep. TR-2004-05, Department of Computer Science, University of Cyprus, Oct 2004.
- [46] WYBRANIETZ, D., AND HABAN, D. Monitoring and performance measuring distributed systems during operation. In *SIGMETRICS '88: Proceedings of the 1988 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (New York, NY, USA, 1988), ACM Press, pp. 197–206.