PAPI: A Portable Interface to Hardware Performance Counters

Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho
University of Tennessee
Knoxville, Tennessee 37996-1301

Abstract

The purpose of the PAPI project is to specify a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. These counters exist as a small set of registers that count events, which are occurrences of specific signals related to the processor's function. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. This correlation has a variety of uses in performance analysis including hand tuning, compiler optimization, debugging, benchmarking, monitoring and performance modeling. In addition, it is hoped that this information will prove useful in the development of new compilation technology as well as in steering architectural development towards alleviating commonly occurring bottlenecks in high performance computing.

Introduction

PAPI provides two interfaces to the underlying counter hardware: a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. The low level interface deals with hardware events in groups called EventSets. EventSets can reflect how the counters are most frequently used, such as taking simultaneous measurements of different hardware events and relating them to one another. For example, relating cycles to memory references or flops to level 1 cache misses can help detect poor locality and memory management. In addition, EventSets allow a highly efficient implementation which translates to more detailed and accurate measurements. EventSets are fully programmable and have features such as guaranteed thread safety, writing of counter values, multiplexing and notification upon crossing a threshold, as well as processor specific features. The high level interface simply provides the ability to start, stop and read the counters for a specified list of events.

PAPI provides portability across platforms. It uses the same routines with similar argument lists to control and access the counters for every architecture. PAPI includes a predefined set of events that we feel represents the lowest common denominator of every good counter implementation. This list can be found in appendix A. Our intent is that the same source code will count similar and possibly comparable events when run on different platforms. If the programmer chooses to use this set of standardized events, then the source code need not be changed and only a recompile is necessary. However, should the developer wish to access machine specific events, the low level API provides access to all available native events and counting modes.

In addition to raw counter access, PAPI provides the more sophisticated functionality of user callbacks on counter overflow and event multiplexing, 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.

Implementation

Internally, PAPI is split up into two levels each having very different functionality. The topmost layer consists of the 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 in addition to some utility functions. All data structure allocation, management and state handling is done by this layer exclusively. In addition, this layer contains code to provide some more advanced functionality not always provided by the operating system, namely event multiplexing and overflow handling. This portable code calls the *substrate*, the internal PAPI layer that handles the machine dependent

portions of accessing the counters. The substrate interface and functionality are well defined, leaving most of the code free from conditional compilation directives. Thus for each operating system/architecture pair, only a new substrate file needs to be written. Our experience indicates that no more than a day's effort is required to generate a fully functional substrate for a new platform.

A distinction should be drawn between the high level PAPI and low level PAPI interfaces. The high level interface has been included only to accommodate the novice user. Its functionality is quite limited and it remains to be seen whether or not it will ever be thread safe. The target audience for the high level interface is likely to be application engineers and benchmarking teams looking to quickly and simply acquire some rudimentary application metrics. The tool designer will likely find the high level interface restrictive.

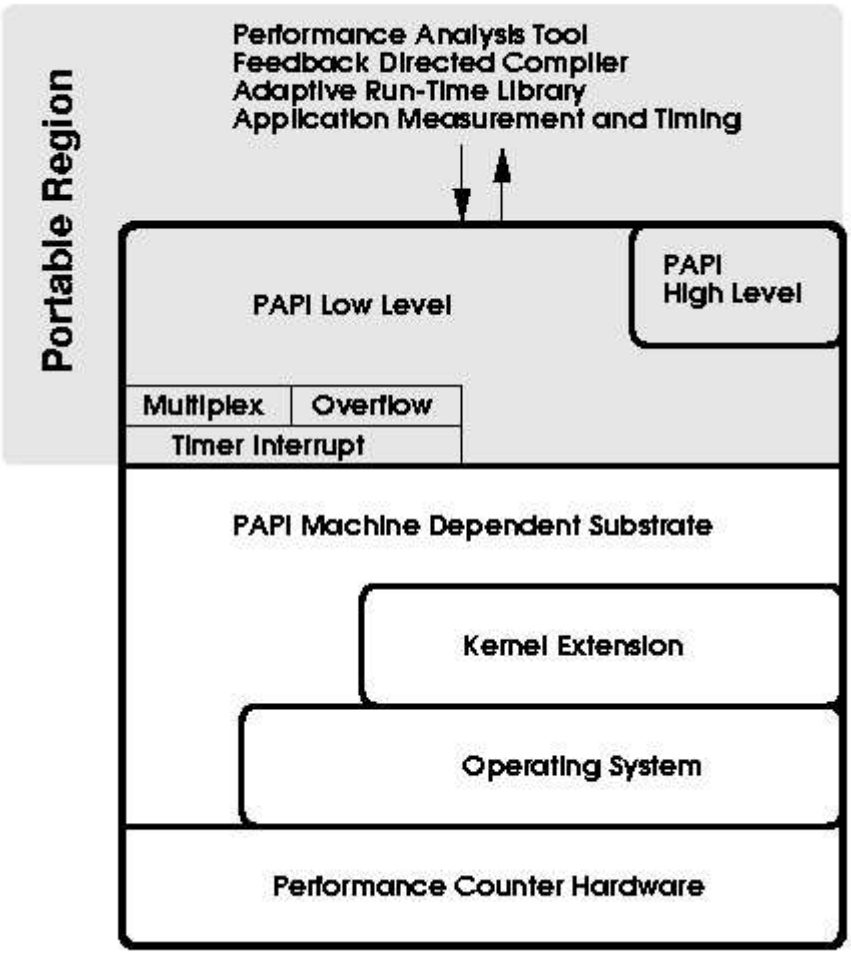

Figure 1: The PAPI Architecture

EventSets

PAPI provides an abstraction from particular hardware events called EventSets. An EventSet consists of countable events that the user wishes to count as part of a group. There are two reasons for this abstraction. The first being one of 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 system call for each particular counter. By exposing this natural grouping to the user, the PAPI library has the opportunity to be much more efficient when accessing the counters. This is especially true when PAPI is used to measure small regions of code inside loops with large iteration counts. The second reason for EventSets is one of application specific usage. In practice, most users have an understandably difficult time in relating a singular counter value to the performance of the region of code under study. More often than not, the relevant 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. It is foreseen that once the user becomes familiar with the semantics of PAPI he or she will evolve their own specialized counter groupings particular to their area of expertise.

EventSets are managed by the user through the use of integer handles which helps simplify inter-language calling conventions. There are no real programming restrictions on the use of EventSets. The user is free to allocate and use any number of them provided the substrate can provide the required resources. They may be used simultaneously and in fact may even share counter values. If the user tries to add more events to an EventSet than are simultaneously countable on the platform in use, PAPI will return an appropriate error code, unless the user has enabled multiplexing. The same holds true if the user starts additional EventSets that require counter hardware currently occupied by another running EventSet.

Overflow

One of the most significant features of PAPI as related to the tool writer is its ability to call user handlers when a particular hardware event exceeds a preset threshold. This is, in fact, the mechanism upon which most profilers are built. Profiling works as follows: when an event exceeds a threshold, a signal is delivered. On systems that do not support counter overflow at the operating system level, this signal is SIGPROF. Other systems, like IRIX 6.5, allow the user to specify that an arbitrary signal is delivered. Part of the arguments to the SIGPROF signal handler are the interrupted process's call stack and register set. The register set contains the address at which the process was interrupted when the signal was delivered. Performance tools like UNIX prof extract this address and hash the value into a histogram. At program completion, this histogram is analyzed and associated with symbolic information contained in the executable. What results is a line-by-line account of where counter overflows occurred in the program. GNU prof in conjunction with the -p option of GCC performs exactly this analysis using process virtual time as the overflow trigger. PAPI aims to generalize this functionality such that histograms can be generated using any countable event as the basis for analysis. The benefit of this being that specific symptoms can be diagnosed as opposed to the guessing game often played by developers as to why their code is taking so much time in a particular region.

Using the same mechanism as for user programmable overflow, PAPI also guards against register precision overflow of counter values. Each counter can potentially be incremented multiple times in a single clock cycle. This combined with increasing clock speeds and the small precision of some of the physical counters means that overflow is likely to occur on platforms where 64-bit counters are not supported in hardware or by the operating system. In those cases, the PAPI implements 64-bit counters in software using the very same mechanism that handles overflow dispatch.

Multiplexing

Most modern microprocessors have a very limited number of events that can be counted simultaneously. This severely restricts the amount of information the user can gather about a code in one pass. As a result, large applications with many hours of run time may require days or weeks of profiling in order for the developer to gather enough information from which to base an analysis. This limitation can be alleviated by multiplexing the counter hardware, or subdividing its usage over the time domain. This functionality presents the developer with the view that any set of hardware events are countable all the time. Naturally, multiplexing incurs a small amount of overhead and has the potential to adversely affect the accuracy of reported counter values. Nevertheless, similar features have proved quite successful in commercial kernel level implementations as found in SGI's IRIX 6.x and Compaq's Tru64 Unix. On platforms where the operating system or counter interface does not support multiplexing, PAPI has the unique capability to multiplex through the use of a high resolution interval timer.

As the task of application optimization often falls into the hands of the initiated computer scientist, bottleneck analysis is often treated as a black box problem. An initial approach is to do an analysis of large number of performance metrics collected over the entire run of the application. This data can then be related to average cost metrics to approximate the dominant bottleneck in terms of hardware resources required by the code. This facilities a much more detailed study of the application using overflow and statistical profiling as outlined above.

Thread Support

Much discussion has occurred on the interaction between threaded run-time environments and PAPI. As very large SMP's become ever more popular in the HPC community, fully thread aware performance tools are becoming an absolute necessity. This presents a significant challenge to the PAPI development team, due largely in part to the variety of thread packages. In particular, we must be careful not to embrace a particular threading package or API. There are two specific problems. As with any API, this interface must be re-entrant, because any number of threads can simultaneously call the same PAPI function. This means that any writeable global structures must be locked while in use. This has the potential of increasing overhead and introducing large sections of machine dependent code to the top layer. Fortunately, PAPI has only one large globally writeable structure, which keeps track of the EventSets currently defined by the user. In addition, it is only writeable by two API calls that are almost exclusively

used during the setup and shutdown of the PAPI library. The second problem, and of more serious concern to the PAPI development team, is the accuracy of event counts as returned by threads calling the PAPI. In order to support threaded operation, the operating system must save and restore the counter hardware upon context switches among different threads or processes. However, there are a significant number of threading API's that hide the concept of user and kernel level threads from the user. Pthreads and OpenMP serving as the most striking examples. As a result, unless the user explicitly takes action to bind their thread to a kernel thread (sometimes called a Light Weight Process or LWP), the counts returned by PAPI will not necessarily be accurate. Pthreads permits any 'Pthread' to run on any LWP unless previously bound. To address this situation, PAPI treats every platform as if it is running on top of kernel threads. Unbound, user level threads that call PAPI will function properly, but will most likely return unreliable or inaccurate event counts. Fortunately, in the batch environments of the HPC community, there is no significant advantage to user level threads and thus kernel level threads are the default. For situations where this is not the case, PAPI has plans to incorporate simple routines to facilitate binding by the less experienced user.

The API

The PAPI specification has been finalized following release of the initial draft and feedback from vendors, users, and tools researchers. What follows is a brief outline and description of the functions contained in PAPI. For more exhaustive documentation, the user is referred to the upcoming release or the PAPI home page at `http://icl.cs.utk.edu/projects/papi/`.

The Low Level API

The following functions are considered the low level API.

`int PAPI_set_granularity(int granularity)`

This function sets the default execution `granularity` over which entire EventSets are counted. By default, the `granularity` is set to the most restrictive supported by the substrate, ideally per thread. Other options include per process, per process group and over the entire system.

`int PAPI_set_domain(int domain)`

This function sets the default execution `domain` in which entire EventSets are counted. By default, the `domain` is set to user mode. Other options include kernel, system and possibly transient mode.

`int PAPI_perror(int code, char *destination, int length)`

This function copies `length` worth of the error description string corresponding to `code` into `destination` address. The resulting string is always null terminated.

`int PAPI_add_event(int *EventSet, int Event)`

This function creates a new `EventSet` or modifies an existing one and adds the hardware counter `Event` to the `EventSet`. This function returns an error on counter resource conflicts.

`int PAPI_add_events(int *EventSet, int *Events, int number)`

Same as above for a vector of hardware `Events`.

`int PAPI_add_pevent(int *EventSet, int code, void *inout)`

This call is similar to `PAPI_add_event()` except it is specifically designed to handle native programmable Events.

`int PAPI_rem_event(int EventSet, int Event)`

This function removes the hardware counter `Event` from `EventSet`.

`int PAPI_rem_events(int EventSet, int *Events, int number)`

Same as above for a vector of hardware `Events`.

`int PAPI_list_events(int EventSet, int *Events, int *number)`

This function decomposes `EventSet` into the hardware `Events` its contains.

`int PAPI_start(int EventSet)`

This function starts counting all the hardware Events contained in `EventSet`. All counts are implicitly initialized to zero. This function returns an error if another running EventSet introduces counter resource conflicts.

`int PAPI_stop(int EventSet, unsigned long long *values)`

This function terminates the counting of all hardware Events contained in `EventSet`. If `values` is non-NULL, the counters contained in `EventSet` are copied.

`int PAPI_read(int EventSet, unsigned long long *values)`

This function copies the running or stopped counters in `EventSet` into the values array. Internal counters are not be re-initialized to zero.

`int PAPI_accum(int EventSet, unsigned long long *values)`

This function accumulates the running or stopped counters in `EventSet` into the values array. Internal counters are initialized to zero.

`int PAPI_write(int EventSet, unsigned long long *values)`

This function assigns the internal counters of the Events contained in `EventSet` to that contained in `values`.

`int PAPI_reset(int EventSet)`

This function initializes the internal counters of the hardware Events contained in `EventSet` to zero.

`int PAPI_cleanup(int *EventSet)`

This function effectively removes `EventSet` from existence. The EventSet must be stopped in order for this call to succeed.

`int PAPI_state(int EventSet, int *status)`

This function returns the state, either running or stopped, of the entire EventSet.

`int PAPI_set_opt(int option, PAPI_option_t *ptr)`

This function enables options of the PerfAPI library, the PerfAPI substrate, specific EventSets or the operating system. This call controls the more advanced features of the PAPI library like multiplexing and overflow dispatch.

`int PAPI_get_opt(int option, PAPI_option_t *ptr)`

This function returns options of the PerfAPI library, the PerfAPI substrate, specific EventSets or the operating system.

`void PAPI_shutdown(void)`

This function shuts down the PAPI library and frees all associated resources.

The High Level API

The following interface provides the novice user with rudimentary access to the counting hardware. It should be noted that this API can be used in conjunction with the low level API, although such use is not recommended. If counter multiplexing has not been explicitly enabled by the user with `PAPI_set_opt()`, this API will only be able to

access those events countable simultaneously by the underlying hardware.

`int PAPI_num_events()`

This function returns the ideal length of the values array used in the following routines. It is guaranteed to be no longer than the maximum number of simultaneously countable events without multiplexing.

`int PAPI_start_counters(int *events, int array_len)`

Start counting the events named in the events array. This function implicitly stops and initializes any counters running as the result of a previous call to `PAPI_start_counters()`. It is the user's responsibility to choose events that can be counted simultaneously by reading the vendor's documentation.

`int PAPI_read_counters(unsigned long long *values, int array_len)`

Read the running counters into the `values` array. This call implicitly initializes the internal counters to zero and lets them continue to run upon return.

`int PAPI_stop_counters(unsigned long long *values, int array_len)`

Stop the running counters and copy the counts into the `values` array.

Current Status

Reference implementations of PAPI are currently underway for the SGI/MIPS R10000, R12000, the IBM Power 2SC, 604e and Power 3, the Cray T3E and Linux/x86/Alpha platforms. We anticipate a 0.99 release of PAPI to coincide with the 1999 Department of Defense HPC User's Group Meeting in Monterey, CA. Please check PAPI project page at http://icl.cs.utk.edu/projects/papi/ for current availability. Subsequent to the 0.99 release, will be full documentation and the a portable performance analysis tool based on hardware counters.

Conclusion

Hardware performance counters have been around for almost a decade. Yet most vendors and commercial tool developers have made little effort towards sharing the information with the HPC community as a whole. As a result, good single processor performance analysis tools have been few and far between. Excellent tools like SGI's SpeedShop have experienced remarkable popularity, but have elicited rather lukewarm responses from rest of the tool development community. The time is right to change this bias, and deliver users the tools they need on the platform of their choice. PAPI provides the necessary infrastructure for a highly accurate and portable performance analyzer that can be installed on every system throughout the MSRC's. Thus the user is free to adopt a consistent methodology for application tuning, based on a consistent set of performance analysis and optimization tools. Much like developers adopted the GNU environment because of its robustness and consistency, so could they adopt a portable framework under which the performance of complicated codes could be analyzed and understood. Users and developers with specific needs are encouraged to contact their MSRC's PET Programming Tools lead for information as to how to get involved in the PAPI project and make their requirements known.

Acknowledgments

This work has been partially supported by the DoD High Performance Computing Modernization Program, ARL and CEWES Major Shared Resource Centers, through Programming Environment and Training (PET).

Views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy or decision unless so designated by other official documentation.

PAPI has been supported in part by the work and information provided by the following individuals:
Erik Hendriks, perf kernel patch for performance counter support on the Linux/PPro/PII platform
Curtis Janssen, porting of Hendriks' perf to support Linux kernel 2.2.x
Don Heller, Pentium performance counter expertise
Alex Poulos, SGI, MIPS R10000 counter documentation

References

Cortesi, D., Topics in IRIX Programming, Document Number 007-2478-006, Silicon Graphics, Inc., 1998.
Hennessy, J., Patterson, D., *Computer Architecture A Quantitative Approach*, Second Ed., Morgan Kaufmann, San Francisco, CA, 1996.
Lewis, B., Berg, D., Multithreaded Programming with Pthreads, Sun Microsystems Press, 1998.
Silicon Graphics, Inc., *Speedshop User's Guide*, Document Number 007-3311-005, Silicon Graphics, Inc., 1998.

Figures

Figure 1: The PAPI Architecture

Tables

Table 1: Standardized Event Definitions

Symbol	Description
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_DCM	Level 2 data cache misses
PAPI_L2_ICM	Level 2 instruction cache misses
PAPI_L3_DCM	Level 3 data cache misses
PAPI_L3_ICM	Level 3 instruction cache misses
PAPI_CA_SHR	Request for access to shared cache line (SMP)
PAPI_CA_CLN	Request for access to clean cache line (SMP)
PAPI_CA_INV	Cache Line Invalidation (SMP)
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_TLB_SD	Translation lookaside buffer shootdowns (SMP)
PAPI_BRI_MSP	Branch instructions mispredicted
PAPI_BRI_TKN	Branch instructions taken
PAPI_BRI_NTK	Branch instructions not taken
PAPI_TOT_INS	Total instructions executed
PAPI_INT_INS	Integer instructions executed
PAPI_FP_INS	Floating point instructions executed
PAPI_LD_INS	Load instructions executed
PAPI_SR_INS	Store instructions executed
PAPI_CND_INS	Branch instructions executed
PAPI_VEC_INS	Vector/SIMD instructions executed
PAPI_FLOPS	Floating Point Instructions executed per second
PAPI_TOT_CYC	Total cycles
PAPI_MIPS	Millions of Instructions executed per second

Click Here To Go Back

Last Modifed September 28, 1999