# Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters

Jack Dongarra Kevin London Shirley Moore Philip Mucci Daniel Terpstra
Haihang You Min Zhou

University of Tennessee
Innovative Computing Laboratory
Knoxville, TN 37996-3450 USA

E-mail: papi@cs.utk.edu

## Abstract

*The PAPI project has defined and implemented a cross-platform interface to the hardware counters available on most modern microprocessors. The interface has gained widespread use and acceptance from hardware vendors, users, and tool developers. This paper reports on experiences with the community-based open-source effort to define the PAPI specification and implement it on a variety of platforms. Collaborations with tool developers who have incorporated support for PAPI are described. Issues related to interpretation and accuracy of hardware counter data and to the overheads of collecting this data are discussed. The paper concludes with implications for the design of the next version of PAPI.*

## 1. Introduction

The Performance API (PAPI) is a specification of a cross-platform interface to hardware performance counters on modern microprocessors [1]. These counters exist as a small set of registers that count *events*, which are occurrences of specific signals related to a processor's function. Monitoring these events has a variety of uses in application performance analysis and tuning, benchmarking, and debugging. The PAPI specification consists of both a standard set of events deemed most relevant for application performance tuning, as well as both high-level and low-level sets of routines for accessing the counters. The high level interface simply provides the ability to start, stop, and read the counters for a specified list of events, and is intended for the acquisition of simple but accurate measurements by application engineers. The fully programmable low-level interface provides additional features and options and is intended for third-party tool developers or application developers with more sophisticated needs.

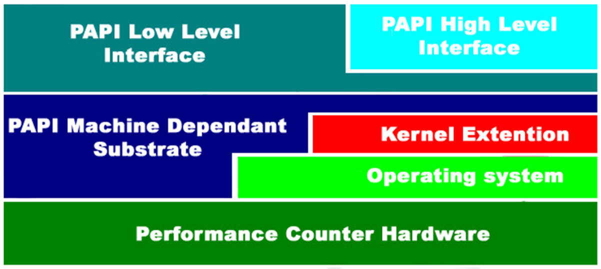

PAPI Low Level Interface
PAPI High Level Interface
PAPI Machine Dependant Substrate
Kernel Extention
Operating system
Performance Counter Hardware

**Figure 1. Layered architecture of the PAPI implementation**

In addition to the specification, the PAPI project has developed a reference implementation of the library. This implementation uses a layered approach, as illustrated in Figure 1. The machine-dependent part of the implementation, called the substrate, is all that needs to be rewritten to port PAPI to a new architecture. Platforms supported by the reference implementation include SGI IRIX, IBM AIX, HP/Compaq Tru64 UNIX, Linux/x86, LInux-IA64, Cray T3E, Sun Solaris, and Windows. For each platform, the reference implementation attempts to map as many of the PAPI standard events as possible to native events on that platform.

PAPI has been under development for four years and has become widely adopted by application and performance analysis tool developers. It is installed and in use for application performance tuning at a number of high performance computing sites throughout the world. The PAPI library is used by a number of end-user performance analysis tools to acquire hardware performance data. We believe the success of PAPI has resulted from a community effort to determine user requirements and from contributions not only from the development team but from vendors and outside developers participating in the open source project. The widespread use of PAPI has raised issues concerning interpretation of hardware counter data, as well as accuracy and efficiency of the counter interfaces. The experiences and lessons learned from the design, development, and use of PAPI are discussed in the following sections. The concluding section discusses plans for future work motivated by these experiences.

## 2. Design and Implementation Experiences

PAPI is a Parallel Tools Consortium (PTools) (http://www.ptools.org/) sponsored open-source project. PTools has provided the venue for a community effort to determine user requirements for PAPI and obtain user feedback during the design and implementation. In addition, the vendor community has participated by providing access to counter interfaces on their platforms The PTools web site maintains user and developer mailing lists for the project which have been actively used for reporting bugs, announcing updates, and discussing design and implementation issues. The development tree for the PAPI source code is maintained in a web-accessible repository. In addition to the PAPI library code, the repository includes documentation and a suite of test cases. Official releases of PAPI occur about twice a year, but pioneer users and tool developers can always obtain the latest version of the code from the development tree. Individuals outside the PAPI development team have contributed ports to different platforms, bug fixes, and test cases. We believe that the community-based open-source approach to design and development has allowed a low-budget academic project to have far-reaching impact in the world of high performance computing.

The separation of library functionality into the high-level and low-level interfaces was motivated by the desire to allow the user to choose between ease-of-use and increased functionality and features. In addition to the high-level start, stop, and read calls, the `PAPI_flops` call is an easy-to-use routine that provides timing data and the floating point operation count for the code bracketed by calls to the routine. The low-level routines target the more detailed information and full range of options needed by tool developers and some users. The low-level interface allows the user to manage events in *EventSets* and provides the functionality of user callbacks on counter overflow and SVR4-compatible statistical profiling, as well as access to all available native events and counting modes. For example, the `PAPI_profil` call implements SVR4-compatible code profiling based on any hardware counter metric. The code to be profiled need only be bracketed by calls to the `PAPI_profil` routine. This routine can be used by end-user tools such as VProf (http://aros.ca.sandia.gov/c̃ljanss/perf/vprof/) to collect profiling data which can then be correlated with application source code.

One issue that was discussed extensively on the PAPI mailing lists was the appropriate level at which to implement software multiplexing of hardware counters. Multiplexing allows more counters to be used simultaneously than are physically supported by the hardware. With multiplexing, the physical counters are time-sliced, and the counts are estimated from the measurements. There was concern that naive use of multiplexing could lead to erroneous results that would not be detected by the user. Erroneous results can occur when the runtime is insufficient to permit the estimated counter values to converge to their expected values. This issue was resolved by requiring multiplexing to be explicitly enabled in the low-level interface, rather than implementing it transparently in the high-level interface. Although high-level and low-level calls can be mixed, requiring the user to operate at the lower level to enable multiplexing presupposes a level of expertise that should include an understanding of the accuracy issues. Several tool developers make use of multiplexing to collect data for analysis by their tools but take care of ensuring that runtimes are sufficiently long to yield accurate results.

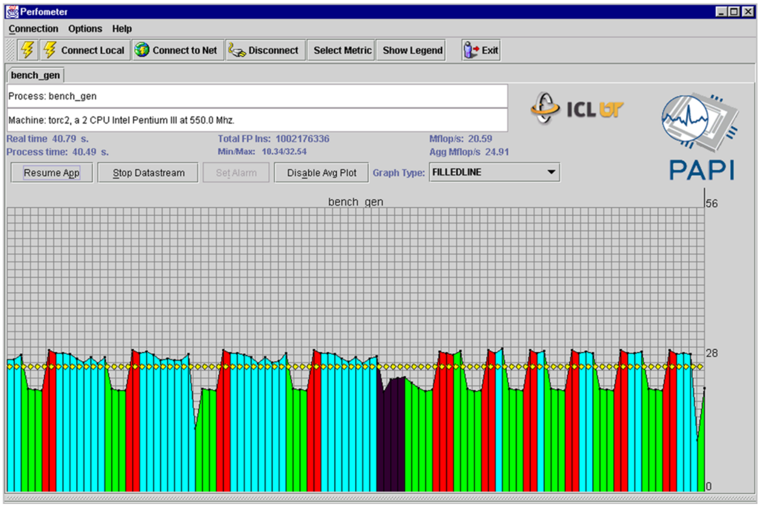
Implementation of the PAPI substrates for different platforms has attempted to use the most efficient native counter interface available on a given platform, whether that be register level operations (Cray T3E), customized system calls implemented in a kernel patch (Linux/x86), or calls to a vendor provided library (IBM pmtoolkit for AIX). In our experience, the requirement for a kernel modification has met resistance from system administrators, especially for large systems at multi-user high performance computing sites, due to security and reliability concerns. Thus, it is encouraging to see that the required kernel modifications are being incorporated into the standard release of some operating systems, for example the incorporation of pmtoolkit into AIX 5. In some cases, vendors have been very cooperative in extending their current interface to better support PAPI. For example, the aggregate counter interface originally available for Alpha Tru64 UNIX included only a handful of events and did not support per-process or per-thread counts. To make all the ProfileMe [3] events available through PAPI and to support per-process counts, Hewlett-Packard engineers extended the Alpha's DCPI interface, resulting in the Dynamic Access to DCPI Data (DADD) package now used by PAPI.

One of the most popular features of PAPI has proven to be the portable timing routines. Using the lowest overhead and most accurate timers available on a given platform to implement the PAPI wallclock and virtual timers enables users and tool developers to obtain accurate timings across different platforms using the same interface.

Although the primary focus of the PAPI project has been on development of the library and provision of a firm foundation for end-user tool developers to be able to access hardware counter data, the project has developed some simple end-user tools that can be used to quickly and easily obtain hardware performance data. However, the development of comprehensive tool support for the full range of parallel programming languages and models has been left to third-party tool development projects which are described in the next section. In this section, we describe two tools developed as part of the PAPI project.

The `dynaprof` tool uses dynamic instrumentation to allow the user to either load an executable or attach to a running executable and then dynamically insert instrumentation probes [10]. Dyanprof uses Dyninst API [2] on Linux/IA-32, SGI IRIX, and Sun Solaris platforms, and DPCL [1] on IBM AIX. The user can list the internal structure of the application in order to select instrumentation points. Dynaprof inserts instrumentation in the form of *probes*. Dynaprof provides a PAPI probe for collecting hardware counter data and a wallclock probe for measuring elapsed time, both on a per-thread basis. Users may optionally write their own probes. A probe may use whatever output format is appropriate, for example a real-time data feed to a visualization tool or a static data file dumped to disk at the end of the run. Future plans are to develop additional probes, for example for VProf and TAU, and to improve support for instrumentation and control of parallel message-passing programs.

Real-time performance monitoring is supported by the *perfometer* tool that is distributed with PAPI. By connecting the frontend graphical display, which is written in Java, to the backend process (or processes) running an application code that has been linked with the perfometer and PAPI libraries, the tool provides a runtime trace of a user-selected PAPI metric, as shown in Figure 2 for floating point operations per second (FLOPS). The user may change the performance event being measured by clicking on the Select Metric button. The intent of perfometer is to provide a fast coarse-grained easy way for a developer to find out where a bottleneck exists in a program. In addition to real-time analysis, the perfometer backend code can save a trace file for later off-line analysis. The Dynaprof tool described above includes a perfometer probe that can automatically insert calls to the perfometer setup and color selection routines so that a running application can be attached to and monitored in real-time without requiring any source code changes or recompilation or even restarting the application.



Process: bench_gen

Machine: torc2, a 2 CPU Intel Pentium III at 550.0 Mhz.

Real time 40.79 s.
Process time: 40.49 s.
Total FP Ins: 1002176336
Min/Max: 10.34/32.54
Mflop/s: 20.59
Agg Mflop/s 24.91

Graph Type: FILLEDLINE

**Figure 2. Real-time analysis using perfometer**

## 3. Interfacing to Third-party Tools

The widespread adoption of PAPI by third-party tool developers has demonstrated the value of implementing low-level access to architecture-specific performance monitoring hardware underneath a portable interface. Whereas tool developers previously had to re-implement such access for each platform, they can now program to a single interface, allowing them to focus their efforts on high-level tool design.

Performance analysis tools typically fall into two categories – profiling and tracing – although some provide both capabilities. Profiling characterizes the behavior of an application in terms of aggregate performance metrics. Profiles are typically represented as a list of various metrics (such as inclusive/exclusive wall-clock time) that are associated with program-level entities (such as routines, basic blocks, or statement in the program). Time is common metric, but any monotonically increasing resource function may be used, such as counts from hardware performance counters. Correlations between profiles based on different events, as well as event-based ratios, provide derived information that 0 to quickly identify and diagnose performance problems. While profiling is used to get aggregate summaries of metrics in a compact form, it cannot highlight the time varying aspects of the execution. To study the spatial and temporal aspects of performance data, event tracing, that is, the activity of capturing events or actions that take place during program execution, is more appropriate. Event usually results in a log of the events that characterize the

[1] http://oss.software.ibm.com/developerworks/opensource/dpcl/

execution.

PAPI has been incorporated into a number of profiling tools, including HPCView (http://www.cs.rice.edu/d̃system/hpcview/), SvPablo (http://www-pablo.cs.uiuc.edu/), TAU (http://www.cs.uoregon.edu/research/paracomp/tau/) and VProf (http:/aros.ca.sandia.gov/c̃ljanss/perf/vprof/). In addition, PAPI is being incorporated into future versions of the Vampir MPI analysis tool (http://www.pallas.com/e/products/vampir/index.htm). Collecting PAPI data for various events over intervals of time and displaying this data alongside the Vampir timeline view enables correlation of various event frequencies with message passing behavior.

SvPablo is a graphical source code browser and performance visualizer that has been developed as part of the University of Illinois' Pablo project [4]. SvPablo supports automatic instrumentation of HPF codes with Portland Group's HPF compiler and interactive instrumentation of C and Fortran programs. During execution of an instrumented code, the SvPablo library maintains statistics on the execution of each instrumented event on each processor and maps these statistics to constructs in the original source code. The statistics include counts of hardware events obtained using PAPI.

TAU provides a particularly good example of how PAPI has been incorporated into a comprehensive performance observation framework. TAU (Tuning and Analysis Utilities) is a portable profiling and tracing toolkit for parallel and threaded and/or message-passing programs written in Fortran, C, C++, or Java, or a combination of Fortran and C [11, 5]. Source code can be instrumented by manually inserting calls to the TAU instrumentation API, or by using the Program Database Toolkit (PDT) (http://www.cs.uoregon.edu/research/paracomp/pdtoolkit/) and/or the Opari OpenMP rewriting tool (http://www.fz-juelich.de/zam/kojak/opari/) to insert instrumentation automatically. The TAU project has used PDT to implement a source-to-source instrumentor that supports automatic instrumentation of C, C++, and Fortran 77/90 programs. The POMP interface for OpenMP provides a performance API for instrumentation of OpenMP codes that is portable across compilers and platforms [8]. The Opari tool rewrites OpenMP directives in 0 equivalent, but source-instrumented forms, inserting POMP performance calls where appropriate. The TAU MPI wrapper library uses the MPI profiling interface to generate profile and/or trace data for MPI 0. TAU MPI tracing produces individual node-context-thread event traces that can be merged and converted to ALOG, SDDF, Paraver, or Vampir trace formats. TAU can use DyninstAPI [2] to construct calls to the TAU measurement library and then insert these calls into the executable code. TAU can use PAPI to generate profiles based on hardware counter data. If TAU is configured without the multiple counters option, then the user selects the metric on which to base the profiling at runtime by setting an environment variable. If TAU is configured with the multiple counters option, then up to 25 metrics may be specified and a separate profile generated for each. These profiles for the same run can then be compared to see important correlations, such as for example the correlation of time with operation counts and cache or TLB misses. TAU includes both command-line and graphical tools for displaying and analyzing profile data.

## 4. Data Interpretation, Accuracy, and Efficiency Issues

The PAPI specification defines a set of standard events, and the PAPI implementation attempts to make as many of these standard events as possible available across platforms. However, even when the same event is available, it may have different semantics on different platforms, depending on the architecture and how the counters are implemented in the hardware. Thus, event counts should be interpreted in the context of the platform on which they were obtained. Understanding this context often requires the user to refer to detailed architecture manuals which may be difficult to obtain or may contain little or no information on the hardware counters. Sometimes additional information may be found in architecture-specific header files. For questions that cannot be resolved from the documentation, test programs may need to be written to determined exactly what events are being counted. These test programs can take the form of micro-benchmarks for which the expected counts are known. Even then, interactions with optimizing compilers can produce unexpected results. In the lower-level interface, PAPI does not attempt any normalization or calibration of counter data but simply reports the counts given by the hardware. In most cases where discrepancies have been reported between expected and measured results, the problem was found to be a misunderstanding of what events were actually being counted. For example, on the IBM POWER3 platform, a discrepancy in the number of floating point instructions was resolved when it was discovered that extra rounding instructions were being introduced to convert between double and single precision and were being included as floating point instructions. At the higher-level interface, however, PAPI sometimes attempts to normalize the data. For example, the PAPI_flops call attempts to return the expected number of floating point operations, which sometimes entails multiplying the measured counts by a factor of two to count floating-point multiply-add instructions as two floating point operations and/or subtracting counts for miscellaneous types of floating point instructions not normally including in floating point operations. In general, however, PAPI leaves the task of interpretation of counter data to the user or to higher-level tools.

As in any physical system, the act of measuring perturbs the phenomenon being measured. The counter interfaces necessarily introduce overhead in the form of extra instructions, including system calls, and the interfaces cause cache pollution that can change the cache and memory behavior of the monitored application. The cost of processing counter overflow interrupts can be a significant source of overhead in sampling-based profiling. A lack of hardware support for precisely identifying an event's address may result in incorrect attribution of events to instruction addresses on modern super-scalar, out-of-order processors, thereby making profiling data inaccurate. The PAPI project is concerned with all these possible sources of errors and is addressing them. PAPI is being redesigned to keep its runtime overhead and memory footprint as small as possible. Hardware support for interrupt handling and profiling is being used if possible.

Using PAPI on large-scale application codes has raised issues of scalability of PAPI instrumentation. PAPI initially focused on obtaining aggregate counts of hardware events, sometimes referred to as "exact counts", although research has shown that errors may exist [6]. However, the overhead of library calls to read the hardware counters can be excessive if the routines are called frequently – for example, on entry and exit of a small subroutine or basic block within a tight loop. Unacceptable overhead has caused some tool developers to reduce the number of calls through statistical sampling techniques [7]. On most platforms, the current PAPI code implements statistical profiling over aggregate counting by generating an interrupt on counter overflow of a threshold and sampling the program counter. On out-of-order processors, the program counter may yield an address that is several instructions or even basic blocks removed from the true address of the instruction that caused the overflow event. The PAPI project is investigating hardware support for sampling, so that tool developers can be relieved of this burden and maximum accuracy can be achieved with minimal overhead [9]. With hardware sampling, an in-flight instruction is selected at random and information about its state is recorded – for example, the type of instruction, its address, whether it has incurred a cache or TLB miss, various pipeline and/or memory latencies incurred. The sampling results provide a histogram of the profiling data which correlates event frequencies with program locations. In addition, aggregate event counts can be estimated from sampling data with lower overhead than direct counting. For example, the PAPI substrate for the HP/Compaq Alpha Tru64 UNIX platform is built on top of a programming interface to DCPI called DADD (Dynamic Access to DCPI Data). DCPI has very low overhead and identifies the exact address of an instruction, thus resulting in accurate text addresses for profiling data [3]. Test runs of the PAPI `calibrate` utility on this substrate have shown that event counts converge to the expected value, given a long enough run time to obtain sufficient samples, while incurring only one to two percent overhead, as compared to up to 30 percent on other substrates that use direct counting. A similar capability exists on the Itanium and Itanium2 platforms, where Event Address Registers (EARs) accurately identify the instruction and data addresses for some events. Future versions of PAPI will make use of such hardware assisted profiling and will provide an option for estimating aggregate counts from sampling data.

## 5. Conclusions and Future Work

The PAPI project has been successful in specifying and implementing a portable interface to hardware performance counters that has become widely accepted and used by application and tool developers. Experiences with version 1 and 2 of PAPI have highlighted some problems that remain to be solved, however, such as reducing the overheads and improving the flexibility of the interface. To address these problems, PAPI is undergoing a major re-design which will result in version 3. Some of the little used features of the previous versions, such as overlapping EventSets, are being eliminated in version 3 to reduce memory usage and runtime overhead and simplify the code. PAPI version 3 will also include more extensive support for hardware-based sampling and profiling, as well as an option for estimating counts from samples where this is supported by the hardware.

In addition to the timing and counter access routines already available, PAPI version 3 will also provide routines for obtaining memory utilization information which have been requested by users. The plan is to implement the following memory utilization extensions:

- memory available on a node
- total memory available/used
- (high-water-mark) memory used by process/thread
- disk swapping by process
- process/memory locality
- location of memory used by an object (e.g., array or structure)

PAPI manages events in user-defined sets called *EventSets*. EventSets are managed explicitly by the user in the low-level interface and by tool interfaces. One of the more difficult tasks in implementing a PAPI substrate for a particular platform has been to map an EventSet to the available physical counters in an optimal manner. Some native events are available on only some of the physical counters, and there are sometimes constraints on what events may be counted simultaneously, even if they can be mapped to different counters. Some platforms manage native events in groups and require counters to be allocated in a group instead of for individual events. In general, the counter allocation problem may be cast in terms of the bipartite graph matching problem, where the graph consists of two sets of vertices – one set representing the events to be mapped, and the other the representing the physical counters available on the machine – with an edge between an event vertex and a counter vertex if that event can be counted on that counter. A matching consists of a set of edges, no two of which are adjacent to the same vertex. The counter allocation problem attempts to find such a matching which includes an edge adjacent to each event vertex. Variations are to obtain a maximum cardinality mapping if not all the events can be mapped, or a maximum weight matching if some events have higher priority than others. We have designed an optimal matching algorithm which has been included in version 2.3 of PAPI. For PAPI version 3, we are attempting to separate the counter allocation into hardware-independent and hardware-dependent portions – the hardware-independent portion solving the graph matching problem and the hardware-dependent problem translating the counter scheme on a particular platform into the graph matching problem. This separation will hopefully make implementing optimal counter allocation on a new platform easier.

Development of the *dynaprof* and *perfometer* tools will continue with improved support for analysis of parallel message-passing programs. In addition, a *papirun* utility that will allow users to execute a program and easily collect basic timing and hardware counter data is under development. We expect to continue our collaborations with third-party tool developers in their use of PAPI to acquire accurate performance data with a minimum of overhead. In addition, we plan to collaborate with performance modeling projects such as that described in [12] in using PAPI to collect data for parameterizing predictive performance models. It is hoped that through additional collaborative efforts, PAPI will become one of a number of modular components for advanced tool design and program analysis on high performance computing systems.

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/papi/. See the Parallel Tools Consortium (PTools) web site at http://www.ptools.org/ to join the PAPI mailing lists and for general information about PTools.

## References

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[2] B. Buck and J. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[3] J. Dean, C. A. Waldspurger, and W. E. Weihl. Transparent, low-overhead profiling on modern processors. In *Workshop on Profile and Feedback-directed Compilation*, Paris, France, October 1998.

[4] L. DeRose, Y. Zhang, and D. Reed. Svpablo: A multi-language performance analysis system. In *Proc. 10th International Conference on Computer Performance Evaluation - Modeling Techniques and Tools- Performance Tools '98*, pages 352–355, 1998.

[5] A. Malony and S. Shende. *Performance Technology for Complex Parallel and Distributed Systems*. Kluwer, Norwell, MA, 2000.

[6] M. E. Maxwell, P. J. Teller, L. Salayandia, and S. Moore. Accuracy of performance monitoring hardware. In *Proc. LACSI Symposium*, Sante Fe, New Mexico, Oct. 2002.

[7] C. L. Mendes and D. A. Reed. Monitoring large systems via statistical sampling. In *Proc. LACSI Symposium*, Sante Fe, New Mexico, Oct. 2002.

[8] B. Mohr, A. D. Malony, S. Shende, and F. Wolfe. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, 2002.

[9] S. Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *International Conference on Computational Science (ICCS 2002)*, Amsterdam, Apr. 2002.

[10] P. Mucci. Dynaprof 0.8 user's guide. Technical report, Nov. 2002.

[11] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable profiling and tracing for parallel scientific applications using C++. In *Proc. ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998.

[12] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *SC 2002*, Baltimore, MD, Nov. 2002.