

## **PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux**

Rick Kufrin

National Center for Supercomputing Applications  
University of Illinois at Urbana-Champaign  
[rkufirin@ncsa.uiuc.edu](mailto:rkufirin@ncsa.uiuc.edu)

### **Abstract**

The motivation, design, implementation, and current status of a new set of software tools called PerfSuite that is targeted to performance analysis of user applications on Linux-based systems is described. These tools emphasize ease of use/deployment and portability/reuse in implementation details as well as data representation and format. After a year of public beta availability and production deployment on Linux clusters that rank among the largest-scale in the country, PerfSuite is gaining acceptance as a user-oriented and flexible software tool set that is as valuable on the desktop as it is on leading-edge terascale clusters.

### **Introduction**

In the preface to his book, *Tcl and the Tk Toolkit*, John Ousterhout describes his motivations for developing the widely used Tcl scripting language as follows [1]:

Tcl was born of frustration. In the early 1980s my students and I developed a number of interactive tools at the University of California at Berkeley, mostly for integrated circuit design, and we found ourselves spending a lot of time building bad command languages. Each tool needed to have a command language of some sort, but our main interest was in the tool rather than the command language. We spent as little time as possible on the command language and always ended up with a language that was weak and quirky. Furthermore, the command language for one tool was never quite right for the next tool, so we ended up building a new bad command language for each tool. This became increasingly frustrating.

In the fall of 1987 it occurred to me that the solution was to build a reusable command language. If a general-purpose scripting language could be built as a C library package, then perhaps it could be reused for many different purposes in many different applications... In the spring of 1988 I decided to implement such a language, and the result was Tcl.

At the National Center for Supercomputing Applications (NCSA), as at many other research institutions in both the academic and corporate sectors, a similar situation arises in the context of performance analysis of software applications. A great deal of money is spent on acquiring, installing, and maintaining state-of-the-art high-performance computing (HPC) systems with the intent of achieving the greatest possible performance for mission-critical applications. Yet at the same time, the process of systematically gathering, analyzing, storing, and sharing data that measures the realized performance of these applications continues to occur largely in an *ad hoc* fashion, sometimes with great difficulty in even acquiring the relevant data. Results of analyses are frequently discarded or forgotten after generating a “scaling” or “optimization” plot or two after sufficient confidence is gained that performance has reached a satisfactory level given the available hardware, supporting libraries, and compiler technology. To paraphrase Ousterhout, application developers and performance analysts end up building a new bad infrastructure for acquisition and reuse of application performance data. It is not unusual to see everything built from scratch on an application-by-application basis.

A further unfortunate consequence of this haphazard approach to application performance analysis is that a substantial proportion of discipline-specific computational scientists *often do not know the actual performance characteristics of their production applications in daily use*. Frequently, nothing more than basic timing information (CPU or wall-clock) is used to assess whether or not an application makes efficient use of computational resources. While timing information represents the “bottom line” in terms of absolute performance, it offers little insight at the microarchitectural level into opportunities for performance improvement. One often hears of the growing gap between peak and achieved performance of real-world applications on the high-performance processors of today—it may be equally true that there is a growing gap in awareness of the underlying causes for performance deficiencies that adversely affect overall productivity.

This article describes an effort to help address this situation that has resulted in the design and creation of an infrastructure for application software performance analysis called PerfSuite. This infrastructure has benefited greatly from the innovation of others as well as from advances made in the development of underlying standards and portable, open source software (OSS) foundations that can reduce the time required to assemble the necessary components into a working system from years to months. In turn, end users of PerfSuite in many cases can achieve within minutes what might have previously taken days, weeks, or more to accomplish. From the outset, PerfSuite has been designed with the end user uppermost in mind: the thesis has been that application developers and users should not have to be burdened with becoming experts in the details and intricacies of low-level libraries that enable access to performance data. Ideally, developers should be empowered to quickly and easily receive accurate answers to fundamental questions such as these:

1. How well is an application performing?
2. How and where might efforts be best directed to improve an application’s performance?

PerfSuite has also been designed with a real-world perspective on the challenges often involved in software deployment on Linux systems today. Unfortunately, it is too often the case in the OSS community that, if an interested person sees potential in a particular software package, he or she soon discovers that the package has dependencies on several other independent packages, each in their own state of development and stability. In the extreme case, a person who simply wants to use the software for the perceived benefits finds himself or herself becoming a virtual member of the software development team. One could rightly argue that this is one of the primary reasons for the continuing success of the OSS community, however it also acts as a barrier to wider adoption by a broader audience. In fact, assembling disparate components into a turnkey, quality-controlled, validated, and supported complete system that requires minimal intervention by the consumer is a key reason for the successful business model pursued by Linux distributors such as Red Hat, SuSE, and others. While PerfSuite is, of course, very small in scope when compared to complete Linux distributions, it has been intentionally designed so that it has no dependencies on any of the supporting software it can use; this makes it possible to download, unpack, configure, and build a working version of PerfSuite that still can provide useful performance analysis data. If desired, the capabilities can be enhanced through external software support, but this is not a requirement.

## Background and Motivation

During the second half of the 1990s, the primary HPC systems available at NCSA were Power Challenge and Origin systems from Silicon Graphics, Inc. In addition to being stable, innovative hardware platforms, these systems offered some of the most advanced performance analysis software available, with the most frequently used components being the IRIX *perfex* command and SpeedShop tool set [2,3]. These two components provide the developer and performance analyst with mechanisms to extract performance data from unmodified applications and present the data in ways that help to quickly evaluate the actual performance of the application using hardware-based metrics. Independently and together, these provide substantial opportunities to learn of areas within an application that may be candidates for performance improvement.

In 2001, NCSA initiated a shift away from traditional shared-memory supercomputers such as the Origin in favor of Linux clusters of commodity microprocessors with high-speed interconnects, suitable primarily for

MPI-based applications or high-throughput “scavenger” applications that benefit from the highest possible CPU cycle availability. Shifts of this nature represent a sea change for users who are accustomed to single-source custom solutions as provided by HPC hardware vendors. Not only are the hardware components of the system commodity items, but also performance analysis software support - largely from the open source community with substantial reliance on the GNU system of the Free Software Foundation [4] - is typically limited to the venerable profiler *gprof*, developed in the early 1980’s as part of the BSD UNIX effort [5].

NCSA’s initial production Linux cluster offerings consisted of two architecturally-distinct systems: a 500+ dual-processor compute node 1GHz Pentium III (Coppermine) system, and a 150+ dual-processor compute node 800 MHz Itanium (Merced) system. Both clusters were configured with a high-speed Myrinet interconnect and ran then-current RedHat 7.x Linux distributions. These clusters entered production and were made available to NCSA’s user community in 2001 and 2002, respectively.

As a member of the technical staff within NCSA’s Scientific Computing Division with a focus on performance analysis, I was part of a center-wide effort to assist in the migration of user applications from the Origin supercomputers to the Linux clusters. Especially useful in this effort was evaluation of the performance of user applications in addition to identification of potential for performance improvement where possible. Unfortunately, an informal survey of the then-available supporting tools to assist in these tasks showed a relatively sparse set of offerings in comparison to those provided by IRIX. Simultaneously, I was involved in a development project to prototype a graphical interface for exploratory data analysis (EDA) using hardware performance counter data gathered on the Origin using *perfex*. The software resulting from this project was tied quite closely to the *perfex*-specific data format: one or more plain text output files listing raw counter values in addition to a pre-defined set of derived metrics that could be calculated from the event counts. It was clear that without *perfex* or equivalent support on the incoming Linux systems, the EDA capabilities under investigation would be restricted to the IRIX environment unless equivalent functionality became available for the Linux platforms. These were the primary reasons that PerfSuite development began; specifics of the implementation are discussed in the next section.

### **PerfSuite Building Blocks and Architecture**

PerfSuite’s design and implementation benefits greatly from ongoing developments in a number of areas that help to address the overall goals of the project. Specific requirements and related work that help to meet them include:

#### *Kernel Support for Performance Counter Access*

The IA-64 Linux kernel was designed from the outset to include support for performance monitoring capabilities that exploit the advanced capabilities of the Itanium performance monitoring unit (PMU). The Perfmon subsystem, authored by Stéphane Eranian of Hewlett-Packard Research, is responsible for driver implementation as well as a user-level library suitable for performance tool development [6]. For x86-based architectures, the Perfctr kernel patch, developed by Mikael Pettersson of Uppsala University, provides a performance counter driver that exposes similar capabilities to tools<sup>1</sup> [7].

#### *Portable, Standard Data Format and Supporting Software*

Historically, performance analysis software such as profilers defined their own internal (usually machine-specific) formats or simply presented the results of a performance analysis experiment directly to the standard output. Primary reasons for this include the desire for fast access as well as conservative use of disk space. Today’s computing systems, however, offer extremely fast disk access as well as capacities that are commonly in the gigabyte range. It is therefore much more feasible with these capabilities to take advantage of data formats that are more portable between machines. During the past decade, the XML standard has emerged as the format of choice for

---

<sup>1</sup> Unlike Linux/IA-64, the standard x86 kernel does not yet contain official support for Perfctr -- a kernel patch and rebuild is required. This is probably the biggest barrier to the adoption and installation of performance software that relies on it, including PerfSuite.

truly portable, self-describing data and has gained rapid acceptance in nearly all fields that rely on computing support [8,9].

#### *Portable, Extensible, Interpreted Programming Languages*

It was recognized at the outset that rapid development was key in being able to deliver the desired capabilities within a reasonable time and with a minimum of person-hours. Further, it was apparent that many of the activities essential for manipulation and presentation of performance data do not require the speed offered by a compiled, system-level programming language such as C or C++. Great strides have been made over the past fifteen years in scripting languages, with Perl, Python, and Tcl being the clear leaders and the choice among them largely a matter of personal preference. Efficient implementations coupled with increasingly faster processors make any of these languages a reasonable choice for implementing all but the most computationally demanding tasks. For access to features not directly supported by the language (such as interaction with performance counter drivers and user-level APIs), it is possible to extend the language through extensions written in a system programming language. Additionally, each of these languages provides some degree of support either directly or through extensions, for easy access to and manipulation of XML documents. Therefore, PerfSuite relies heavily on implementation using a scripting language (Tcl) and a robust extension for XML support (tDOM) wherever possible [10,11,12].

#### *Standard, Portable, Hardware Performance Event Definitions, API, and Implementation*

Hardware performance event counters are available on all major processors in common use today. However, there are substantial differences in the events available on each processor, even in different models within the same processor family. Further, programmatic access to the events has historically involved very low-level APIs, much too low-level to be of use to the typical application developer. One of the most important developments in the performance analysis community over the past several years has been an extended effort by the Innovative Computing Laboratory at the University of Tennessee-Knoxville that has resulted in a standard, cross-platform API for performance counter access. This API, known as the Performance Application Programming Interface (PAPI), is available for a number of current processors and operating systems (not just Linux) and is enjoying growing acceptance as a *de facto* standard for portable access to hardware performance counters [13].

Along with the definition and implementation of the API itself, the PAPI project also defines a set of "standard events deemed relevant and useful in tuning application performance". The existence of these standard events relieves the application developer and performance analyst of the burden of sifting through a sometimes overwhelming number of processor-specific event names and reduces these to a single common and related event name that is uniform across processors. For example, rather than having to know that total level 2 cache misses should be specified as "L2\_LINES\_IN" on Pentium 3, "L2\_MISSES" on Itanium, and the sum of "SYS\_DC\_REFILLS\_TOT" and "SYS\_IC\_REFILLS" on Athlon, the user need only specify "PAPI\_L2\_TCM" in their source code; the proper mapping is taken care of by the PAPI library. It is, however, important to recognize that the standard events provided by PAPI may have different semantics on different processors -- for the most effective insight, the user is advised to research the vendor documentation describing the exact meaning of each event for the processor of interest.

Another valuable feature of the PAPI library is support for performance counter multiplexing, based on the work of John May of Lawrence Livermore National Laboratory [14]. Multiplexing provides the ability to "timeslice" a larger number of performance events than would otherwise be possible to count simultaneously given a finite number of performance registers (two on Pentium III, four on Itanium). As a statistical technique, multiplexing sacrifices accuracy by estimating the true event count over the course of the run, but in general produces quite satisfactory results

especially for runs of reasonable length (the IRIX kernel, which also provides counter multiplexing support, suffers from the same limitation).

It is accurate to say that the existence of the PAPI library was the key item that enabled PerfSuite development to begin, as PAPI already supported the target platforms (Pentium III and Itanium) that were of immediate interest for use at NCSA.

Following preliminary research and experimentation, it was determined that each of these building blocks were either already well established or were in a sufficiently stable form to be usable for development of a prototype by the time the initial work on PerfSuite began in mid-2002. The earliest implementation consisted of a small library that contained two functions, callable from either C or Fortran. This library, conditionally compiled according to the target CPU, took care of the details of initializing the PAPI library, configuring a set of pre-selected PAPI standard events, enabling multiplexing support, starting event counting, and finally stopping the performance monitoring and writing the results to standard output. The library also featured several hard-coded "derived metrics" that could be calculated from the collected event data (for example, floating point operations per second, cycles per instruction, and various other ratios). Although useful as a quick proof of concept, the library still fell short of what experience showed would be necessary to be interesting and useful to a broader audience. In particular, it had the following shortcomings:

- Modification of source code, recompilation, and relinking required
- No control over event selection by the user
- Limited accuracy, due to "always-on" counter multiplexing
- No support for multithreaded applications
- Output format that was not amenable to downstream manipulation by machine
- No suspend/restart facility to allow monitoring of restricted portions of application
- Performance analysis limited to aggregate event counting; no support for event-based profiling
- Architectural features of the machine useful in calculation of derived metrics (for example, cache line size) were hard-coded into the library
- Library dependencies on supporting software; in the case of x86-based machines, this implied a requirement for a patched Linux kernel before it could be used

Despite these limitations, the library quickly proved to be useful by providing access to hardware-based performance data without requiring a substantial investment by the user. Subsequent work continued to focus on a general-purpose library and API as the foundation, rather than working from the perspective of any particular tool as the ultimate goal. This design choice has proven to pay dividends in terms of reuse of the software for multiple needs.

Another important consideration in the redesign following the prototype implementation was the selection of format and content, both input and output. Substantial prior experience in working with production applications *and the user communities that develop and use them* on both shared- and distributed-memory parallel systems made it clear that it was necessary to look past single-use scenarios for the performance data collection process.

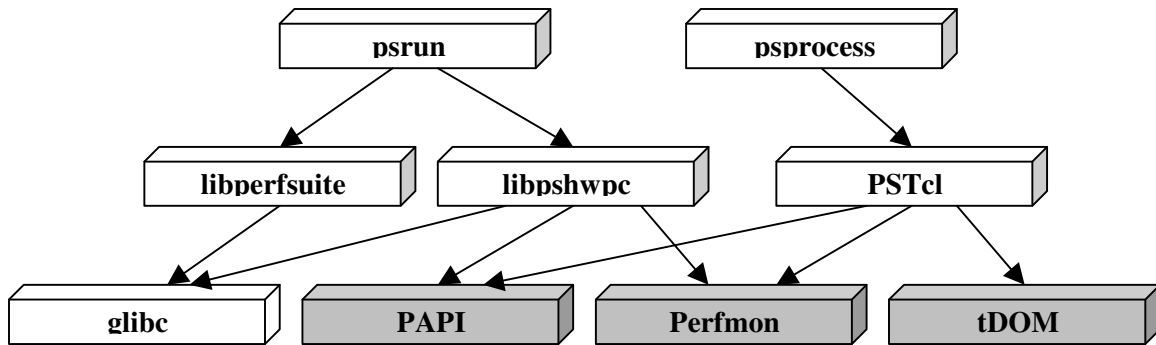


Figure 1. PerfSuite architecture.

Over the course of several months in late 2002 and early 2003, a more general-purpose library and tool infrastructure was designed and written. This infrastructure, which remains in place in current versions of PerfSuite, is shown in Figure 1. Shaded components in this figure correspond to optional external support. In the next section, each core component of PerfSuite is reviewed.

### Descriptions of Libraries and Tools

In order to address the limitations of the initial prototype, to address the desire for flexibility/portability, and to remove dependencies on external software support, the original library interface to PAPI was split into two separate libraries, with one - `libperfsuite` - implementing base functionality that could operate regardless of the presence of external software and the other - `libpshwpc` - serving as a generic gateway to common features that could be provided by specific lower-level access to performance measurements.

Along with this redesign, a very high-level and simple API specification was implemented. This API has nothing inherently specific to a particular type of performance measurement and in spirit most closely reflects action-oriented tasks that a user might wish to initiate, such as “begin measurement at point A”, “suspend measurement at point B”, and “stop measurement at point C”. To some extent, these actions take an object-oriented approach by deferring the details of *what* is measured until runtime and instead only specifying a performance measurement framework that isn’t fully instantiated until coupled with a specific configuration.

In addition, the redesign standardized on the XML format as the common representation for both input and output of the libraries and tools, and XML applications were created in support. The functionality and flexibility that this enables is demonstrated in later sections.

### libperfsuite

Despite the fact that hardware-based performance data access is central to PerfSuite software, it is still the case that there are a number of useful pieces of information and measurements interesting to the performance analyst that are independent of performance counters. Examples include access to accurate, high-resolution timers, information about architectural features such as details of the memory hierarchy, and resource usage information such as CPU time consumed or the resident set size of a running application. Several of these measurements are either not available or not implemented in versions of the Linux kernel or supporting libraries. For example, many developers are often surprised that the system call `getrusage()`, as available in standard Linux distributions, does not return meaningful values for memory usage. `libperfsuite` provides access to this type of information that can be retrieved directly by user programs and is also used by tools such as `psrun` (described below). `libperfsuite` is a standalone library that is fully functional on any supported platform, regardless of the presence of other support libraries<sup>2</sup>.

<sup>2</sup> It is assumed that the GNU C library is available.

## libpshwpc

As in the earlier prototype for library-accessible control of performance monitoring, a primary focus of the current version of PerfSuite is hardware performance event data. This is the role of libpshwpc (the library's name stands for PerfSuite HardWare Performance Counters). The library exposes to the user and to PerfSuite tools the following small set of functions:

- `int ps_hwpc_init(void)` initialize library
- `int ps_hwpc_start(void)` start measurement
- `int ps_hwpc_suspend(void)` pause measurement
- `int ps_hwpc_read(ps_hwpc_value_t *values)` return current performance data
- `int ps_hwpc_stop(char *filename)` stop measurement and write XML data
- `int ps_hwpc_shutdown(void)` end use of library
- `int ps_hwpc_numevents(int *numevents)` return current number of counters used
- `int ps_hwpc_eventnames(char ***eventnames)` return current event names
- `int ps_hwpc_psruntime(void)` start *psrun*-like collection

The above prototypes are callable from C-based languages and return a code indicating the success or failure of the call. Fortran subroutine versions are implemented that include an additional argument (the conventional “ierr” return status code).

It may be apparent on examination of the libpshwpc API that there are no parameters passed to the routines indicating anything specific about the performance experiment to be performed. Instead, this information is deferred until runtime and is determined through a configuration file that is provided as an XML document.

Extensions to libpshwpc also add support for gathering profiling data using statistical sampling techniques. Sampling can be done in a number of different ways:

1. Through the standard `profil()` subroutine available in the GNU C library. This is the same technique used by the *gprof* tool and compilers that support generation of *gprof*-compatible data. The sampling strategy involves employing interval timers to deliver SIGPROF interrupts to the application at fixed intervals (10ms on x86 systems, 1ms on IA-64 platforms). Unlike *gprof* however, libpshwpc also supports the sampling of shared library data.
2. Through direct use of the `setitimer()` system call, with interrupt handlers installed by PerfSuite. This technique provides a superset of the functionality provided by `profil()` in two ways: the user can select alternate time domains in addition to `ITIMER_PROF`, which is decremented when the process is executing in both user and system mode. The two alternate domains are `ITIMER_VIRTUAL` (decremented when executing in user mode only) and `ITIMER_REAL` (decremented in real time, regardless of the execution mode of the process). The `ITIMER_REAL` option in particular can provide a substantially different viewpoint of execution over the lifetime of an application as it may include samples collected at times that the process was waiting for events such as I/O activity or message-passing to be completed.
3. By hardware event-based support, which represents a generalization of the time-based techniques above. The performance counters on all current general-purpose processors that Linux supports provide an option to generate an interrupt after a specified number of

occurrences of one or more hardware events, which is handled in much the same manner as traditional interrupts on timer expiration. The value of this type of profiling lies in the ability to obtain profiles that pinpoint the locations within the application where the most occurrences of events of interest, including cache misses, processor stalls, floating point operations/assists, vector operations, and so on.

A final extension to the prototype implementation of libpshwpc is the ability, through the `ps_hwpc_suspend()` and `ps_hwpc_start()` functions, to pause and restart performance measurement (whether counting or profiling) in order to isolate one or more regions of an application.

## psrun

*psrun* is a command-line utility that enables performance measurement of unmodified dynamically linked applications. From the user perspective, it is probably the most visible and frequently used component of PerfSuite. *psrun* combines the functionality available through libperfsuite and libpshwpc with operating system support for runtime loading of shared libraries to perform its work. Conceptually, *psrun* is straightforward; when invoked to measure the user's application (e.g., `psrun a.out`), the following occurs:

1. The command line supplied by the user is parsed by the *psrun* program, which also verifies any options specified and identifies the executable to be measured. Options that tailor the performance measurement for individual runs are communicated to libpshwpc through environment variables; in most cases, UNIX-style command line flags (e.g., `-o outputprefix`) are translated by *psrun* to their equivalent environment variable setting. Alternatively, the user can choose to set the appropriate environment variable directly, avoiding the need to use *psrun* command-line options.
2. A "helper" shared library - libpsrun - is added to the environment by *psrun*, specified through the LD\_PRELOAD environment variable. On Linux systems, LD\_PRELOAD can be used to cause one or more shared libraries to be loaded before all others<sup>3</sup>. The LD\_PRELOAD technique is commonly used to selectively override one or more functions contained in standard shared libraries (for example, one might wish to trap calls to the `malloc()` memory allocator to assist in tracking or debugging memory usage or locating memory leaks). There are two variants of libpsrun available; the proper version is chosen based on whether the program to be measured is single- or multi-threaded. To accommodate multithreaded applications, libpsrun includes a function that overrides `pthread_create()` in order to allow measurement of the performance of individual threads.
3. *psrun* creates a new process and spawns the user's requested executable via a `fork()/exec()` pair. Optionally, a "monitor thread" may also be created on user request; this thread has the responsibility of periodically querying resource usage of the newly created process as well as monitoring other runtime factors that may affect performance such as the system load average.
4. The user's process begins execution with libpsrun loaded (due to the prior setting of LD\_PRELOAD). libpsrun contains a distinguished *constructor function* that is called automatically before the user's program enters its `main()` routine<sup>4</sup>. The constructor function has the responsibility of loading libpshwpc into the user's process and starting performance measurement by calling `ps_hwpc_init()` and `ps_hwpc_start()`.

<sup>3</sup> Functionality similar to LD\_PRELOAD is often available on other operating systems and libraries, possibly through a different name or mechanism. For example, on IRIX platforms, the variable `_RLD_LIST` or several variants may be used.

<sup>4</sup> The current implementation of libpsrun uses "function attributes", recognized by the GNU *gcc* compiler. Alternate approaches are possible, such as the use of the distinguished library entry point `_init()`.



The constructor also arranges for performance data to be written out before the user's process exits by registering `ps_hwpc_stop()` as an "exit handler" that is called automatically when the user's process exits normally<sup>5</sup>.

5. Lastly, *psrun* waits for the user's process to terminate using the `wait4()` system call. If resource monitoring has been requested, *psrun* issues a cancellation request to the monitor thread, causing it to write out the resource data collected prior to honoring the cancellation.

It is important to note that, with the exception of resource data collection by the monitor thread, *psrun* neither collects nor writes any data from the performance measurement. Instead, this is performed by individual threads within a process through preloaded shared libraries described above.

### **psprocess**

The output of `libpshwpc`, whether generated by an application incorporating direct calls to the library or through *psrun*, is an XML document containing the raw performance data collected in addition to information describing machine characteristics. While this information can be useful by itself and is human-readable due to the textual nature of XML data, in most cases it does not present the data in a form that is immediately useful to the average developer or performance analyst. Instead, humans tend to prefer data that is expressed in a *normalized form* that lends itself to comparative use. For example, rather than learning that a key computational kernel executed 86,124,489,172 floating point operations in 594,547,754,568 cycles on a system clocked at 800 MHz, it is much more straightforward to grasp that the kernel executed at a rate of approximately 116 million floating point operations per second (MFLOPS). This type of statistic is often referred to as a "derived metric" that can be calculated using raw measurements, possibly combined with information about the machine characteristics (such as the clock speed in this example). Given the wide variety of hardware performance events that are tracked by current microprocessors, the number of possible metrics that can be derived from arbitrary measurements is potentially staggering; however, in many cases only a few metrics are of interest for a particular combination of hardware, software, algorithm, and data. Still, it is desirable to maintain as much flexibility as possible in terms of what metrics are *possible and supported*.

A similar need arises in the case of statistical profiling of applications. At the lowest level, sampling strategies simply record the value of the system's program counter at regular intervals. This value is literally a specific numeric address within the text space of the machine instructions. Clearly, learning that the majority of profiling samples collected during a particular experiment correspond to, e.g., address 0x4001ce70 is not of great use to developers, who instead think in terms of source code lines, functions, and files. However, machine addresses are exactly what is collected by sampling and therefore are what is deposited in the XML documents written by `libpshwpc`, so that a subsequent mapping of addresses to program locations is necessary to be useful.

Transformation of raw data collected and written by `libpshwpc` into human-comprehensible form is one of the primary tasks performed by the tool *psprocess*. In terms of the complexity of the work performed by various components of PerfSuite, *psprocess* and software packages that it uses are by far the most complex. However, *psprocess* is also surprisingly compact and manageable, with the current version written in approximately 2,000 lines of code. This is certainly due to the choice of a high-level scripting language (the implementation language of *psprocess* is Tcl). *psprocess* relies on core Tcl features and in addition makes extensive use of the tDOM XML package and Tcl-based interfaces to hardware performance counter libraries that were developed as part of PerfSuite, primarily for this purpose.

---

<sup>5</sup> One of `onexit()`, `atexit()`, or a `gcc` destructor is used to register the process exit handler.

Examples of the use of *psprocess* include:

- Summarizing single-processor aggregate performance event data and calculating derived metrics (standard or user-defined) from the raw event counts
- Producing understandable reports from statistical profiling sample counts, using a Tcl extension that is part of PerfSuite for mapping program text addresses to source code lines, functions, and files
- Combining individual performance measurements into “multi-reports”, representing a logical collection of measurements (as might be generated, for example, by a run of a parallel program), and further processing these multi-reports to display appropriate descriptive statistics for the collection such as mean, maximum, minimum, variance
- Translating data collected by PerfSuite to formats specific for external tools such as *gprof* or *VProf* [15]
- Using a prior counting run to estimate a reasonable sampling period for a subsequent profiling experiment with a particular application (referred to as *calibration*)
- Retrieving PerfSuite-generated XML data over the Internet via the HTTP protocol and post-processing it locally (HTTP support is well-integrated with the standard Tcl distribution)

*psprocess* adapts its behavior according to the type of XML document supplied as well as to options supplied by the user. For common cases, the user needs only to invoke *psprocess*, supplying the name of the input XML document, and *psprocess* will “do the right thing”. This “natural” mode of use has a substantial benefit in reduction of time required to learn to use *psrun* and *psprocess* to obtain useful performance information.

#### **psinv**

Users of HPC systems, especially those conducting performance analysis work, often find it helpful or necessary to determine machine characteristics such as CPU type, architectural details such as cache line sizes, installed memory, available hardware performance events, and so on. There is currently no UNIX “standard” command that provides this type of information, although there have been a number of useful utilities developed from time to time, such as *sysinfo* or *hinv*. These types of programs provide helpful high-level expectations for application performance on a given system, especially within environments where multiple systems are available. Additionally, low-level optimization strategies that tailor software behavior for particular machine characteristics (e.g., cache line size) can benefit from having this information readily accessible. *psinv* is a small utility that exposes this type of information for easy access. It relies on *libperfsuite* and any installed supporting libraries and provides a quick inventory of performance-related aspects of the system.

### **Examples of Use**

#### **Basic Event Counting**

The libraries and tools described in the prior section provide a foundation for routinely obtaining performance measurements of unmodified applications, either through user-directed experiments or by an automated process that can be tailored as required to meet the needs of a specific site. In either usage model, the underlying mechanisms are identical. The particular measurements to be collected are contained in an XML *configuration file* that is specified through either a command-line option to *psrun* or through the environment. If no configuration file is supplied, then a default configuration is used that collects aggregate event counts over the course of the application run. This mode of operation is equivalent to the IRIX *perfex* command. The user invokes *psrun* directly on the executable and arguments supplied:

```
% psrun myprogram
```

If the application terminates normally, then the output of *psrun* is written to an XML document contained in a disk file; the document name conforms to a standard template that includes the name of the executable, the process ID, and the hostname of the computer on which the program ran<sup>6</sup>.

Figure 2 shows an example of the type of output collected by *psrun* in counting mode, when post-processed with *psprocess*. This example has been considerably abbreviated and shows only a portion of the machine information displayed, the performance events that are collected, and the derived metrics that are calculated by PerfSuite.

### **Multithreaded Applications**

The procedure for using PerfSuite with multithreaded applications is nearly the same -- the only difference is in the inclusion of a command-line flag “-p” that requests tracking of individual POSIX threads:

```
% psrun -p threadedprogram
```

In the threaded case, the naming template for the XML output documents include an additional component: a unique thread ID (generated internally by libpshwpc).

Although historically, Linux systems have tended to be single-CPU workstation-class systems, a great deal of progress has been made over the past several years to support symmetric multiprocessor (SMP) platforms. With improvements to key areas such as the scheduler and POSIX thread support, current systems such as the SGI Altix have demonstrated successful scaling to hundreds of processors. Application developers remain keen to take advantage of the potential improvements through shared-memory (thread-based) parallelism, and the standard in SMP parallel programming models is the OpenMP standard [18]. A majority of the currently-available commercial and open source compilers that support OpenMP rely on the POSIX thread library for thread management, with the result that the thread support contained in PerfSuite maps well to OpenMP-based applications. It is important to note, however, that there is no “awareness” of OpenMP constructs within PerfSuite; this implies that to a great extent tools like *psrun* are at the mercy of the thread management of the runtime libraries. If the OpenMP implementation chooses to reuse existing threads in ways that do not directly correspond to the *application’s* expression of parallelism, this will be reflected in the resulting XML output.

### **Profiling**

By supplying a non-default configuration file, the user can request that statistical profiling is used in the performance experiment. In this case, only a single measurement is currently supported (either a specific hardware event or one of the other time-based sampling techniques discussed earlier). Figure 3 shows an example of the type of output that can be obtained using this measurement technique. Although abbreviated, the example contains representative output; in particular, it shows that the current implementation is restricted to “flat” profiles that simply record the frequency of samples attributed to each sampled address. No contextual information is gathered that allows the generation of call graph (or call tree) displays. In practice, we find that flat profiles provide the first level of information usually of interest to developers and performance analysts, however it may be useful to examine flat profiles in conjunction with call graph data generated using other available means to obtain the most complete profiling view.

### **Custom Configuration Files**

Figure 4 shows an example of a custom configuration file that can be used for aggregate counting using direct access to the Perfmon subsystem on IA-64 platforms. This example demonstrates how one might go

---

<sup>6</sup> Early versions of PerfSuite did not include the hostname. It was quickly discovered that clashes of process IDs in a clustered environment would cause performance data to be overwritten, so the hostname was included to avoid this loss of data.

about counting *useful graduated instructions per cycle* on this processor. Due to the nature of the EPIC architecture, the compiler can generate a substantial number of `nop` instructions in order to fill out available slots for which no other instruction could be scheduled. The IA-64 PMU includes these instructions along with other (presumably “useful” instructions) in the final event counts. Therefore, if one calculates a metric such as CPI (cycles per instruction) with no-ops included, the result may appear inflated. This PerfSuite configuration file requests both total IA-64 retired instructions and retired no-ops; the difference represents the number of retired instructions spent performing useful work. This type of measurement is not currently supported when restricted to PAPI standard events, for which no-ops are not defined, so it is convenient to drop to the native processor events for finer control (note that PAPI does support processor native events; this example only refers to the collection of PAPI standard events).

### **User-Defined Metrics**

Figure 5 shows an example of defining a derived metric that can be used by *psprocess* when post-processing counting data. Definitions such as these may be grouped into an XML document and provided via the command line when issuing the *psprocess* command. The syntax used is reminiscent of the MathML XML standard, although not strictly compliant. PerfSuite supports simple arithmetic combinations of hardware event counts that are evaluated by *psprocess* at runtime, using the supplied event data. This example also shows the use of metric “descriptions”, which support internationalization through Tcl.

### **Interoperability with External Tools**

In PerfSuite development efforts to date, little or no attention has been paid to graphical display of performance data gathered by the libraries and tools, both by design and by necessity. As mentioned early in this document, the primary motivating factor for development of PerfSuite under Linux was to replace functionality lost while transitioning from single-vendor HPC solutions to systems based on commodity microprocessors and open source software. It is clear that development of a comprehensive set of tools for performance measurement, analysis, and visualization is an effort that can potentially consume substantial resources and is well beyond the scope of PerfSuite development to date. This is, in itself, sufficient reason to continue to restrict development to core data gathering and basic text-based display.

Another reason to restrict development is subtler: development of graphical techniques for information visualization is a complex and active field of research in its own right. One of the tradeoffs that are often encountered when using non-generic graphical tools tailored to a specific domain is that the developers, who produce the tools, by necessity, make a number of choices *for you*. Although the display techniques may be state-of-the-art and extremely innovative, in many cases the user of the tool has limited control over the nature of the visualizations produced.

In order to avoid being “trapped” by unwise, ill-designed, or unmaintainable decisions regarding graphical display, the focus centers instead on careful design of extensible data containers that lend themselves to transformations that are acceptable to existing display technologies. The choice of XML as the standard container for PerfSuite-generated performance data has paid repeated dividends in this regard. XML technologies such as Extensible Stylesheet Language Transformations (XSLT) and the ready availability of high-quality XML parsers greatly simplify the task of transforming PerfSuite’s raw XML documents into formats that are usable by existing tools.

Figure 6 shows two examples of useful tools that may provide insight into performance data collected by *psrun* and filtered by *psprocess*. The upper display is generated by a tool for exploratory data analysis called *Attribute Explorer*, a joint development effort by researchers at Imperial College and IBM [17]. Shown side-by-side is performance data gathered from two runs of a parallel (MPI-based) application. Each “box” in the main window displays a collection of performance metrics as calculated by *psprocess*. Within each box, data points corresponding to each metric are discretized (binned) and displayed on a per-processor basis within smaller “cells” positioned within the box and plotted as column-oriented histograms.

Further, the coloring of each cell indicates which run is the source of the measurement. For the experiment shown, the only difference between runs was the number of processors active and taking part in the calculation within each SMP node of a Linux cluster (alternatively stated, the graphic compares performance data captured for both 1-processor-per-node and 2-processors-per-node runs of an identical application). The value of this type of display is the potential to easily distinguish patterns contained within a relatively large number of observations (over 1000 data points are displayed simultaneously in the figure). Patterns of interest might include those for which there is a clear visual separation between individual runs, indicative of generally distinct performance characteristics. Conversely, boxes in which colors are uniformly distributed across the range of measurements are likely to correspond to metrics that do not distinguish between multiple experiments (runs). This type of display can also be valuable a wide range of comparative studies, such as multiple applications, or different system software or hardware configurations.

The lower display in Figure 6 shows a more traditional source-code browsing capability: it was generated by the *vprof* tool developed at Sandia National Laboratories. *vprof* is part of a larger, open source, package for performance data collection and visualization that predates and has several features in common with PerfSuite. The *vprof* tool was one of the earliest Linux-based performance tools to see general use at NCSA and continues to enjoy popularity, owing to its ease of use and flexible presentation styles (text-based, HTML, and graphical interfaces are available). The display in Figure 6 shows data that was collected by PerfSuite visualized by *vprof*. Although the *vprof* 0.12 format cannot handle the full range of profiling data that can be collected by PerfSuite (in particular, profiling data from shared libraries is not supported), it is still quite useful and convenient as an easy method of reviewing profiling data correlated to the source code. The interoperability of PerfSuite and *vprof* was achieved by adding a module to *psprocess* that translates PerfSuite-generated profiling data into the *vprof* binary equivalent. The module is quite compact, requiring only about 100 lines of Tcl to perform the translation. A similar capability has also been implemented in *psprocess* to generate the “gmon.out” format understood by the GNU *gprof* profiling utility. Both *vprof* and *gprof* support are included in the standard PerfSuite distribution.

### **Generation of Custom, On-Demand Displays**

Another factor that influences the decision to emphasize core data collection facilities over extensive GUI development is the rapid pace of advancement of libraries for graphical display. It is no longer the case that in-depth knowledge of the inner workings of earlier libraries such as Motif is prerequisite to GUI development. Indeed, the scripting languages of today (not only on UNIX-based systems, but also Windows) provide ample support for quickly producing “one-off” displays with very little effort. While this type of RAD development will not result in polished, commercial-quality, general-purpose tools, it could be argued that day-to-day use of performance data may only infrequently require extensive graphical investigation.

Figure 7 shows an example of this type of rapid-development display, showing a Tcl/Tk script that can be used to generate bar and pie charts using an open source pure Tcl package called Plotchart and the tDOM Tcl XML extension. Using fewer than 20 statements, the script can plot rudimentary but serviceable charts that can be useful to developers involved in optimization and performance analysis on a daily basis. Anyone who has worked in an iterative mode, perhaps adjusting compiler flags or inserting directives, recompiling, rerunning and checking performance at each step will likely appreciate the value of this mode of operation, which can be entirely sufficient for “quick and dirty” performance tuning sessions.

Although Tcl/Tk and associated support packages are used in the example, it’s likely that similar functionality may be found within the communities associated with each of the popular interpreted languages available today. By combining the rapid development offered within these languages with convenient performance data generation as available in PerfSuite, the ultimate goal of easy-to-use, robust, scalable, and comprehensible support for performance analysis will soon be commonplace, not only within the HPC world, but across the spectrum of development efforts targeted to open source based platforms.

|   |   |
|---|---|
| PerfSuite Hardware Performance Summary Report                           |   |
| Version   | : 1.0   |
| Created   | : Mon Dec 30 11:31:53 AM Central Standard Time 2002 |
| Generator   | : psprocess 0.5                                     |
| XML Source  | : /u/ncsa/anyuser/performance/psrun-ia64.xml        |
| Processor and System Information  |   |
| =====   |   |
| Node CPUs   | : 2   |
| Vendor  | : Intel   |
| Family  | : IPF   |
| Model   | : Itanium   |
| CPU Revision  | : 6   |
| Clock (MHz)   | : 800.136   |
| Memory (MB)   | : 2007.16   |
| Pagesize (KB)   | : 16  |
| Cache Information   |   |
| =====   |   |
| Cache levels  | : 3   |
| -----   |   |
| Level 1   |   |
| Type  | : data  |
| Size (KB)   | : 16  |
| Linesize (B)  | : 32  |
| Assoc   | : 4-way set associative                             |
| Type  | : instruction                                       |
| Size (KB)   | : 16  |
| Linesize (B)  | : 32  |
| Assoc   | : 4-way set associative                             |
| Index Description   | Counter Value                                       |
| =====   |   |
| 1 Conditional branch instructions mispredicted.....                     | 4831072449  |
| 2 Conditional branch instructions correctly predicted.....              | 52023705122   |
| 3 Conditional branch instructions taken.....                            | 47366258159   |
| 4 Floating point instructions.....                                      | 86124489172   |
| 5 Total cycles.....   | 594547754568  |
| 6 Instructions completed.....   | 1049339828741                                       |
| Statistics  |   |
| =====   |   |
| Graduated instructions per cycle.....                                   | 1.765   |
| Graduated floating point instructions per cycle.....                    | 0.145   |
| % graduated floating point instructions of all graduated instructions.. | 8.207   |
| Graduated loads/stores per cycle.....                                   | 0.219   |
| Graduated loads/stores per graduated floating point instruction.....    | 1.514   |
| Mispredicted branches per correctly predicted branch.....               | 0.093   |
| Level 3 cache line reuse (data).....                                    | 2.498   |
| Bandwidth used to level 3 cache (MB/s).....                             | 385.087   |
| % cycles with no instruction issue.....                                 | 10.410  |
| % cycles stalled on memory access.....                                  | 43.139  |
| MFLOPS (cycles).....  | 115.905   |
| MFLOPS (wallclock).....   | 114.441   |
| MIPS (cycles).....  | 1412.190  |
| MIPS (wallclock).....   | 1394.349  |
| CPU time (seconds).....   | 743.058   |
| Wall clock time (seconds).....  | 752.566   |
| % CPU utilization.....  | 98.737  |

Figure 2. Example PerfSuite output (counting run)

```

Profile Information
=====
Class      : PAPI
Event      : PAPI_TOT_CYC
Threshold  : 1000000
Samples    : 5254005
Domain     : user
Run Time   : 6872.56 (wall clock seconds)
Min Self % : (all)

File Summary
-----
Samples    Self %   Total %   File
2538551    48.32%   48.32%   /home/perfsuite/apps/gromacs/gromacs-3.1.4/src/gmxlib/innerf.f
865132     16.47%   64.78%   /home/perfsuite/apps/gromacs/gromacs-3.1.4/src/mdlib/pme.c
850101     16.18%   80.96%   /home/perfsuite/apps/gromacs/gromacs-3.1.4/src/mdlib/ns.c

Function Summary
-----
Samples    Self %   Total %   Function
1839543    35.01%   35.01%   inl3130
541829     10.31%   45.32%   ns5_core
389741     7.42%    52.74%   inl0100

File:Line Summary
-----
Samples    Self %   Total %   File:Line
232686     4.43%    10.70%   /home/perfsuite/apps/gromacs/gromacs-3.1.4/src/mdlib/pme.c:269
174262     3.32%    14.01%   /home/perfsuite/apps/gromacs/gromacs-3.1.4/src/mdlib/ns.c:1274
99352      1.89%    15.91%   /home/perfsuite/apps/gromacs/gromacs-3.1.4/src/mdlib/ns.c:1490

```

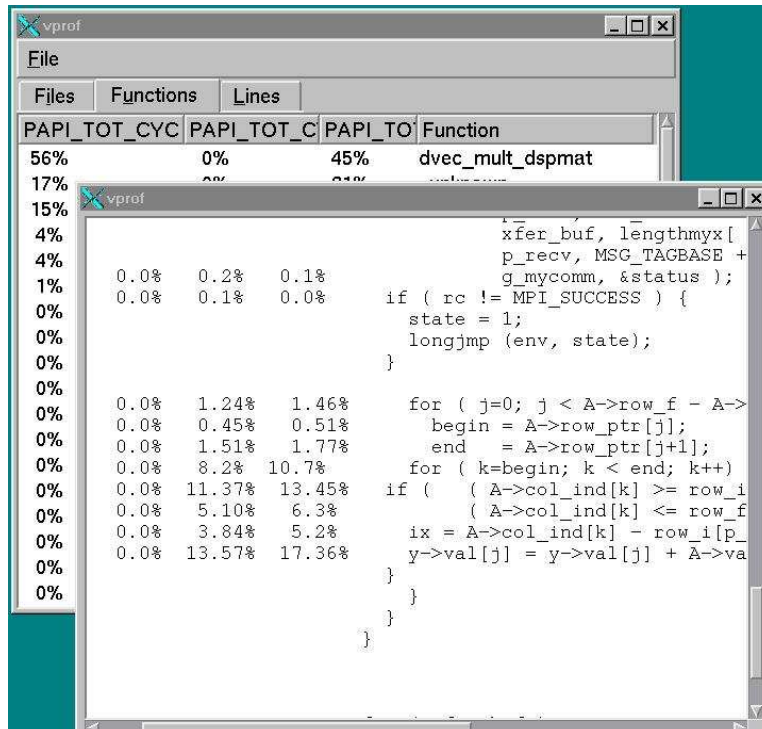
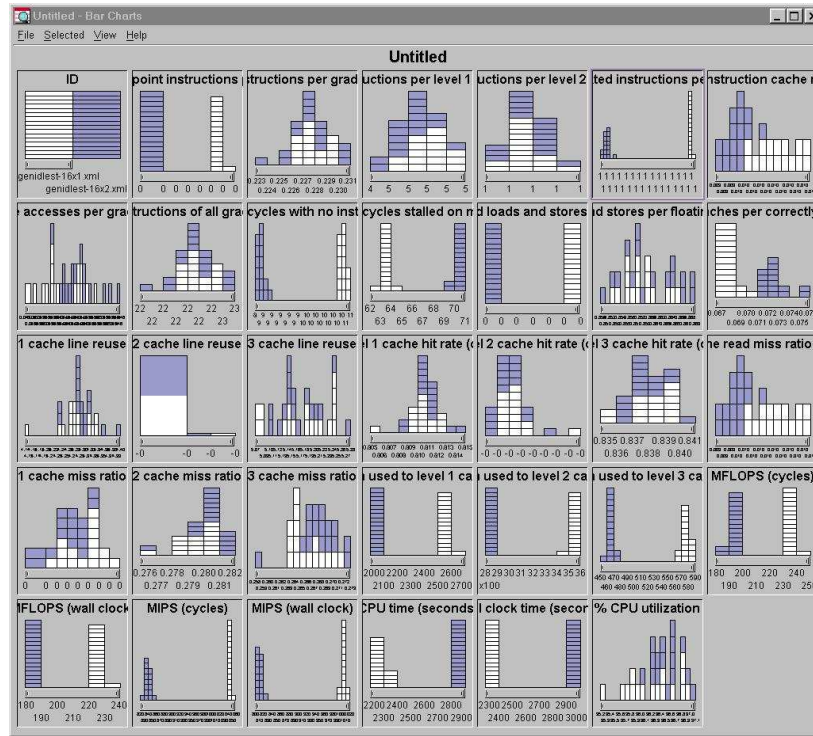
Figure 3. Example PerfSuite output (profiling run)

```
<ps_hwpc_eventlist class="perfmon">
  <ps_hwpc_event type="preset">IA64_INST_RETIRED_THIS</ps_hwpc_event>
  <ps_hwpc_event type="preset">NOPS_RETIRED</ps_hwpc_event>
  <ps_hwpc_event type="preset">CPU_CYCLES</ps_hwpc_event>
</ps_hwpc_eventlist>
```

Figure 4. Example XML configuration document (counting)

```
<psmetrics class="hwpc">
  <metric namespace="PAPI" type="ratio">
    <name>PS_RATIO_GFPINS_CYC</name>
    <description lang="en_US">Graduated floating point instructions per cycle</description>
    <definition>
      <apply>
        <divide>
          <ci>PAPI_FP_INS</ci>
          <ci>PAPI_TOT_CYC</ci>
        </divide>
      </apply>
    </definition>
  </metric>
</psmetrics>
```

Figure 5. Example derived metric definition



**Figure 6. Two examples of display techniques of external tools using PerfSuite-generated data. The upper display is produced by Attribute Explorer; the lower display by VProf.**



```

# Acquire the total number of samples during the run.
set totalsamples [[$xmldoc getElementsByTagName totalsamples] text]
set other 0

# Collect samples on a function-by-function basis.
foreach element [[$xmldoc getElementsByTagName function] {
  set fname [$element getAttribute name]
  set count 0
  # Accumulate samples attributed to this function.
  foreach sample [$element childNodes] {
    incr count [$sample text]
  }
  # Restrict display to no more than ten functions, and only
  # those containing greater than five percent of all samples.
  # Place all other samples in "Other" category.
  set pct [expr {round($count/double($totalsamples) * 100.0)}]
  if { ($pct >= 5) && ([llength pct] < 10) } {
    lappend pct $fname $pct
  } else {
    incr other $pct
  }
}

# Incorporate "other" samples and plot the results.
lappend pct "Other" $other
canvas .c -bg white

# The following two lines determine the chart type, here a pie chart.
set chart [Plotchart::createPiechart .c]
$chart plot $pct
$chart title [file tail $filename]

pack .c

```

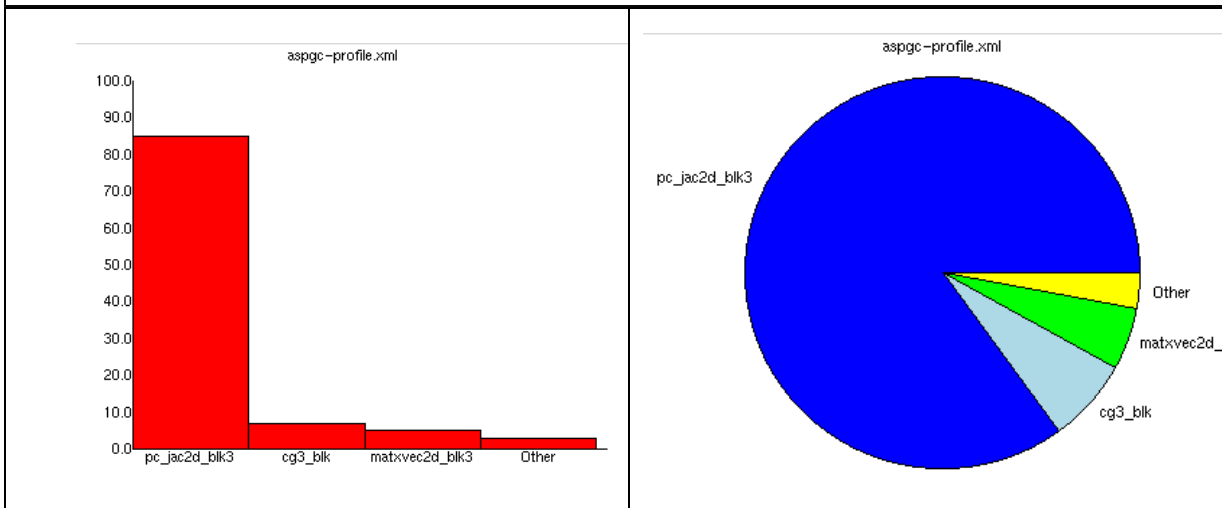


Figure 7. Example plotting script and generated displays

## Acknowledgements

PerfSuite's evolution has been substantially accelerated through the assistance, feedback and support from numerous people, to whom I owe a debt of thanks. The PAPI development team at the Innovative Computing Laboratory at the University of Tennessee-Knoxville has been a great source of assistance and encouragement (the PAPI library itself represented the earliest supporting software on which PerfSuite was layered) - Phil Mucci of ICL has had a particular influence on PerfSuite development and has offered consistent advice and support as the project has evolved. At NCSA, the efforts of the HPC systems and consulting staff have been invaluable for early testing and ongoing troubleshooting and diagnosis as PerfSuite was and continues to be deployed on large-scale production systems with a diverse user community. Greg Bauer has been especially helpful in early testing and feedback/suggestions for improvement of the PerfSuite libraries and tools. Finally, thanks are due to John Towns (the earliest supporter of PerfSuite), who is largely responsible for creating a productive environment in which ideas are encouraged to become realities.

## References

1. Ousterhout, J. *Tcl and the Tk Toolkit*. Reading: Addison-Wesley, 1994.
2. Zagha, M. et al. *Performance Analysis using the MIPS R10000 Performance Counters*. In Proceedings of Supercomputing 1996. IEEE/ACM, 1996.
3. Silicon Graphics, Inc. *SpeedShop User's Guide*. Mountain View: SGI, 2003.
4. Free Software Foundation. The GNU Operating System. <http://www.gnu.org/>
5. Graham, S., Kessler, P., and McKusick, M. *gprof: A Call Graph Execution Profiler*. In Proceedings of ACM SIGPLAN Symposium on Compiler Construction. ACM, 1982.
6. Eranian, S. Perfmon Project web site. Hewlett Packard Labs, <http://www.hpl.hp.com/research/linux/perfmon/>
7. Pettersson, M. Perfctr web site. Uppsala University, <http://www.docs.uu.se/~mikpe/linux/perfctr/>
8. Harold, E. and Means, W. *XML In A Nutshell*. Sebastopol: O'Reilly, 2004.
9. World Wide Web Consortium. *Extensible Markup Language (XML)*. <http://www.w3.org/XML/>
10. Tcl Core Team. Tcl/Tk web site. <http://www.tcl.tk/>
11. Löwer, J. *tDOM - A fast XML/DOM/XPath package for Tcl written in C*. In Proceedings of First European Tcl/Tk User Meeting. 2000.
12. Löwer, J. and Ade, R. tDOM web site. <http://www.tdom.org/>
13. Innovative Computing Laboratory. PAPI web site. University of Tennessee-Knoxville, <http://icl.cs.utk.edu/projects/papi/>
14. May, J. *MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs*. In Proceedings of 2001 International Parallel and Distributed Processing Symposium. IEEE, 2001.
15. Janssen, C. VProf web site. Sandia National Laboratories, <http://aros.ca.sandia.gov/~cljanss/perf/vprof/>
16. Mosberger, D., and Eranian, S. *ia-64 linux kernel*. Upper Saddle River: Prentice-Hall, 2002.
17. Tweedie, L. et al. *The Attribute Explorer*. Video Proceedings of CHI'94 and Conference Companion Proceedings, CHI'94
18. OpenMP Architecture Review Board. *OpenMP: Simple, Portable, Scalable SMP Programming*. <http://www.openmp.org/>