

A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters

S. Browne*, J. Dongarra⁺, N. Garner*, K. London*, and P. Mucci*

Introduction

For years collecting performance data on applications programs has been an imprecise art. The user has had to rely on timers with poor resolution or granularity, imprecise empirical information on the number of operations performed in the program in question, vague information on the effects of the memory hierarchy, etc. Today hardware counters exist on every major processor platform. These counters can provide application developers valuable information about the performance of critical parts of the application and point to ways for improving performance. Performance tool developers can use these hardware counters as a basis for tools and interfaces that provide users with insight into diagnosing and fixing performance problems. The current problem facing users and tool developers is that access to these counters is often poorly documented, unstable or unavailable to the user level program. The focus of the PAPI project is to provide an easy to use, common set of interfaces that will gain access to these performance counters on all major processor platforms, thereby providing application developers the information they need to tune their software on different platforms. The goal is to make it easy for users to gain access to the counters to aid in performance analysis, modeling, and tuning.

The approach of the PAPI project has been to work with the high performance computing community, including users and vendors, to choose a common set of hardware events and to define a cross-platform library interface to the underlying counter hardware. The common set of events are those considered to be most relevant and useful in tuning application performance. As large a subset as possible has been mapped to the corresponding machine-specific events on the major HPC platforms and it is hoped that most of these events will be made available in the future on all major HPC platforms to improve the capability for tuning applications across multiple platforms. The intent is that the same metric would count similar, and possibly comparable, events on different platforms. Direct comparison between systems is *not* the intention of the PAPI predefined events. Rather, the intention is to standardize the names for the metrics, not the exact semantics, which necessarily depend on the processor under study. The PAPI library provides two interfaces to the underlying counter hardware: a high level interface for the acquisition of simple measurements, and a fully-programmable, thread-safe low level interface directed towards users with more sophisticated needs. The high level interface simply provides the capability to start, stop, and read the counters for a specified list of events. The target audience for the high level interface is application engineers and benchmarking teams looking to quickly and simply acquire some rudimentary performance measurements. The tool developer will likely find the high level interface too restrictive. The low level interface provides the more sophisticated functionality of

* Computer Science Dept., University of Tennessee, Knoxville

⁺ Computer Science Dept., University of Tennessee, Knoxville and Oak Ridge National Laboratory

user callbacks on counter overflow and hardware based SVR4 compatible profiling, regardless of whether or not the operating system supports it. These features provide the necessary basis for any source level performance analysis software. Thus, for any architecture with even the most rudimentary access to hardware performance counters, PAPI provides the foundation for truly portable, source level performance analysis tools based on real processor statistics.

Obtaining performance data for large-scale applications can be cumbersome and inefficient with traditional tools. The data collected may consist only of summary statistics and not provide much insight into the dynamic runtime behavior of the application. Most tools require that the data to be collected be specified prior to runtime and do not allow for runtime selection or adjustment of which events to measure or the level of detail or granularity at which measurements should be made. To address these issues, the tool infrastructure being developed to complement the PAPI library facilitates runtime tracing and analysis of hardware counter data, as well as runtime control over performance instrumentation. Utility routines layered on top of the PAPI library enable application developers to easily delineate interesting portions of the application, such as subroutines and loop nests, and to output hardware counter data at runtime via a socket connection to interactive end-user performance analysis tools. In addition, a dynamic instrumentation capability currently under development will provide mechanisms for calls to PAPI library and utility routines to be dynamically inserted into and removed from running applications, thus allowing runtime control over the selection of hardware events and of counting modes and granularity.

PAPI Architecture and Design

The PAPI architecture uses a layered approach, as shown in Figure 1. Internally, the PAPI implementation is split into portable and machine-dependent layers. The topmost portable layer consists of the high and low level PAPI interfaces. This layer is completely machine independent and requires little porting effort. It contains all of the API functions as well as numerous utility functions that perform state handling, memory management, data structure manipulation, and thread safety. In addition, this layer provides advanced functionality not always provided by the operating system, namely event profiling and overflow handling. The portable layer calls the substrate, the internal PAPI layer that handles the machine-dependent specifics of accessing the counters. The substrate uses whatever methods appropriate to facilitate counter access, whether that be register level operations (T3E), customized system calls (Linux/x86), or calls to another library (AIX 4.3). The substrate interface and functionality are well defined, leaving most of the code free from conditional compilation directives. For each architecture/operating system pair, only a new substrate layer needs to be written. Experience indicates that no more than a few weeks are required to generate a fully functional substrate for a new platform, if the operating system provides the necessary support for accessing the hardware counters. Substrates have been implemented for Pentium Pro/II/III on Linux, SGI/MIPS R10000/R12000 on IRIX 6.x [4], IBM Power 604/604e/630 on AIX 4.3 [1], Cray T3E/EV5 on Unicos/mk, and Sun Ultra on Solaris 8. Work is underway on the Compaq Alpha Ev4/5/6 on Tru64 Unix substrate.

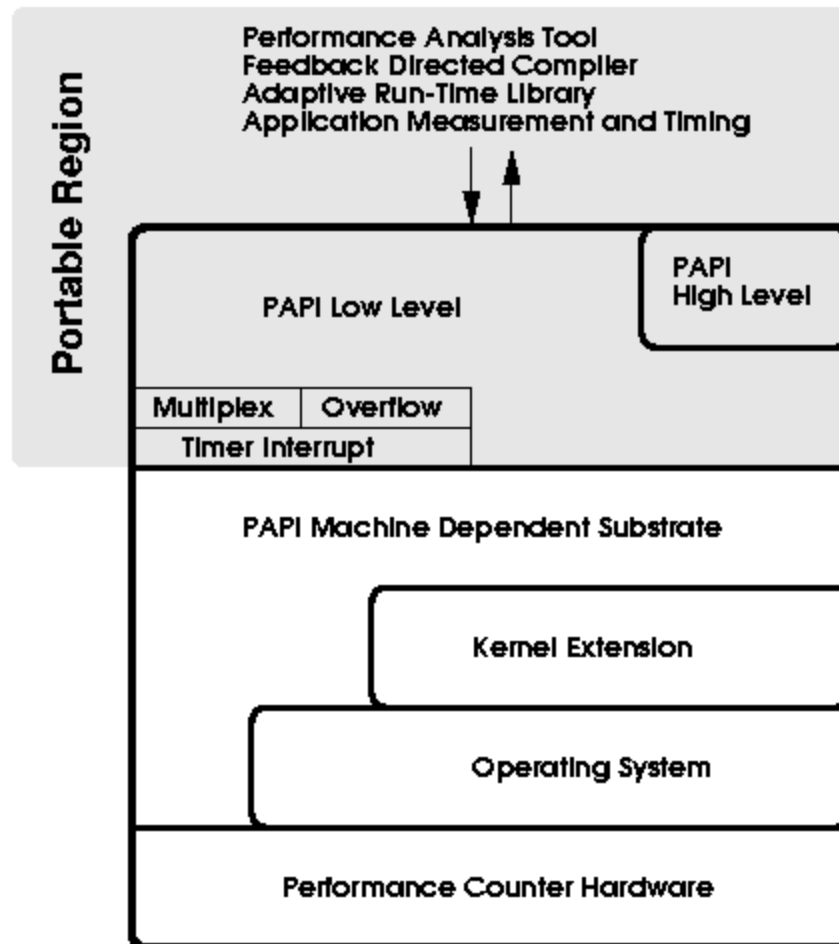


Figure 1. PAPI Architecture

PAPI provides an abstraction from particular hardware events called *EventSets*. An EventSet consists of events that the user wishes to count as a group. There are two reasons for this abstraction. The first reason is efficiency in accessing the counters through the operating system. Most operating systems allow the programmer to move the counter values in bulk without having to make a separate system call for each counter. By exposing this grouping to the user, the PAPI library can greatly reduce its overhead when accessing the counters. This efficiency is especially important when PAPI is used to measure small regions of code inside loops with large iteration counts. The second reason for *EventSets* is that users can evolve their own specialized counter groupings specific to their application areas. More often than not, relevant performance information is obtained by relating different metrics to one another. For example, the ratio of loads to level 1 cache misses is often the dominant performance indicator in dense numerical kernels. EventSets are managed by the user via integer handles, which simplifies inter-language calling interfaces. Multiple EventSets may be used simultaneously and may share counters.

Most modern microprocessors have a very limited number of events than can be counted simultaneously. This limitation severely restricts the amount of performance information that the user can gather during a single run. As a result, large applications with many hours of run time may require days or weeks of profiling in order to gather enough information on which to base a performance analysis. This limitation can be overcome by multiplexing the counter hardware. By subdividing the usage of the counter hardware over time, multiplexing presents the user with the view that many more hardware events are countable simultaneously. This unavoidably incurs a small amount of overhead and can adversely affect the accuracy of reported counter values. Nevertheless, multiplexing has proven useful in commercial kernel level performance counter interfaces such as SGI's IRIX 6.x. Hence, on platforms where the operating system or kernel level counter interface does not support multiplexing, PAPI plans to provide the capability to multiplex through the use of a high resolution interval timer.

One of the most significant features of PAPI for the tool writer is its ability to call user-defined handlers when a particular hardware event exceeds a specified threshold. For systems that do not support counter overflow at the operating system level, this is accomplished by setting up a high resolution interval timer and installing a timer interrupt handler. PAPI handles the signal by comparing the current counter value against the threshold. If the current value exceeds the threshold, then the user's handler is called from within the signal context with some additional arguments. These arguments allow the user to determine which event overflowed, how much it overflowed, and at what location in the source code. Statistical profiling is built upon the above method of installing and emulating arbitrary callbacks on overflow. Profiling works as follows: when an event exceeds a threshold, a signal is delivered with a number of arguments. Among those arguments is the interrupted thread's stack pointer and register set. The register set contains the program counter, the address at which the process was interrupted when the signal was delivered. Performance tools such as UNIX `prof` extract this address and hash the value into a histogram. At program completion, the histogram is analyzed and associated with symbolic information contained in the executable. What results is a line-by-line account of where counter overflow occurred in the program. Most existing profilers use process time as the overflow trigger. PAPI generalizes this functionality so that a histogram can be generated using any countable event as the basis for analysis.

The PAPI project has developed two tools that demonstrate graphical display of PAPI performance data in a manner useful to the application developer. The tool front-ends are written in Java and can be run on a separate machine from the program being monitored. All that is required for real-time monitoring and display of application performance is a socket connection between the machines. The first tool, called the *perfometer*, provides a runtime trace of a chosen PAPI metric, as shown in Figure 2 for floating operations per second (PAPI_FLOPS). This particular example illustrates how calls to PAPI routines at the beginnings and ends of procedures can provide information about the relative floating point performance of those procedures. The same display can be generated for any of the PAPI metrics. The second tool, called the *profometer*,

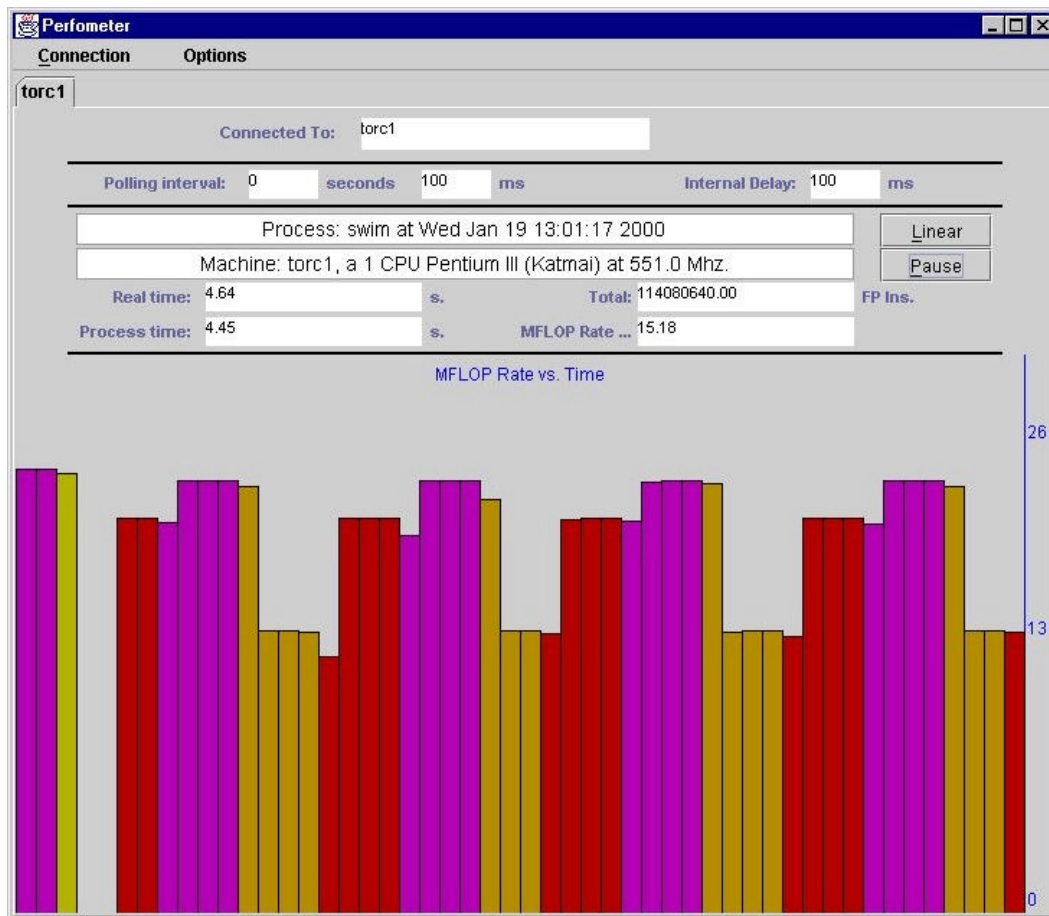


Figure 2. Perfometer displaying MFLOPs

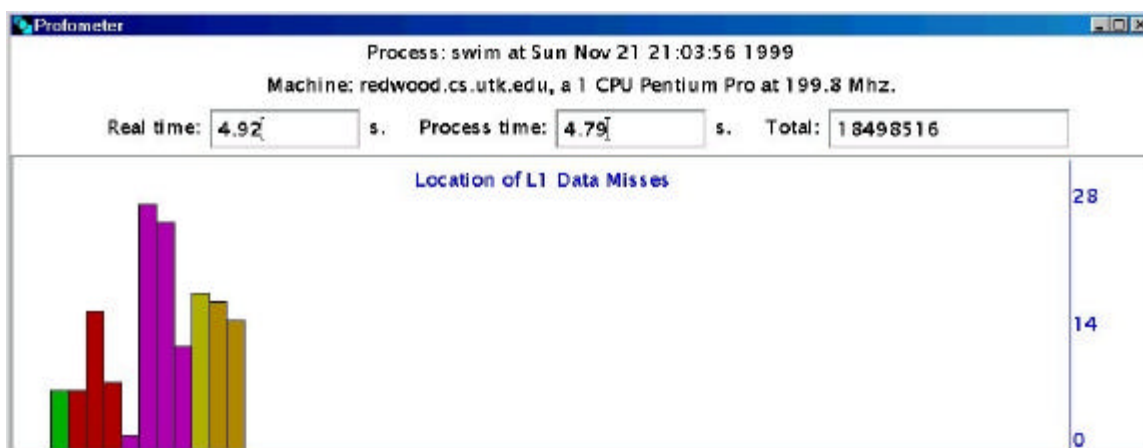


Figure 3. Profometer display of L1 data cache misses

provides a histogram that relates the occurrences of a chosen PAPI event to text addresses in the program, as shown in Figure 3 for L1 data cache misses. Again, the same display can be generated for any of the PAPI metrics. Future plans are to develop the capability of relating the frequency of events to source code locations, so as to allow the application developer the ability to quickly locate portions of the program that are the source of performance problems. In addition to these tools, PAPI support has been incorporated into a number of commercial and research performance analysis tools, including vprof from Sandia National Laboratory [8], SvPablo from University of Illinois and IBM [5,6], DEEP/MPI from Pacific Sierra Research, and TAU from University of Oregon.

Performance Tuning Using PAPI Metrics

The PAPI predefined events include accesses to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, and functional unit and pipeline status. Each of these areas is discussed in greater detail below, along with a discussion of how the metrics in each area relate to application performance tuning. Background on architectural issues discussed in this section can be found in [3,7].

Because modern microprocessors have multiple levels in the memory hierarchy, optimizations that improve memory utilization can have major effects on performance. Levels of the memory hierarchy range from small fast register memory to larger slower levels of cache memory to still larger and slower main memory. Different levels of cache memory may have separate instruction and data caches or may consist of a unified cache that buffers both instructions and data in the same cache. Most modern microprocessors have two levels of cache, although some now have three levels (e.g., three levels are optional with the Compaq EV5 and EV6 architectures). A load or store instruction generates a memory addressing process that first attempts to locate the addressed memory item in the L1 cache. If the item is present in the L1 cache, the result is an L1 *cache hit*. If the item is not present, the result is an L1 *cache miss* and an attempt is next made to locate the item in the L2 cache, with the result being an L2 cache hit or miss. The operating system uses main memory as a cache for a larger virtual address space for each process and translates between virtual addresses used by a program and the physical addresses required by the hardware. Memory is divided into blocks called pages. To keep the overhead of address translation low, the most recently used page addresses are cached in a translation lookaside buffer (TLB). When a program references a virtual address that is not cached in the TLB, a TLB miss occurs. If, in addition, the referenced page is not present in main memory, a page fault occurs. The latency of data access becomes greater with each level of the memory hierarchy, with a miss at each level multiplying the latency by an order of magnitude or more. The L1 and L2 cache hit rates indicate how cache friendly a program is and these rates can be derived from PAPI metrics. PAPI also provides TLB-related performance data.

For SMP environments, PAPI assumes a four-state MESI cache coherence protocol (possible states: modified, exclusive, shared, and invalid). In a cache-coherent SMP, the

system signals to a CPU when the CPU needs to maintain the coherence of cache data. An *intervention* is a signal stating that some other CPU wants to use data from a cache line that the CPU receiving the signal has a copy of. The other CPU requests the status of the cache line and requests a copy of the line if it is not the same as memory. An *invalidation* is a signal that another CPU has modified a cache line that this CPU has in its cache and that this CPU needs to discard its copy. A high number of cache line invalidations (PAPI_CA_INV) is an indicator of cache contention. The CPU that produces the high count is being slowed because it is using data being updated by a different CPU. The CPU doing the updating will be generating a high number of PAPI_CA_SHR events. Thus, a large number of PAPI_CA_SHR events also indicates that the program is being slowed by memory contention for shared cache lines. In an SMP, each processor has its own TLB, but page tables may be shared. If a processor changes a page table entry that another processor may be using, the first processor must force the second processor to flush its TLB. This notification that a page table entry has changed is called a translation lookaside buffer shutdown (PAPI_TLB_SD). A large number of PAPI_TLB_SD events indicates memory contention for the same page table entries.

PAPI metrics include counts of the various types of instructions completed, including integer, floating point, load and store, and synchronization instructions. A detailed discussion of how the counts for these various types of instructions can be used to analyze application performance is included in the full paper. PAPI includes events for measuring how heavily the different functional units are being used. Data for these events provide information about the ‘mix’ of operations in the code. Several of the PAPI metrics allow detection of when and why pipeline stalls are occurring. Because pipelining is for the most part beyond the control of the application programmer, the PAPI metrics relevant to pipelining are mainly intended to provide performance data relevant to compiler writers (e.g., for use in compiler feedback loops). However, the application programmer may be able to use pipeline performance data, together with compiler output files, to restructure application code so as to allow the compiler to do a better job of software pipelining. The application programmer may also find it useful to look at pipelining performance data when experimenting with different compiler options.

Letting an instruction move from the instruction decode stage of the pipeline into the execution stage is called *instruction issue*. An instruction is *completed* once all logically previous instructions have completed and only then is its result added to the visible state of the CPU. Because of speculative execution, a mispredicted branch can cause executed but not completed instructions to be discarded. Resource contention can cause instructions to be issued more than once before being completed. Normally, branch mispredictions and reissues are rare, and the number of issued instructions (PAPI_TOT_IIS) should correspond fairly closely to the number completed (PAPI_TOT_INS). A high number of mispredicted branches (PAPI_BR_MSP) indicates that something is wrong with the compiler options or that something is unusual about the algorithm. If the number of issued instructions greatly exceeds the number completed and the count of mispredicted branches remains low, then the load/store pipeline is likely

experiencing resource contention, causing load and store instructions to be issued repeatedly.

Ratios derived from a combination of hardware events can sometimes provide more useful information than raw metrics. Two ratios defined as PAPI metrics are floating point operations completed per second (PAPI_FLOPS) and total instructions completed per second (PAPI_IPS). Another useful ratio is completed operations per cycle (PAPI_TOT_INS/PAPI_TOT_CYC). A low value for this ratio indicates that the processor is stalling frequently. The typical value for this ratio will depend on the underlying processor architecture. The ratio of completed loads and stores per cycle (PAPI_LST_INS/PAPI_TOT_CYC) indicates the relative density of memory access in the program. The ratio of floating operations completed per cycle (PAPI_FP_INS/PAPI_TOT_CYC) indicates the relative density of floating point operations.

Related Work

The Performance Counter Library (PCL) is a common interface for accessing performance counters built into modern microprocessors in a portable way [2]. PCL supports query for functionality, start and stop of counters, and reading the current values of counters. Performance counting can be done in user mode, system mode, or user-or-system mode. PCL supports nested calls to PCL functions to allow hierarchical performance measurements. However, nested calls must use exactly the same list of events. PCL functions are callable from C, C++, Fortran, and Java. Similar to PAPI, PCL defines a common set of events across platforms for accesses to the memory hierarchy, cycle and instruction counts, and the status of functional units then translates these into native events on a given platform where possible. PAPI additionally defines events related to SMP cache coherence protocols and to cycles stalled waiting for memory access. Unlike PAPI, PCL does not support software multiplexing or user-defined overflow handling. The PCL API is very similar to the PAPI high-level API and consists of calls to start a list of counters and to read or stop the counter most recently started. PCL is available for Tru64 Unix on Alpha 21164 and Alpha 21264 processors, for CRAY Unicos/mk on Alpha 21164, for SGI IRI 6.x on R10000 and R12000 processors, for Solaris 2.x on UltraSPARC I/II, and for Linux 2.0.36 on Pentium/PPro/Pentium II/Pentium III. In the PCL Solaris and Linux implementations, the counters are not saved on context switches. PCL does not support native events.

Conclusions

PAPI aims to provide the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessor lines. The main motivation for this interface is the increasing divergence of application performance from near peak performance of most machines in the HPC marketplace. This performance gap is largely due to the disparity in memory and communication bandwidth at different levels of the memory hierarchy. To address

this problem, users need a compact set of robust and useful tools to quickly diagnose and analyze processor specific performance metrics. PAPI focuses on developing a reusable, portable, and functionality-oriented infrastructure for performance tool design and advanced program analysis.

The PAPI specification and software, as well as documentation and additional supporting information, are available from the PAPI web site at <http://icl.cs.utk.edu/projects/papi>.

Acknowledgments

The authors would like to thank John Levesque and Luiz DeRose at the IBM Advanced Computing Technology Center for assistance with the IBM reference implementation and integration of PAPI with SvPablo, Curtis Janssen at Sandia National Laboratory for integration with vprof and comments on PAPI profiling, Alex Poulos and Uros Prestor at SGI/Cray for assistance with the MIPS R10K/R12K implementation, and Monika ten Bruggencate at SGI/Cray for assistance with the Cray T3E implementation. The authors would also like to thank the Parallel Tools Consortium (Ptools) (<http://www.ptools.org/>) for sponsoring PAPI as a Ptools project.

References

- [1] Stephan Andersson, Ron Bell, John Hague, Holger Holthoff, Peter Mayes, Jun Nakano, Danny Shieh, and Jim Tuccillo. POWER3 Introduction and Tuning Guide. IBM, October 1998. <http://www.redbooks.ibm.com>
- [2] Rudolph Berrendorf and Heinz Ziegler. PCL – the Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors, Version 1.3. <http://www.fz-juelich.de/zam/PCL/>
- [3] Mark Brehob, Travis Doom, Richard Enbody, William H Moore, Sherry Q. Moore, Ron Sass, Charles Severance, “Beyond RISC – The Post-RISC Architecture”, Michigan State University Department of Computer Science, Technical Report CPS-96-11, March 1996.
- [4] David Cortesi, Origin 2000 and Onyx2 Performance Tuning and Optimization Guide. Document Number 007-3430-002, Silicon Graphics Inc., 1998. <http://techpubs.sgi.com/>
- [5] Luiz DeRose and Daniel A. Reed. “SvPablo: A Multi-Language Performance Analysis System”, Proceedings of the 1999 International Conference on Parallel Processing, September 1999, pp. 311-318.
- [6] Luiz DeRose, Ying Zhang, and Daniel A. Reed. “SvPablo: A Multi-Language Performance Analysis System”, 10th International Conference on Computer Performance Evaluation – Modeling Techniques and Tools – Performance Tools’98, pp. 352-355. Palma de Mallorca, Spain, September 1998. <http://vibes.cs.uiuc.edu/>

[7] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach, second edition. Morgan Kaufmann, 1996.

[8] Curtis L. Janssen. The Visual Profiler, Version 0.4, October 1999.
<http://aros.ca.sandia.gov/~cljanss/perf/vprof/doc/README.html>

[9] Luc Smolders. System and Kernel Thread Performance Monitor API Reference Guide, Version 0.5. IBM RS/6000 Division, May 1999.