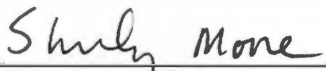


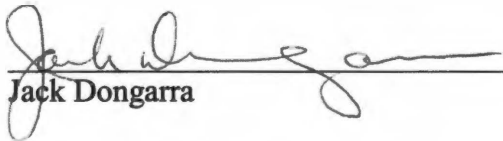
To the Graduate Council:

I am submitting herewith a thesis written by Kevin Scott London entitled "Enhanced Performance Monitoring – Extending PAPI to monitor multiple counter domains." I have examined the final paper copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.



Shirley Moore, Major Professor

We have read this thesis
and recommend its acceptance:



Jack Dongarra



Dali Wang

Accepted for the Council:



Vice Chancellor and
Dean of Graduate Studies

Thesis
2005
.L66

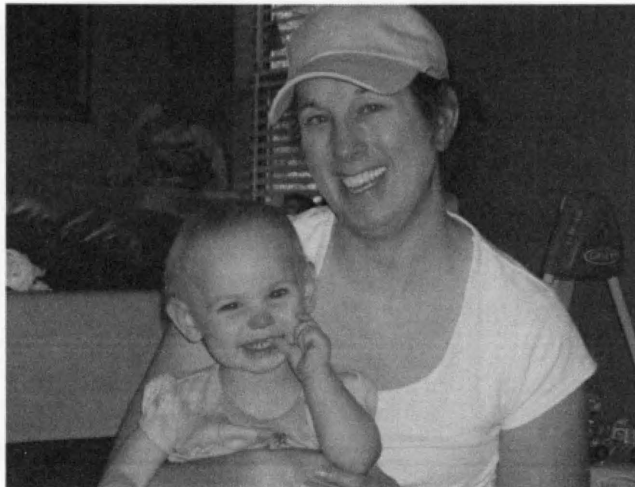
**ENHANCED PERFORMANCE MONITORING:
EXTENDING PAPI TO MONITOR MULTIPLE COUNTER DOMAINS**

**A Thesis
Presented for the
Masters of Science
Degree
The University of Tennessee, Knoxville**

**Kevin Scott London
August 2005**

Dedication

This thesis is dedicated to my wife, Christy London, my daughter Sydney London, my parents, Jack London and Edna London and my brother Lee London. Without their support and encouragement this thesis would not have been possible. I love you all.



Acknowledgments

I wish to thank all those who helped me complete my Master of Science degree in Computer Science. I would like to thank Dr. Shirley Moore for her guidance, suggestions on the thesis and serving as my chair on the committee. I would like to thank Dr. Dan Terpstra for letting me discuss ideas with him and for his help on suggestions on the thesis. I would like to thank Dr. Jack Dongarra for the many opportunities I have had while working for him and for serving on my committee. Without that experience this thesis would have been impossible. I would like to thank Dr. Graham Fagg for comments on the thesis and for being there when I needed a game for a temporary distraction from the thesis. I would like to thank Dr. Dali Wang for serving on my committee. I would like to thank Erika Fuentes who explained the concept of principal component analysis.

Finally, I would like to thank my family, their patience and encouragement made this work possible.

Abstract

High performance computing is playing an increasingly important role in the scientific community. As simulations start replacing physical experiments there will be a need for larger computing resources and better optimized applications. The Performance Application Programming Interface (PAPI) already addresses the concerns for optimizing a serial application by providing a portable interface to the performance counters found on modern processors. Developers can use these counters to understand exactly how their application is running on the processor and use this information to determine the best way to optimize the application. This information is only a portion of the complete picture. As applications become more complex the time to solution increases and the only way for an acceptable time to solution is to parallelize the application. Once an application is parallelized a developer needs more information to determine how to optimize the application. This information is available from the network interface cards, switches, distributed memory performance counters and other off processor locations. This additional information is very valuable to developers who need to optimize a parallel application. The computer systems are growing larger each year to accommodate these parallel applications [16]. Many of the larger systems are thousands of processors and there is one system that is planned to be one hundred thousand processors! Because these systems are getting so large another concern is power consumption and more importantly the thermal properties of the processors. Using thermal sensors is impractical because it is expensive, slow to respond to thermal changes and has design implications. There is ongoing research to use the processors performance counters to estimate the thermal properties of the processor. These algorithms are very complex and many times use more than 20 performance counter metrics for the thermal estimation. The algorithm is processor-specific which means a new algorithm has to be created for every new processor and in some cases even new revisions of an older processor. Because this process is time consuming it would be beneficial to monitor the thermal sensors and the hardware counters at the same time, but currently there is not a single package out there that allows these kinds of features.

This thesis will show that it is possible to construct a single interface that allows access to a variety of counters both on processor and off processor. The PAPI interface will be extended to support monitoring the multiple counter domains, but the interface will be preserved as much as possible. The preservation of the interface will ensure that the current PAPI user base will not have to make extensive changes to their applications, but will benefit by gaining the ability to monitor more than processor hardware counters. The extended PAPI interface will then be used to show how the ability to simultaneously monitor multiple counter domains is beneficial in verifying the accuracy of a thermal algorithm.

TABLE OF CONTENTS

Chapter	Page
1. INTRODUCTION.....	1
2. RELATED WORK.....	6
3. PAPI OVERVIEW.....	8
4. VECTORIZING PAPI.....	14
5. REMOVING CODE DEPENDENCIES.....	21
6. MULTIPLE SUBSTRATE SUPPORT.....	25
7. TEMPERATURE AND HARDWARE COUNTERS.....	29
8. FUTURE WORK.....	38
9. CONCLUSION.....	39
REFERENCES.....	40
APPENDICES.....	43
VITA.....	71

LIST OF FIGURES

Figure	Page
1. PAPI architecture.....	9
2. Extended PAPI architecture.....	13
3. Direct function call interface to the substrates.....	15
4. Function table interface to the substrates.....	16
5. Dendrogram for HPCC using PAPI preset events.....	30
6. Graph of normalized data from leaf node 1.....	32
7. Graph of reduced PAPI preset metrics.....	33
8. Principal components percent of variance explained.....	34
9. Principal component 1.....	34
10. Top 10 principal components percent of variance explained	35
11. Scatter plot principal component 1 vs. principal component 2	36

1. Introduction

High performance computing is playing an increasingly critical role in advanced scientific research as simulation and computation are becoming widely used to augment and/or replace physical experiments. However, the gap between peak and achieved performance for scientific applications running on high-end computing (HEC) systems has grown considerably in recent years. The complex architectures and deep memory hierarchies of HEC systems present difficult challenges for performance optimization of scientific applications. Tools are needed that collect and present relevant information on application performance in a scalable manner so as to enable developers to easily identify and determine the causes of performance bottlenecks. According to the Report of the High-End Computing Revitalization Task Force (HECRTF) [1], the single most important metric for high-end system performance is time to solution for the scientific applications of interest. Time to solution includes not only execution time, but also development time. Portable, easy-to-use, effective performance tools aim to reduce both development and execution time.

For a performance tool to be effective in reducing both development and execution time the tool needs to provide a developer with more than high resolution timing routines. Using timers to generate a profile of where the program is spending a majority of its time is important for a developer. Information from time-based profiles allows the developer to focus on specific routines when optimizing, allowing optimization to result in bigger performance gains. This is important because to achieve the largest performance gain the developer needs to make the common case fast. That is the developer should spend the majority of his or her focus on optimizing the code where the application spends the most amount of time. The benefit of this can be seen by calculating the speedup of an application by using Amdahl's Law.

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \quad [2] \text{ Where the enhanced}$$

portion is the section of code being optimized. If a developer optimized a section of code so it ran twice as fast and the code made up 10% of the execution time then the overall

speedup would be 1.05 or about a 5% increase in speed. However, if a section of code that accounts for 80% of the execution time can be optimized to run 20% faster then the overall speedup would be 1.15 or about a 15% increase in speed. Clearly developers need this information to best utilize their time and skills, but time-based information alone leaves many important questions unanswered – questions such as does a bottleneck exist in a particular section of code and what is causing a bottleneck. A bottleneck is where an application is not performing at the peak performance of the processor because the processor is stalled waiting on some resource such as the completion of a memory reference. Not all bottlenecks can be removed by software modifications, but many of the bottlenecks can with optimizations such as changing the underlying algorithm, reorganizing data structures and other optimization techniques. The number of bottlenecks that can be removed depends on if the technology, skill, and time exists to deal with them

To determine if a bottleneck exists that can be removed through software optimizations a developer needs more information than the execution time of a routine. Additional information such as the number of cache misses, total floating point instructions, the number of mis-predicted branches and other performance metrics is needed to help developers determine if a bottleneck exists. Fortunately, modern processors have special performance registers that can be configured to count performance metrics such as these. The specific performance metrics that can be counted vary with the particular processor being utilized. Most processors can count performance metrics specific to the cache such as cache misses at the different levels of cache. These metrics provide information to a developer on how well the application is taking advantage of the systems memory hierarchy. For example, if there are a lot of level 1 cache misses while doing matrix operations then the developer can optimize the application by blocking data accesses. Instead of operating on entire rows or columns of the matrix, a blocked algorithm operates on blocks of the matrix. This maximizes access to the data loaded in the cache before the data is replaced.[2] Additionally, most processors have events such as total floating point instructions (or operations) that can help a developer determine what percentage of a processor's peak performance a particular function is achieving. This

helps a developer in determining if a bottleneck exists in the function. Performance metrics such as those mentioned are essential to a developer in determining where and why bottlenecks exist in the code. Without this information developers would have to optimize their code by trial and error, which in most cases will not produce optimal code and the process is long and tedious. For this information to be useful in a multi-process environment the performance metrics need to be tied directly to an application, otherwise the usefulness of the counts is limited. Fortunately, most operating systems save the performance registers on context switches and those that do not can be modified to do so with kernel modifications. The only system where this does not hold true is on Microsoft Windows because there is no access to the kernel source code and performance registers are not saved and restored during context switches. Because the registers are saved the metrics can be directly related to the application. Unfortunately, developers could not easily access the performance counters because each platform had a different interface to access the counters and many times the interface was not well documented. Some platforms did not have an interface to access the counters at all and if they did, in some cases the performance registers were not saved on a context switch. This meant that the developer would need to write a kernel driver to save the performance registers on a context switch if the developer wanted to relate the counts back to his or her application. The Performance Application Programming Interface library (PAPI) was developed to address the problem of accessing the performance counters[4]. PAPI provides a portable, easy-to-use and well-documented API that provides developers access to setup, start, stop and read the performance registers on modern processors. PAPI allows developers an easy way to get performance information about their application. This solved many optimization problems and PAPI was quickly adopted, but PAPI is geared toward serial performance and developers need more information for application steering, system monitoring and to optimize parallel applications. The performance metrics that PAPI provided were only one portion of the complete picture.

Other components besides the processor, such as memory interface chips, network interface cards, and network switches, also have counters that count various events related to system reliability and performance. Unlike on-processor counters, the off-

processor counters usually measure events in a system-wide rather than a process or thread-specific context. However, when an application has exclusive use of a machine partition, it may be possible to interpret such counts in the context of the application. The current situation with off-processor counters is similar to the situation that existed with on-processor counters before PAPI. There are a number of different platform-specific interfaces, some of which are not documented. These counters become very valuable in tuning parallel applications and most scientific applications need to be parallelized to get the performance the application requires. Because communication bottlenecks are as important in a parallel application as memory bottlenecks are in a serial application, what is needed is a set of standard routines for accessing the off-processor counters on a number of platforms and networks. Some examples of off-processor counters that could be important in parallel applications are those for Myricom networks, Infiniband and Cray X1's E-chip and M-chip.

In addition, there are many sensors such as thermal sensors that can be beneficial to users as well. Thermal sensors are important because high-performance systems contain larger numbers of processors which require a lot of electricity. The more power that is used by the system the bigger the problem of thermal management becomes [5,6,7,8]. There are standardized interfaces for accessing the thermal hardware sensors, but these sensors cannot be used for runtime application throttling. The sensors are expensive to implement, placement of the sensors is vital to their accuracy, response times to temperature changes are slow and the sensors themselves add additional heat to the system [5,8]. Because of this there has been research by many different groups on using processor hardware counters to estimate the temperature of a processor [5,7,8]. The formulas for estimating the temperature are very complicated and normally require 22 or more performance metrics to calculate. In addition, the formulas are different for every processor due to power consumption and location of the various logical units on the chip. This means that a new formula has to be created and verified for every processor on the market [5,6,7,8]. Changing PAPI to concurrently monitor processor metrics and thermal sensors could help in identifying which processor metrics should be used in a formula and in the verification of a formula once it has been formulated.

The challenge is to develop a portable interface while still providing access to information that is necessarily somewhat platform-specific. In addition to routines to control and access the counters, there need to be query routines that will provide information about what counters and events are available, including both standardized and platform-specific events.

Because PAPI is a de facto standard in the performance industry, it is natural to extend PAPI such that the interface and internals can be used to read the off-processor counters. This approach will provide the established PAPI user base with a familiar interface and provide the additional information needed to optimize their parallel applications. PAPI consists of platform-independent code that is very portable and platform-dependent code that are called substrates. Each substrate contains the processor-specific code to access the hardware counters. Currently PAPI can only be compiled with a single substrate at a time. Because a developer may want to read off-processor and on-processor counters at the same time, PAPI needs to be modified to support compiling and using multiple substrates simultaneously. Additionally, this requires the API to allow developers to specify which off-processor hardware they want to access. For example, a computer may have two network interface cards and users need a way to specify which network interface card they want performance counts from. This thesis will examine the extensive code changes to PAPI that were made and what design decisions were considered to allow simultaneous access to multiple substrates, while preserving the familiar PAPI user interface as much as possible. Then this thesis will demonstrate how monitoring multiple counter domains is beneficial by using the extended PAPI to monitor thermal sensors and processor counters while running the HPCC challenge benchmarks. Then it is demonstrated which processor metrics are most important when determining an algorithm to estimate the thermal properties of a processor by using statistical analysis of the collected data.

2. Related Work

There are a number of other projects that allow access to performance counters found on processor and off processor as well. All of the projects were considered for extension to monitor multiple counter domains, but PAPI was chosen after consideration as PAPI was judged the most suitable.

The Performance Counting Library (PCL) is a performance counter interface developed by Rudolf Berrendorf, Heinz Ziegler, and Bernd Mohr from Germany. This project was developed about the same time PAPI and in many respects had the same goals of creating a standardized cross-platform interface to performance counters. The project has standardized events across platforms, a standardized interface and is well-documented and easy to use. PCL does not provide many features that PAPI supports that are beneficial to users. PCL does not allow users to count non-standardized events and so a user does not have access to all performance metrics that a processor supports. Also, PCL does not provide the overflow and profiling support that PAPI provides. Finally, PCL appears to be a dead project as the last update to the software was on January 2003 [9]. Because of the missing features and the project status, PAPI was chosen over PCL to implement the concurrent multiple substrates.

There are several packages available to count performance counters on specific processors such as: HPM Toolkit from IBM [10], perfmon from HP [11], perfctr by Mikael Petterson [12], and Speedshop from SGI [13]. These have tools to access the performance counters and in some cases, such as perfctr, PAPI uses these packages to get access to the counters. The problem with extending these packages is they are geared to a particular platform such as Linux, Irix or other operating system. Also, vendor packages such as the original Speedshop are not open source and so they cannot be modified to support multiple counter domains.

The Advanced Configuration and Power Interface (ACPI) is an open industry specification that was co-developed by Hewlett-Packard, Intel, Microsoft, Phoenix and Toshiba. This specification creates a portable interface to the power management and thermal features found on the hardware. The interface is similar to PAPI because the interface is portable, but the access to the hardware is necessarily hardware specific. By using the ACPI interface a program does not need to be rewritten on every new platform. However, ACPI will need to be ported to the new platform[14]. This interface is exactly what is needed to get access to the thermal sensors found on modern processors. It was decided not to extend ACPI to support multiple counter domains for several reasons. First the specification for ACPI is controlled by many members and is specific to power management and because of this the proposed extension would not be received well. More importantly PAPI is better suited to counting from different types of counters and can easily leverage the ACPI interface to gain access to the thermal sensors.

There are several projects that use function pointers to allow runtime selection of modules. The LAM project created the System Services Interface (SSI) which is a two-tiered modular framework for system interfaces. The SSI interface allows multiple instances of the same module types and then modules can be chosen at runtime. The main difference between the PAPI vectorized function table and the SSI interface is that SSI requires unique function names for the modules. This removes the chance of naming conflicts[15]. The PAPI vectorized function table does not have this requirement. Instead, the modules can either declare the functions static or use a unique name. The modules then return the function pointers to the library.

3. PAPI Overview

PAPI has been under development for many years. The main goal of PAPI is to provide a portable, well documented and easy to use interface to access the performance counters found on modern processors. The goals in the current release were to eliminate unused features, add functionality where needed and redesign the library for robustness, feature set flexibility, simplicity and speed and portability to new platforms. The main goals of PAPI have always remained the same throughout development, but many of the implementation goals were changed as dictated by past experience. For example, in PAPI the performance metrics that a user wants to monitor are organized into an EventSet and this is what is used to start, stop and read the counters. In PAPI 2.4.5 the library allowed multiple eventsets to be started simultaneously, but in practice we found that none of the users ever used this feature. So in PAPI 3.0.8.1, which is the current release of PAPI, the limitation was placed that only one EventSet could be started at any given time. This allowed the code to be streamlined and this resulted in a 15% to 90% reduction in overhead depending on the platform and function.[3] In PAPI 3.0.8.1 the portable code found in the substrates was moved up to the hardware-independent layer. This reduced the amount of code needed to port PAPI to a new platform. Only a single substrate can be compiled into PAPI and the portable code is tightly coupled to the substrate that is compiled in. These are limitations that will need to be addressed to support multiple substrates.

The current version of PAPI is a thread-safe library and has two APIs: the high level interface and the low level interface. The high-level interface is geared toward use by the scientific communities that want an easy way to get coarse grained information about their application. The high-level interface is built on top of the low-level interface, as shown in figure 1. The high-level interface provides simplified function calls such as

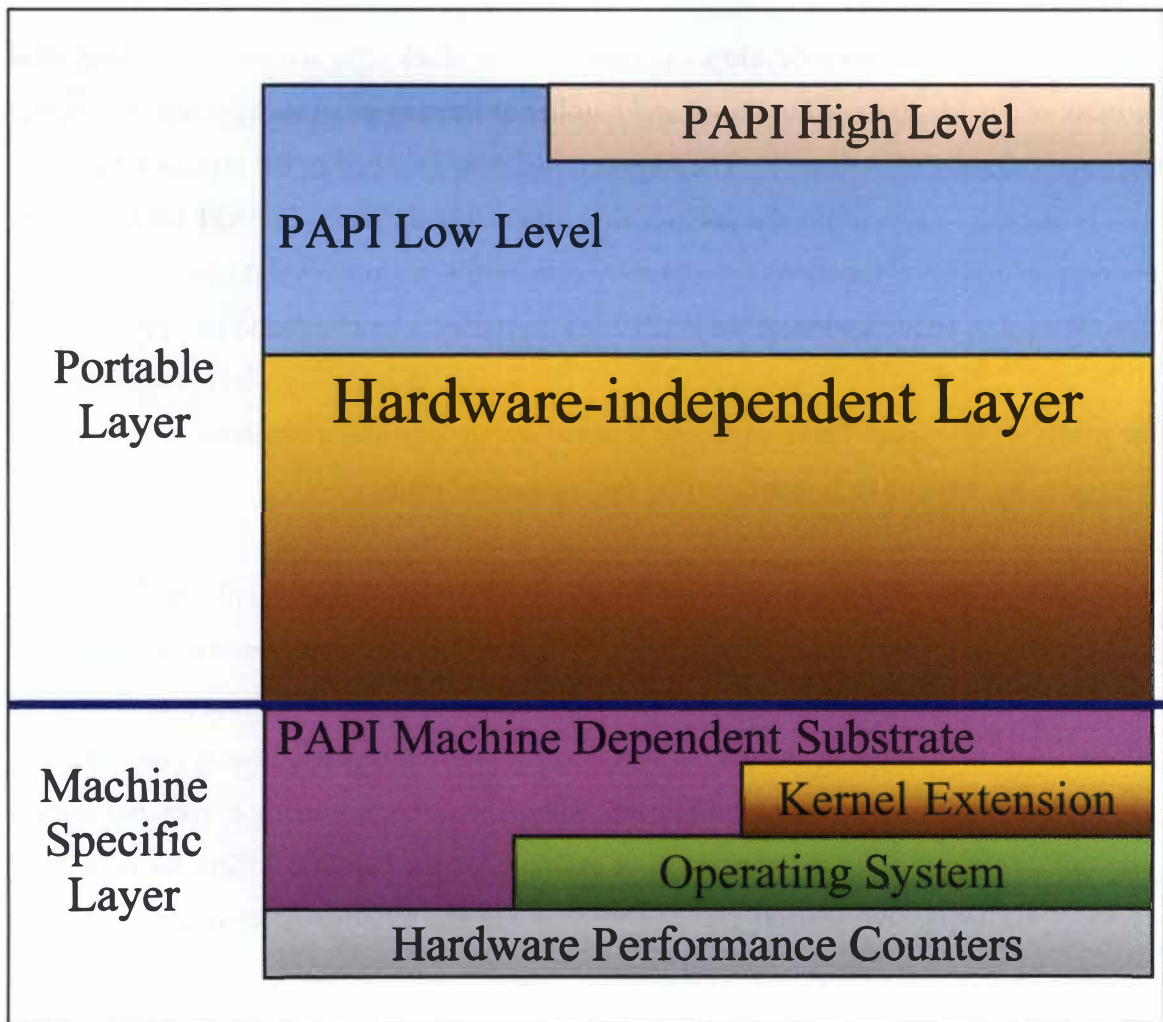


Figure 1. PAPI architecture

`PAPI_flops()` that allows a developer to measure wall clock time and processor time, total number of floating point instructions and number of floating point instructions per second between calls to `PAPI_flops()`. The high-level interface simplifies the interface for a user but doing so also limits the functionality exposed to the user. In PAPI 3.0.8.1, there are only 8 high-level functions for a user to learn, while the low-level provides 55 functions. The specification of the PAPI 3.0.8.1 interfaces can be found in Appendix A. If the extra functionality is not needed by the developer, then the high-level interface is the preferred interface. Three advanced features found only in the low-level interface are multiplexing, event-based overflow and event-based profiling.

Multiplexing allows users to count more performance metrics than the physical number of counters on the processor. Multiplexing is implemented by switching the performance metrics that are being monitored. The switching is done during time slices and this allows PAPI to extrapolate the counts to reflect the entire time period being monitored. This is important because many times users will want to monitor many events, but there may not be enough physical counters. For example, on the Pentium 3 there are only 2 counters available. While users could rerun the code and monitor different events, this may not be practical due to time constraints.

Overflow lets a user set a threshold on a metric and when that threshold is exceeded then the PAPI library calls a registered user function. Overflow is often used as another option to using a simple timer. For example, a developer may want to add fault tolerance to an application that performs many floating point operations. Instead of using timers to save the programs state the developer could monitor floating point instructions and setup a threshold. Then in the registered callback function the developer could save the state of the application.

Profiling provides a statistical histogram of a user's program. Profiling works a lot like overflow, but when the threshold is exceeded a program counter (PC) is recorded. The PC is used by PAPI to determine the location of the user's code that the threshold was exceeded. This is very useful for creating statistical profile of a program. The normal

profile is time based so you can answer the question where is a program spending a majority of its time. With profiling in PAPI you can answer questions like where in my program are a majority of Level 1 cache misses happening. This is a useful technique for determining the performance characteristics of a program.

The low-level interface also provides highly accurate timing routines, information about the processor and the executable being monitored, and information about any shared libraries that are being used. A key difference between the low-level interface and the high-level interface is that the high-level interface hides the fact that PAPI organizes everything into eventsets. In the high-level interface there is only one eventset, but in the low-level interface a user can create as many eventsets as needed, but only one eventset can be counted at one time. When counters are started, stopped or read an eventset is passed into the function (implicitly passed in when using the high-level interface) and this contains the information that the library needs to not only start the counters, but also which processor counters to read and what order they need to be returned. This is important because processors contain more than one counter and on most processors when a command to read a performance counter is issued you receive all performance counters. Each eventset is created with a call to `PAPI_create_eventset()` which creates the memory necessary to store the events being counted and the space needed for the internal PAPI book keeping.

Once the eventset is created then the user can add events to it by calling `PAPI_add_event()`. In PAPI an event can be either a PAPI preset or native code. A PAPI preset is an event that is standardized as much as possible across platforms, but not all PAPI presets are supported on all the platforms. A native event is processor specific and corresponds to a single performance metric that is supported by the processors performance counters. By definition all the native events are supported on the processor the native events belong to. There are only 103 presets in the current release of PAPI, but there may be thousands of native events depending on the processor. An event is specified by a code that is pre-defined in two different places -- the first is `papiStdEventDefs.h` which has all the standard PAPI preset events, and the second is in

the substrate dependent header file which has all the native events. In addition to adding events to an eventset by code, the code can be looked up by name and then used in `PAPI_add_event()`. This allows users to easily count the events they want without needing to know specific bit patterns needed on the various platforms. Once the eventset is set up, it can be used to start, read and stop the counters. This is an overview of the PAPI library that is exposed to the user, but there are many details that are hidden in the hardware-independent portion of PAPI and the substrate code.

These details will be discussed as they impact design decisions and overall understanding of how PAPI was modified to support multiple substrates as shown in figure 2. First the Vectorizing PAPI section will explain why the function calls to the substrate had to be changed, list possible solutions and why vectorization was chosen as the best solution. The next section will discuss how PAPI is dependent on the substrate header files, why the substrate header files have to be uncoupled from the library to support multiple substrates, and what changes in the PAPI library are necessary to support the uncoupling. The next section will discuss the changes to the library to support concurrently running multiple substrates, how the exposed API was changed and why the changes to the API were necessary. The next section will discuss future work on the multiple substrate support. Finally, there will be a detailed example on how multiple substrates are beneficial to the scientific community. The example will discuss how thermal properties have become a big issue in high performance computing, why simply reading a thermal sensor is not a viable option, the research using hardware counters to estimate thermal properties, how multiple substrates can be used to verify the estimates, and how this work could be extended to help determine which counters to use in the estimation.

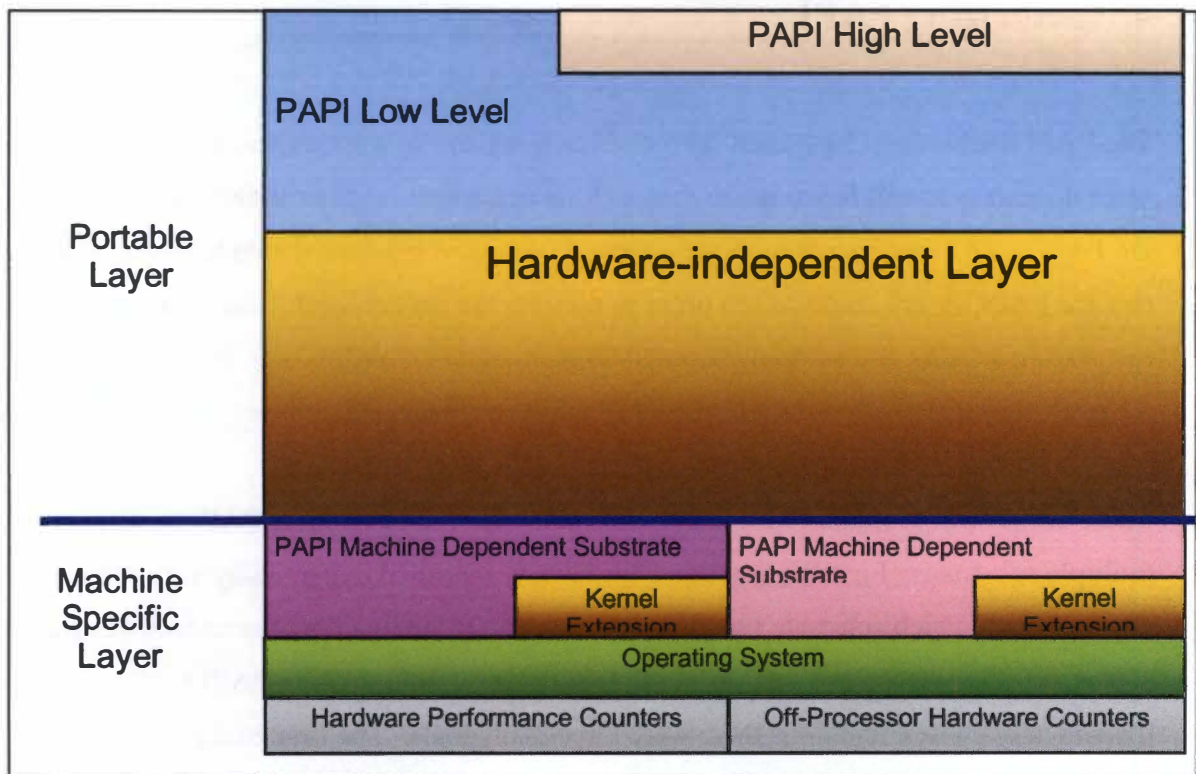


Figure 2. Extended PAPI architecture

4. Vectorizing PAPI

The PAPI hardware-independent layer does as much as the work as possible, but it is often necessary to call functions in the machine dependent layer as shown in figure 3. In the PAPI 3.0.8.1 implementation there were 33 different machine dependent functions that the PAPI library could call in order to perform the task at hand. These functions range from starting and stopping counters to intercepting overflow signals and passing the necessary information back to the PAPI hardware-independent layer.

There are several problems with this implementation when extending PAPI to support multiple substrates. First the function calls have to be called dynamically at runtime depending on which substrate functions are needed. If the functionality cannot be chosen at runtime then we can not support multiple substrates. In designing PAPI to allow dynamically chosen functions there were three main goals. The first goal is to preserve readability and maintainability of the code. The second goal is to simplify substrates by eliminating the need to create empty (stub) functions to satisfy the linker. The third goal is to introduce as little overhead as possible.

The dynamic function selection could be done with if statements to determine which substrate's function to call based on the events in an eventset. This approach is not a good solution because it does not satisfy either of the first two goals. It makes the code very unreadable as the more substrates you have the more if statements that are needed and this does not eliminate the need for stub functions. In addition, there will be some platforms where a substrate cannot be compiled and then the function calls to this substrate would need to have stub functions created for the library to compile. Another way to dynamically select the functions to call would be to use a vectorized function table. A vectorized function table is a lookup table that holds pointers to functions. These pointers are updated at runtime, and in your code instead of calling the function directly the code uses

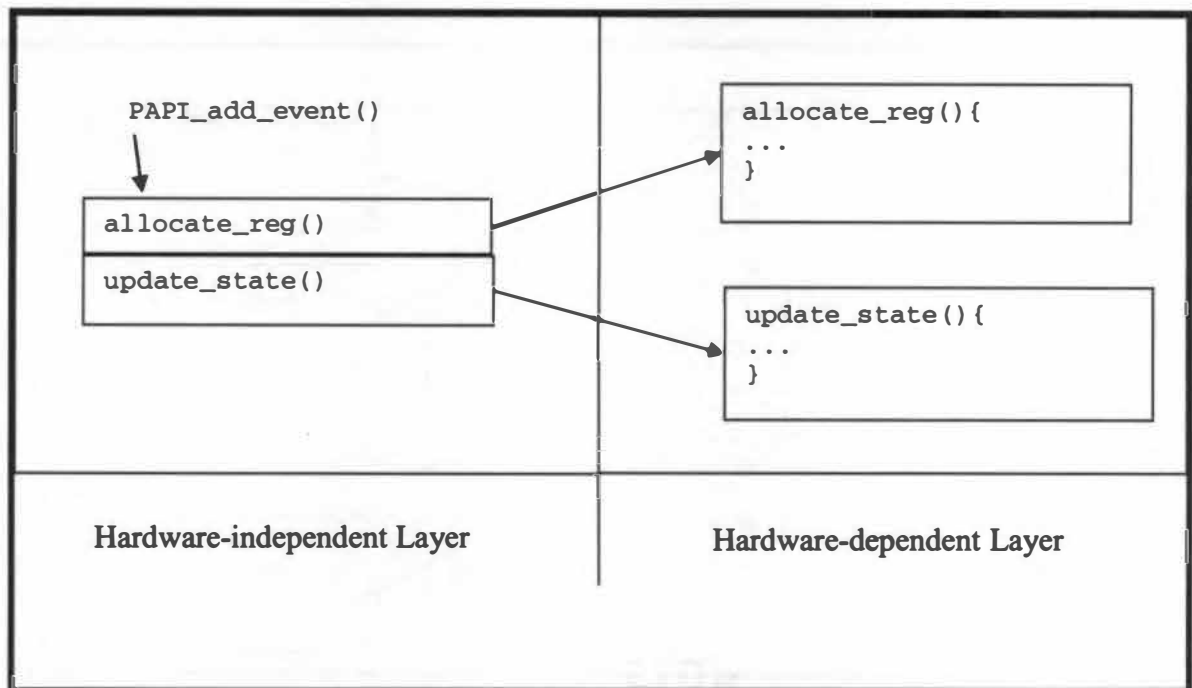


Figure 3. Direct function call interface to the substrates

an offset into the lookup table and then the pointer in the table is used to access the function as shown in figure 4. Because function tables indirectly access a function, there is extra overhead associated with each call. The extra overhead needs to be tested to make sure we satisfy the third main goal. This is a very important goal because additional overhead could mislead developers about performance problems, reduce the effective granularity of the counters, and possibly increase application runtime to a point that it is impractical to use PAPI. To determine the impact of indirect function calls, a test was devised to compare the overhead of a direct function call versus an indirect function call. The code for the test can be found in Appendix B. The test was run on a laptop computer with the following specifications: Pentium 4 M – 2.00 GHz with 512 MB of memory and running Fedora Core 2. X-windows and other non-essential processes were stopped to minimize the impact of other processes as much as possible. The test uses PAPI's high resolution timers which on this particular system have a resolution of 119 cycles. The test calls a loop that has either direct or indirect function calls. The loop is run 1,000,000,000 times to create a section of code long enough for the timing routines to be accurate and then the test was run 20 times to provide an average

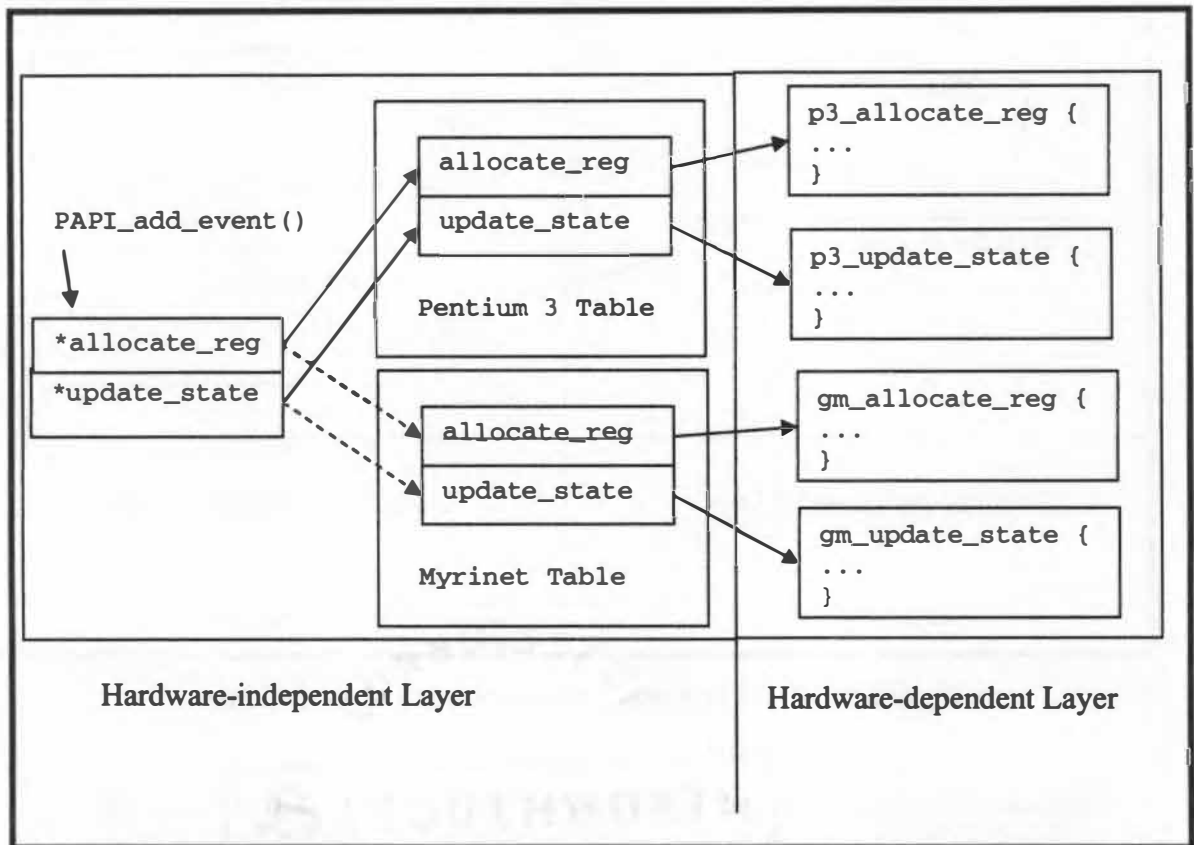


Figure 4. Function table interface to the substrates

result. This method of testing could possibly provide some caching benefits for the indirect function calls, but this is a simplified test to determine if function tables are a viable option. The average number of cycles for a direct function call was 12.2 cycles. The indirect function call added 4.2 cycles to this for a total of 16.4 cycles or a 34% increase in cycle count for the function call. This is a big performance hit for a single function call, but there are very few function calls being used and the overall cycle count for a PAPI function call is normally less than 1 percent of the total cycle count for the function. Switching to an indirect function call should increase the overhead by less than 3 percent. The reason the overhead could exceed 1 percent is because a PAPI function call may contain up to 4 indirect function calls before completing. The most important calls for PAPI to introduce as little overhead as possible are `PAPI_start()` and `PAPI_stop()` pairs and `PAPI_read()` calls. It was determined that a performance

penalty of less than 5% is an acceptable maximum to achieve the benefits afforded by the use of function pointers. According to the costs test provided with PAPI the cycle overhead for a `PAPI_read()` call on a Linux system with a Pentium 4 processor is 383 cycles and thus 4.2 extra cycles is only a 1% increase and would be within our maximum acceptable overhead. The Linux substrate is the lightest weight substrate in PAPI because the Linux substrate does not use system calls to read the counters and thus the increase should be even smaller on other substrates. For example, on a Solaris system with a UltraSparc III processor the cost of a `PAPI_read()` call is 1480 cycles and using a function pointer adds 12 cycles (calculated by the test code found in Appendix B) and so on this platform is only a 0.8% increase in overhead. As the overhead is minimal, I proceeded to turn the direct function calls into indirect function calls by using a vectorized function table. After the work is completed the PAPI library should be retested with the cost utility to determine the actual amount of extra overhead. This is necessary because creating the table could have hidden caching affects and other performance problems that are not apparent in the simple test used to see the viability of the function table.

To achieve the first goal a new header file was created called `papi_vector_redefine.h`. The purpose of this header is to use `#defines` to rename the functions in the hardware-independent layer into the proper function table entry lookups. For example, the `PAPI_read()` call is redefined with the following line:

```
#define _papi_hwd_read _papi_vector_table._vec_papi_hwd_read
```

This header file is then included in all the hardware-independent PAPI files to redefine the function calls. The next step is to create the vector function table structure and supporting functions. Initially, the vector table was created as a global table that supports a single substrate at a time. This was done so that the vector table implementation could be tested before integrating multiple substrates. To achieve the second goal, several different dummy functions were created to populate the function table. Several different dummy functions had to be created because the dummy function's return type must match the return type of the function. Dummy functions were created to return the error

code: `PAPI_ESBSTR`, which is the error code that specifies a function is not supported by a substrate. Additional dummy functions were created to return `PAPI_OK` because a substrate may not need a function that is necessary for the library to work properly, but if the function returned `PAPI_ESBSTR` then the PAPI library would believe there was an error condition that needed to be handled. For example, if `_papi_hwd_init()` returns an error code then the PAPI library assumes that the substrate was not initialized properly and will shutdown the library and return an error to the user application. However, the substrate may not need the `_papi_hwd_init()` function to properly initialize and thus returning an error code is erroneous behavior. These dummy functions simplify substrates because unsupported functionality does not have to be stubbed out. Once the dummy functions were completed, two functions were written for vector table support:

- `_papi_hwi_setup_vector_table()`, and
- `_papi_hwi_initialize_vector_table()`

and two functions were written for debugging support:

- `vector_print_routine()`, and
- `vector_print_table()`.

The `_papi_hwi_initialize_vector_table()` function takes one argument, a pointer to the function table that needs to be initialized. This routine populates the function table with appropriate dummy function pointers. The `_papi_hwi_setup_vector_table()` function takes two arguments, a pointer to the function table to be updated and a pointer to the substrate's opaque data structure that defines which functions a substrate supports. Each entry in the structure consists of two elements -- the first element is a pointer to the function and the second element is an integer identifier that uniquely identifies the function. The functions are identified by `#defines` in the PAPI headers that are included by every substrate. By creating the `_papi_hwi_setup_vector_table()` function, the substrate passes in a structure that the hardware-independent layer uses to fill out the function table. These functions make it easy for the substrate to setup the function table, guarantee a correct function table, and simplify the code in the substrates. Finally, the debugging functions were created to help a substrate developer determine if the substrate is setting up the function table correctly. The `vector_print_table()` routine takes a pointer to the function table and prints out which entries are mapped to a dummy

function and which entries are mapped to a user function. This allows the substrate developer a quick way to determine what is currently implemented in the substrate and if the function table entries are correct.

After implementing the vector code and porting all the substrates to use the new vector code, the library still needed to be tested to determine the amount of overhead that was added to the library. To test the amount of overhead the `papi_cost` utility was used. This program computes the minimum, maximum, mean and standard deviation of execution times for `PAPI_start()` / `PAPI_stop()` pairs and `PAPI_read()` calls. The `papi_cost` test was run on the systems on which the original test of function pointers was performed for comparison purposes. The tests were run 10 times and then the average of the 10 runs was used in the results located in table 1. The mean cycles was chosen to represent the cycle count in the table because on average this best represents the number of cycles in overhead that a users application will incur when using the PAPI function call being benchmarked.

Clearly even in the fastest implementation of the PAPI library the extra overhead is very minimal as the original tests suggested it would be. On Linux, the `PAPI_read` vectorized call had essentially the same overhead as the direct method. Meanwhile, the `PAPI_start/PAPI_stop` pair only incurs a 1.5% increase in overhead on the Linux platform. The other platforms show a similar pattern with slight increases or decreases of

Table 1. Comparison of overheads for direct and indirect PAPI function calls

OS / Function Type	Direct PAPI_read (cycles)	Direct PAPI_start / PAPI_stop (cycles)	Indirect PAPI_read (cycles)	Indirect PAPI_start / PAPI_stop (cycles)
Aix	2160.6	8300.4	2168.4	8332.6
Irix	4455.7	12574.6	4386.6	12762.7
Linux	409	18243.9	408.7	18502.7
Solaris	1924.2	6449.1	1975.4	6454.3

overhead when using the function pointers instead of direct function calls. The increase of overhead is much less than the goal of a maximum 5% increase in overhead. Thus using function pointers for accessing the substrate calls is a viable option to solve the need of dynamically calling different functions at runtime.

5. Removing Code Dependencies

In PAPI 3.0.8.1 the hardware-independent layer relies heavily on the data structures and global variables that are defined in the substrate header files. The hardware-independent layer does not explicitly access the data structures defined in the substrate header files, but it does statically allocate the space for the structures by including the structures in various hardware-independent data structures. The independent layer also copies, moves and compacts arrays of the structures as needed. This allows the hardware-independent layer to offload work that a substrate normally would have to do, such as thread safety and memory management, while maintaining the hardware-independent layer's ability to interface with multiple substrates. While the data structures are named the same inside the substrates, each substrate has different definitions of the structures and therefore different sizes as shown in table 2. There are several global variables that the PAPI library uses from the substrate header files such as `MAX_COUNTERS` and `MAX_COUNTER_TERMS` to determine various array sizes inside the library. In addition to the global variables all of the system include header files come from the substrate header files.

The current setup with PAPI 3.0.8.1 creates many obstacles that will need to be addressed to allow the PAPI library to compile multiple substrates into the library. First, the substrate data structures will cause naming conflicts between the different substrates as symbols would be multiply defined. Because the sizes of the substrate-specific data sizes are not known at compile time the library will need space allocated for these data structures. The system include files still need to be included in the library, but the substrate header file cannot be included because of the naming conflicts. Finally, the library will need some way to determine the sizes of substrate-dependent data structures and to find the value of various substrate global variables such as `MAX_COUNTER_TERMS`.

To address the naming conflict of the data structures, all references to the hardware-dependent structures need to be removed. This can easily be achieved by changing them

Table 2. Sizes of substrate data

OS	hwd_context_t	hwd_control_state_t	hwd_reg_alloc_t		hwd_register_t
AIX	128 bytes	128 bytes	44 bytes		44 bytes
Linux		4 bytes	500 bytes	52 bytes	36 bytes
Linux - Itanium 2		8 bytes	98,480 bytes	4 bytes	4 bytes
Solaris		4 bytes	64 bytes	4 bytes	8 bytes

to void pointers. There are four machine-dependent structures that need to be changed to void pointers: `hwd_context_t`, `hwd_control_state_t`, `hwd_reg_alloc_t` and `hwd_register_t`. Originally, the plan was to redefine the hardware-dependent data structures such as `hwd_context_t` to `void *` by including a header file in the independent layer that uses `#defines` to make the conversion. Because the header file that does the conversions would not be included in the substrates, the definitions of the structures would not be clobbered. This solution was problematic because of the header inter-dependencies in the PAPI library. It turned out that it was impossible given the header structure to redefine the structures to void pointers in just the hardware-independent layer. The best compromise that could be achieved was to manually change the declarations of the hardware-dependent data structures contained in the independent data structures to void pointers. Then the header file that contains the prototypes of the hardware-dependent functions was changed to redefine the data structures only when the header file was being included in the hardware-independent sections of code. This allowed the PAPI library and the substrates to use the same prototype header file. Next, the code needs to be changed to allocate the memory for these data structures dynamically and because the arrays are managed by the independent layer, the code needs to be carefully changed to guarantee that the arrays are handled correctly. Some of the things that had to be handled carefully are making sure that `memsets` and `memcpyys` to the hardware-independent data structures do not clobber the pointers to the hardware-dependent data structures that are contained within. Also the memory needs to be allocated by either the substrates or the hardware-independent layer. It is easier to

allocate the memory in the substrate as it already knows the sizes of the data structures, but the hardware-independent layer should do all the memory management. There are many reasons for the hardware-independent layer to do the memory management, but the most compelling reason is to support the memory management module. This memory management module provides extensive memory debugging routines and allows PAPI to determine if the library can free a pointer. This is important because in a few places PAPI does not know if the memory is allocated by the PAPI library or by the application and a library should never free memory that is allocated by an application. Without the memory management module, the PAPI library would have to assume that the memory is allocated by the application and this would create a possible memory leak. The PAPI team currently writes and maintains all the substrates and so the appropriate memory allocation routines can be guaranteed to be used by the substrates. However, it is hoped that in the future hardware vendors and other third parties will develop and maintain various substrates for PAPI and then it cannot be guaranteed that the correct memory routines were used. This means if PAPI is going to use the memory management module, then the hardware-independent layer will have to allocate all the memory used by the library. Because the independent layer does not know the size of the substrate data structures at compile time, an interface needs to be provided for the independent layer to determine the size at runtime. To keep the programming model familiar to the PAPI maintainers, the function `papi_sizeof()` was added that mimics the functionality of `sizeof()`, but reports the sizes of the various substrate-specific data structures. There were a few ways to deal with the global variables that affect the amount of memory that needed to be allocated. The first is to define the variables to be the same on all substrates. This works well for variables such as `MAX_COUNTER_TERMS` as most derived events of interest would not be composed of more than about 8 events with the exception of thermal estimates of a processor. This scheme doesn't work well for variables such as `MAX_COUNTERS` because this impacts the overhead and storage needed for all eventsets. Most platforms have between 2 and 8 counters with the exception being the Pentium 4 with 18 and the Cray X1 with 64. By forcing all platforms to allocate the space for 64 counters and then copy those data structures would increase the overhead and storage requirement to a point that this option is not feasible. The best way to store the

information is to create a hardware-independent data structure that can store the necessary information about the substrate such as MAX_COUNTERS. Finally, the system header files were moved to an operating system header file for both the substrate and the hardware-independent layers to include. Then the Makefiles needed updating to reflect the changes.

As a test for the changes, the Pentium 3 and Pentium 4 substrates were updated with the described changes and compiled together into the PAPI library. Because both use the same naming scheme for functions, they had to be declared as static functions to avoid naming conflicts. Once this was done, the initialization routines for the two substrates were renamed with a p3_ or p4_ prefix and a new initialization routine was written to determine if PAPI was being run on a Pentium 3 or a Pentium 4 and then the appropriate substrate is initialized. After these changes, the Makefile had to manually be updated to compile in both substrates. The resulting PAPI library successfully ran the test suite on both the Pentium 3 and Pentium 4 processors. There are a few things that would be generally useful in the future. Anytime a project may have modules (substrates), any module specific data should be passed around above the module as void pointers. Also the module should either allocate the needed memory or provide the layers above with a mechanism to determine the sizes needed. If these ideas are thought of during the design phase, implementing the ability to load multiple modules will be much easier.

6. Multiple Substrate Support

Changing PAPI to allow simultaneous substrates to be loaded requires many changes to the PAPI internals. The main challenge is to support multiple substrates while preserving the API exposed to applications that link with PAPI. In some cases the API has to be slightly changed, but most of the APIs remain unchanged.

In PAPI 3.0.8.1 there are several important data structures that are tied to the substrate. These data structures are tied to either the thread context or they are tied to the eventset data structure. The thread context is inherited by all eventsets that are active in a thread. The substrate-specific data contained in the thread context has information that the substrate uses to obtain thread specific information. This includes reading the executable information, shared library information and in some substrates access to the hardware counters. The eventset data structures are unique for every eventset that is created. The eventset data structures include substrate-specific data that the substrate uses to set the counters to the correct state and the resources necessary to choose the performance metrics on the counters.

The data structures are important, but the flow of control through PAPI is also important in supporting multiple substrates. There are several PAPI functions that a program would need to call to use PAPI to read the hardware counters. First the application would initialize the library by calling `PAPI_library_init()`. When `PAPI_library_init()` is called, initialization tasks are carried out which include reading the hardware and executable information, setting up the counter data structures, and allocating any memory that may be necessary. The substrate initializes a global structure that contains all the hardware and executable information as well as what functionality the substrate supports. Next an eventset would need to be created by calling `PAPI_create_eventset()`. The `PAPI_create_eventset()` function allocates memory for the hardware-independent and substrate-specific data structures and then links the eventset to the thread context. Once the eventset has been created, events are added with calls to `PAPI_add_event()` by

specifying an integer that uniquely identifies the event. If a native event is being specified then the application needs to call `PAPI_name_to_code()` to find the unique identifier to pass into `PAPI_add_event()`. This checks the high order bits of the identifier to determine if the event is a native or a preset event. If it is a preset then the native events are identified and the substrate is called to check for conflicts and setup the data structures necessary to count the events specified. Then the program can use `PAPI_start()`, `PAPI_read()`, and `PAPI_stop()` to use the counters to obtain performance information. All three of these use the eventset that was setup in the `PAPI_create_eventset()` call to pass the necessary information to the substrate to start, read, and stop the counters. After the program has finished using the PAPI library, a call to `PAPI_shutdown()` will de-allocate the memory and call the substrate to shutdown the interface to the performance counters.

There are several obstacles that have to be overcome with the flow of logic before PAPI will be able to use simultaneous multiple substrates. First, when the library is initialized, the substrate is initialized by calling the substrate. There needs to be logic in place to determine what substrates are loaded and what functions need to be called to initialize the substrate. In addition, the global data structure will need to be changed to support multiple substrates and there needs to be a way for the library to determine the size of substrate dependent data structures. The biggest obstacle is deciding on how to deal with the events themselves. When events are added, the substrate is called to see if an event exists and so a mechanism has to be in place to decide which substrate an event belongs to. Another decision to be made is if events from multiple substrates should be supported in one eventset or if an eventset should be tied to a substrate. Finally, shutdown needs to have the information necessary to shutdown all substrates to make sure all the memory and other resources are cleaned up.

To determine which substrates are compiled into the PAPI library there are several different solutions to address the problem. Users could specify which substrates to compile with defines in the Makefile, and the defines can be used to determine which

substrates are loaded up during runtime. The problem with this solution is that users may not know what hardware is installed on the system. To make things easier on the user, the configure script was changed to detect what hardware is available and this information is saved in a config header file that can be used to determine which substrates are compiled into the library. There is still a problem with determining what function call to use to initialize the substrate as all of the functions have to be uniquely named to prevent naming collisions. One solution would be to define a standardized naming scheme for the substrates so that the initialization routine would be the name of the substrate followed by `_init`. Instead of forcing substrates to adhere to a naming scheme a new function was created called: `_papi_hwi_find_init(char *name)`. The function takes the name of the substrate and returns a function pointer to the initialization routine. An array of strings created by the configure script contains a list of all substrates that are compiled in. Then this array can be iterated on and the functions called to initialize all of the substrates. The number of substrates loaded is then stored in a global variable so when `PAPI_shutdown()` is called, all the substrates will have their shutdown function called. The function `papi_sizeof()` routine had to be changed so that it takes the substrate index as an additional argument. As long as the library knows which substrate memory is being allocated for then the `papi_sizeof()` routine can return the correct size. Unfortunately in some cases such as creating a thread there is not an eventset or anything else available to determine which substrate that the memory needs to be allocated for. Because every substrate may need a different thread context the decision was made to create an array of pointers in the thread data structure. This way each substrate can be allocated a context of the appropriate size and use the space as needed. The original solution was to allow events from different substrates in a single eventset because this is convenient for the users. The problem with this solution is that the overhead with the `PAPI_start()`, `PAPI_read()`, and `PAPI_stop()` calls would be dependent on which events the user adds to the eventset. Because the additional overhead involved could be 100% or more depending on which events are added, this solution was deemed inappropriate. Most users would assume that the overhead of a call does not change depending on the event that is added. Because of this a single eventset can only contain events from a single substrate. However, each substrate can have an

eventset running at any given time. This will keep the overhead consistent for the user depending on which substrate the eventset is tied to, but still allow users to monitor events from different substrates at the same time. Next the exposed API needed to be changed to give users the ability to tie substrates to an eventset. One solution was to set which substrate an eventset supports when the first event is added to the eventset. However, I decided to use the `PAPI_create_eventset()` call to set which substrate an eventset supports. Setting the substrate seems natural in this function call and this way a user knows explicitly which type of events can be added to an eventset. I wanted to keep `PAPI_create_eventset()` the same for backwards compatibility and originally I tried using varargs such that the eventset would be tied to the processor substrate unless a substrate index is explicitly passed in. The problem was that there is no way to determine how many arguments (or even if any arguments) are being passed into the function. Because of this I created a new function called `PAPI_create_ext_eventset()`. This function needs to be called for creating an eventset for any substrate other than the substrate for the processor. This provides backwards compatibility and does not affect the external API much.

After making these changes, PAPI supports running simultaneous multiple substrates. There were several lessons learned that are applicable to other projects in general. The first is when specifying the exposed API it would be beneficial to add an integer as the last argument. Then if additional arguments are needed to a function the function could be changed to use varargs with the last integer specifying what extra arguments are included. This would allow extra functionality to be added to a library without the need for new functions or breaking backward compatibility. Also, it would be beneficial to determine if modularity of the backend is something that will be desired in the project in the future. If this is something that will be needed, then making sure the memory is allocated based on a function provided by a module would make supporting multiple modules much easier.

7. Temperature and Hardware Counters

Now that PAPI has been extended to support multiple counter domains, PAPI will be used to try and determine which performance metrics need to be considered to estimate the thermal properties of a processor. For this example, I will be instrumenting the HPC Challenge benchmarks. This will make for an interesting case study because there are several different benchmarks that stress different parts of a system. The HPC Challenge benchmarks include HPL, DGEMM, STREAM, PTRANS, RandomAccess, FFTE and communication bandwidth/latency benchmarks. The HPL, DGEMM and FFTE benchmarks are designed to stress the floating point performance of the processor. The RandomAccess measures the rate of integer random updates of memory. The STREAM benchmark measures the sustainable memory bandwidth. Because in this example we are looking at the temperature of a single processor and not the system as a whole, the PTRANS and communication benchmarks will be less interesting [17]. The machine being used in the example is a cluster of 1.4 GHz AMD Opteron processors. The test will monitor the 42 PAPI presets that Opteron processors support, but an Opteron only has 4 counters. The events will need to be multiplexed or there will need to be several runs. Because each run could have different thermal properties, I decided that it would be better to multiplex the events. Because PAPI's multiplexing support only has one event running at any given time, I decided to write a simplified form of multi-way multiplexing. Eleven eventsets were created to hold all 43 possible presets. A timer was setup to swap eventsets in and out every 20,000 microseconds. Once all eventsets have been run, the temperature sensor is read and the values of the counters are scaled and logged to a file. To reduce the possibility that swapping the eventsets out matches the frequency of a program (the same eventset monitors the same section of code every time), the order that the eventsets are started is changed every five iterations. At the beginning of the program, ten different orders are randomly setup and then the program iterates through these when choosing a new order to start eventsets. The new functionality was created in `papi_setup.c` (Which can be found in Appendix D) and this was compiled into the HPCC benchmark to collect the necessary PAPI metrics. Once the code was compiled with PAPI support the benchmark was running with an N size of

12,000.

The results for all the codes were concatenated together to provide a diverse pattern of event counts. The run produced 1631 data points of the 43 events being monitored. Then hierarchical clustering techniques were used to gain insight about the overall cluster structure of the data by creating dendrogram graphs. Figure 5 shows the dendrogram of the HPCC run with all 43 events being monitored and 30 leaf nodes were created. As we are interested in correlation to the temperature we are interested in the cluster of leaf nodes that contain the temperature metric. Leaf node 1 contains the following events: ACPI_THERM, PAPI_RES_STL, PAPI_L2_STM, PAPI_FPU_IDL, PAPI_L2_DCM, PAPI_L2_ICM, PAPI_TLB_TL, PAPI_HW_INT, PAPI_TLB_DM. As leaf node 25 is part of the cluster and similar in height we should take a look at PAPI_TOT_CYC as that event is the only one in that leaf node. When examining the data the temperature varies, but PAPI_TOT_CYC remains the same on each data point, so that metric was removed as possibly relating to the temperature output of the processor. To determine how the other metrics relate to temperature, the data was normalized and graphed. As can be seen in

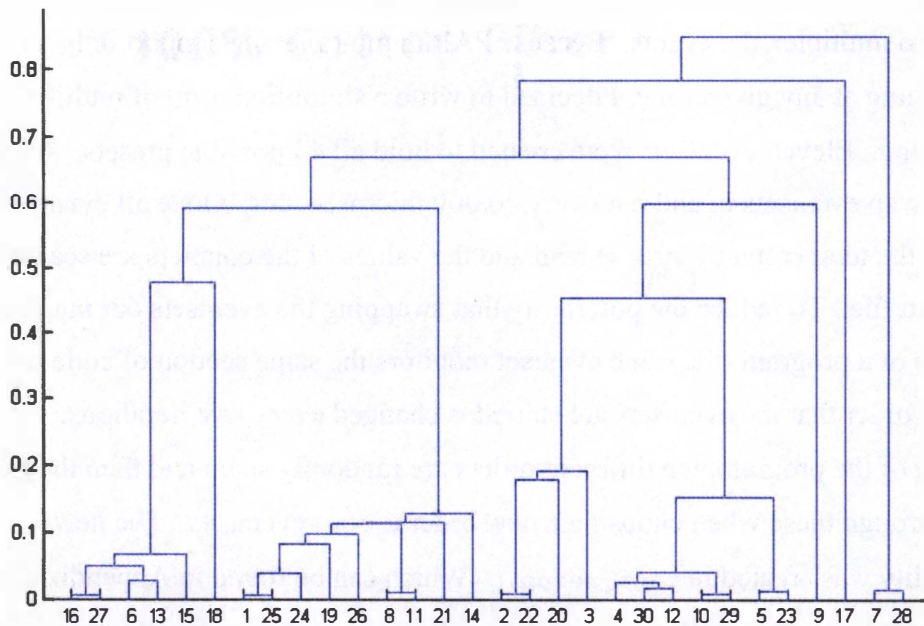


Figure 5. Dendrogram for HPCC using PAPI preset events

figure 6, the PAPI preset metrics are inversely related to the temperature. After closer inspection several PAPI presets have very similar lines. In the graph the PAPI_L2_DCM, PAPI_L2_TCM, PAPI_TLB_TL, PAPI_L2_STM and PAPI_TLB_DM metrics all have similar characteristics in the graph and because of this using just one of these events suffices to estimate the temperature of the processor. After removing these events the graph in figure 7 is created. Each of the remaining events has slightly different peaks and troughs. Thus they will be needed in any algorithm being used to estimate the thermal characteristics of the processor.

Next principal component analysis (PCA) techniques were used on the data to determine if the dendrogram is accurate. First, the matrix was normalized and weighted to bring all the values between 0 and 1. Then the principal components were generated and the scree plot can be seen in figure 8. This shows that we need at least 10 principal components to take into account more than 80% of the variability of the data. The principal components were graphed to see to which components the thermal event was strongly correlated. As seen in figure 9, the thermal event is strongly correlated to the first principal component. The thermal event was moderately correlated to the second principal component. The thermal event had little correlation to the rest of the principal components. As shown in figure 10 by using the first two principal components this only accounts for approximately 36% of the variance in the data, but since the thermal event showed little correlation in the following 8 principal components this should not be a problem.

Examining the first principal component in figure 9 we can see that events 4,5,30,31,32,33,34,35,38,39,40 and 43 are strongly related to the thermal counter (event 1). Events 14, 19, 24, 41 and 42 are inversely related to the thermal counter. Event numbers can be found in Appendix D and their definitions are found in Appendix A. In figure 11 the events are scattered among 4 different quadrants. The thermal event is found in the top left graph and the events surrounding it are the events that are correlated to it. The events found furthest away (bottom right quadrant) are the events that are inversely related to the thermal event. The nine events that are related to the thermal

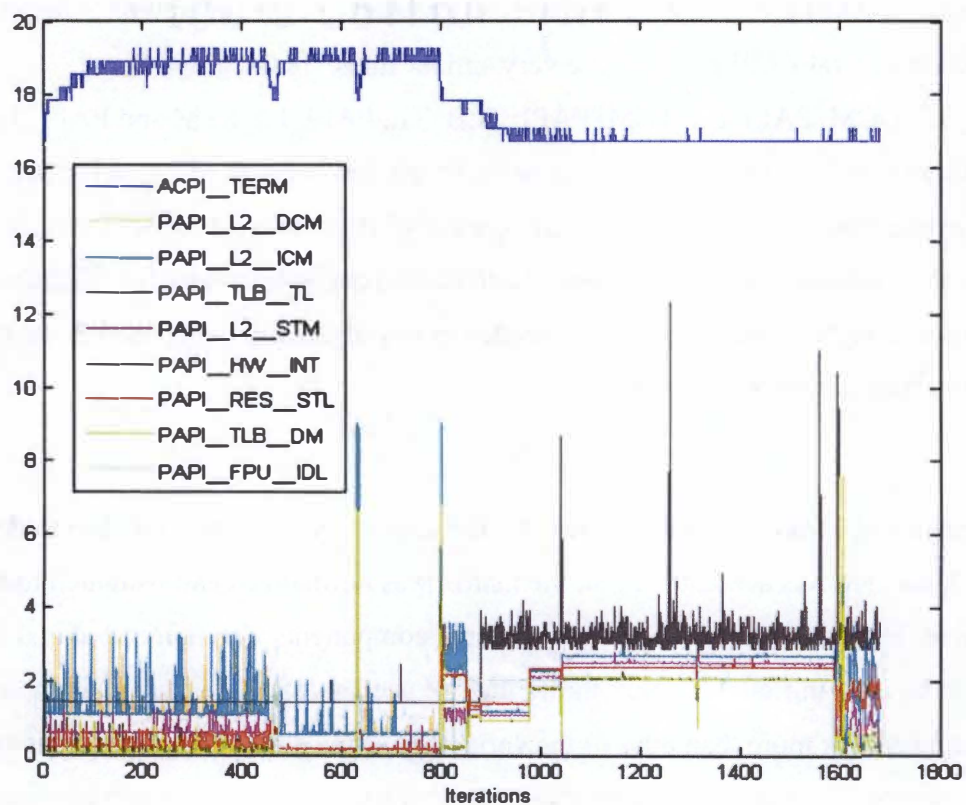


Figure 6. Graph of normalized data from leaf node 1

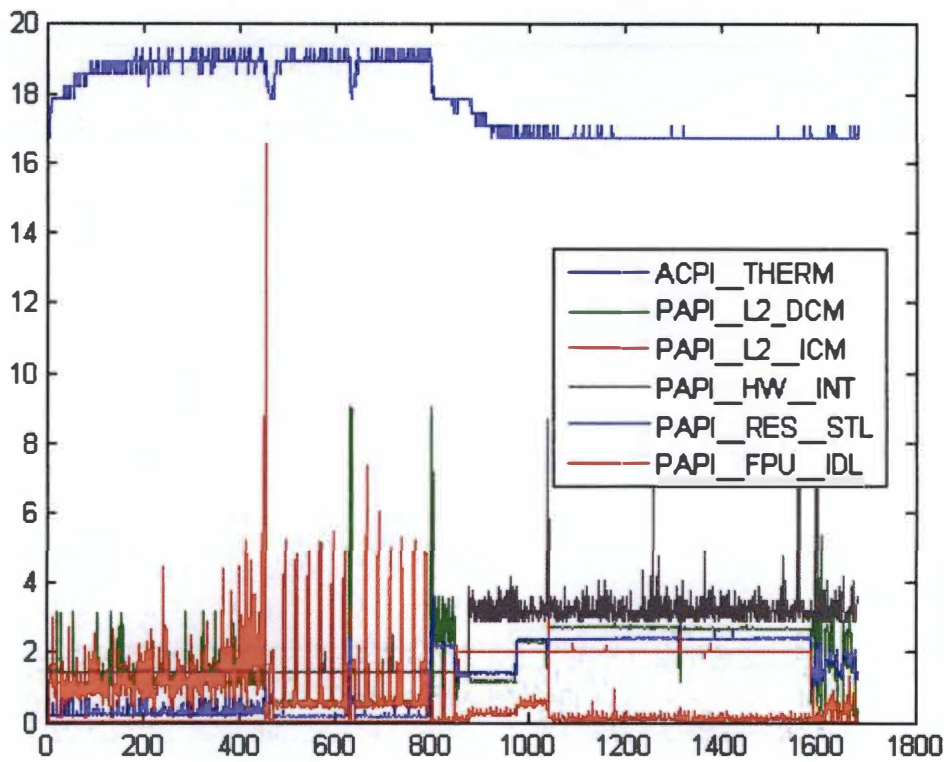


Figure 7. Graph of reduced PAPI preset metrics

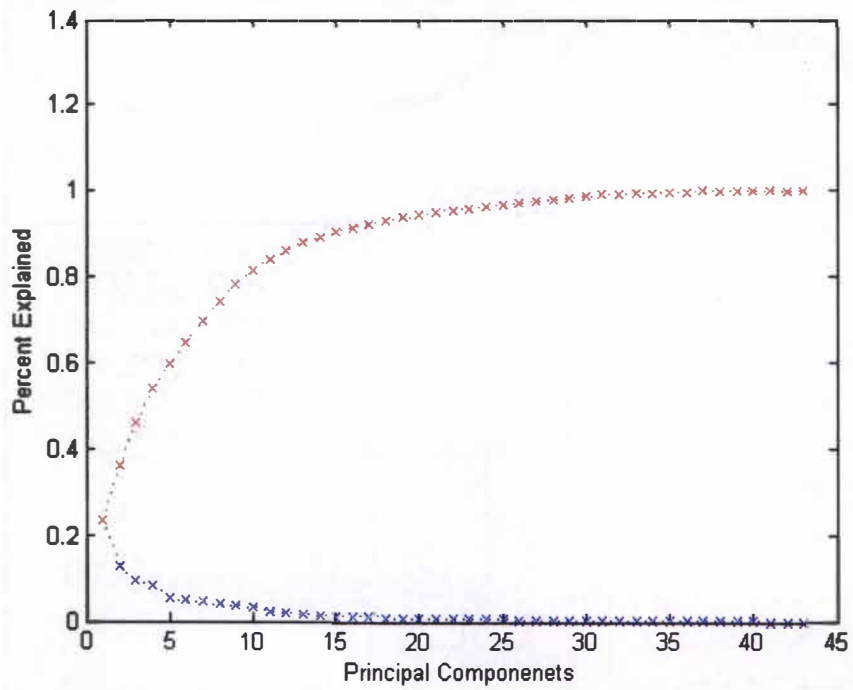


Figure 8. Principal components percent of variance explained

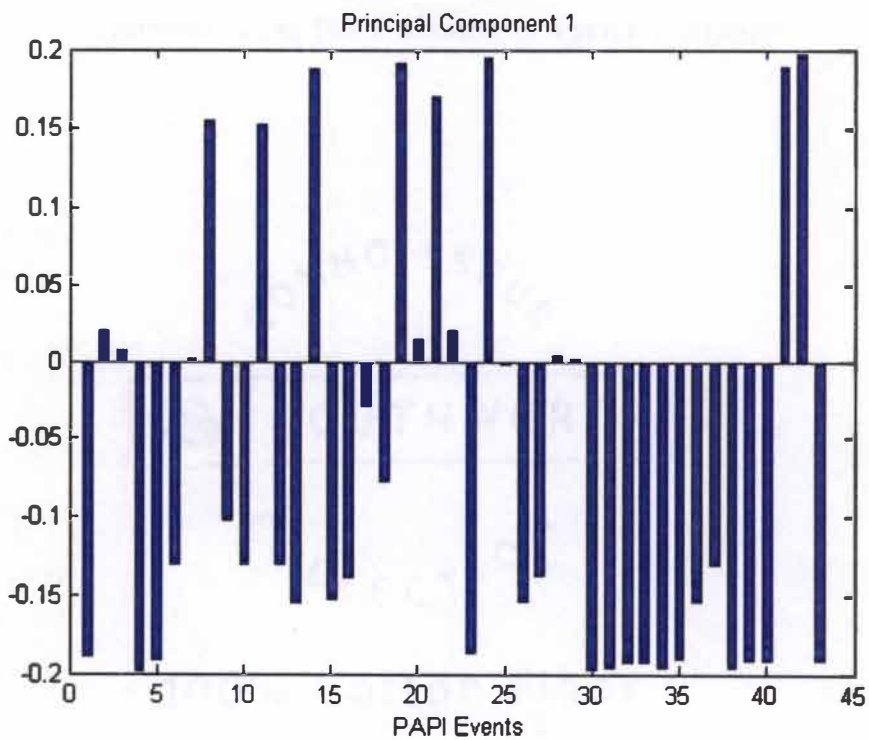


Figure 9. Principal component 1

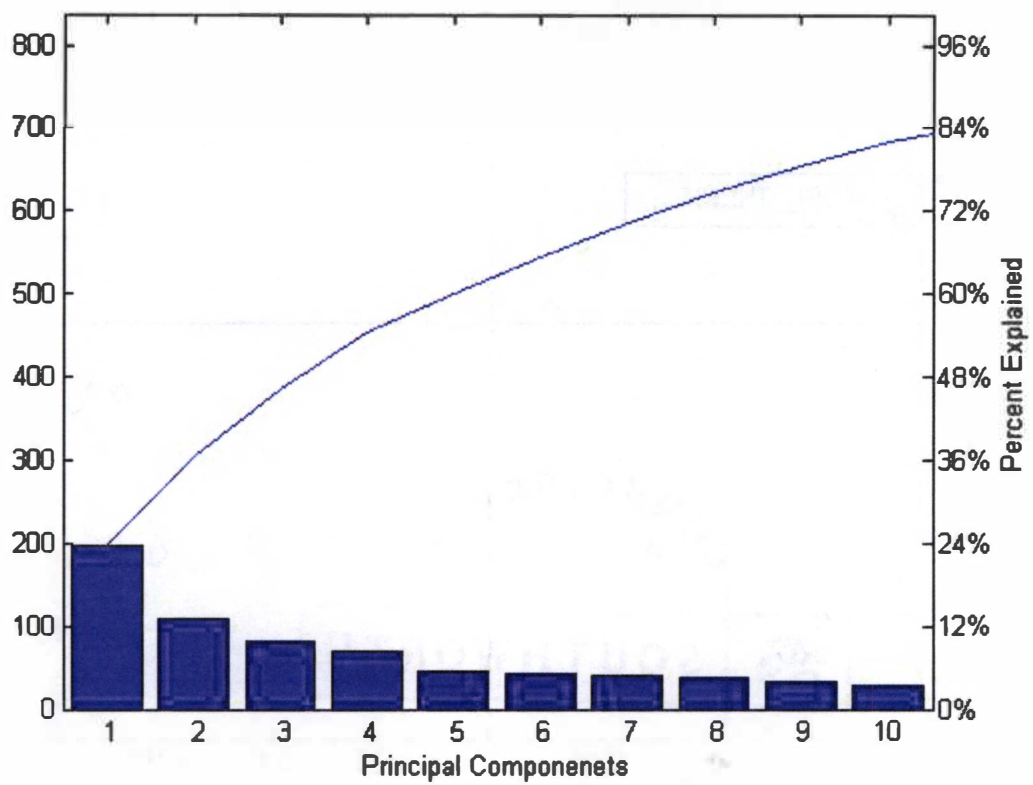


Figure 10. Top 10 principal components percent of variance explained

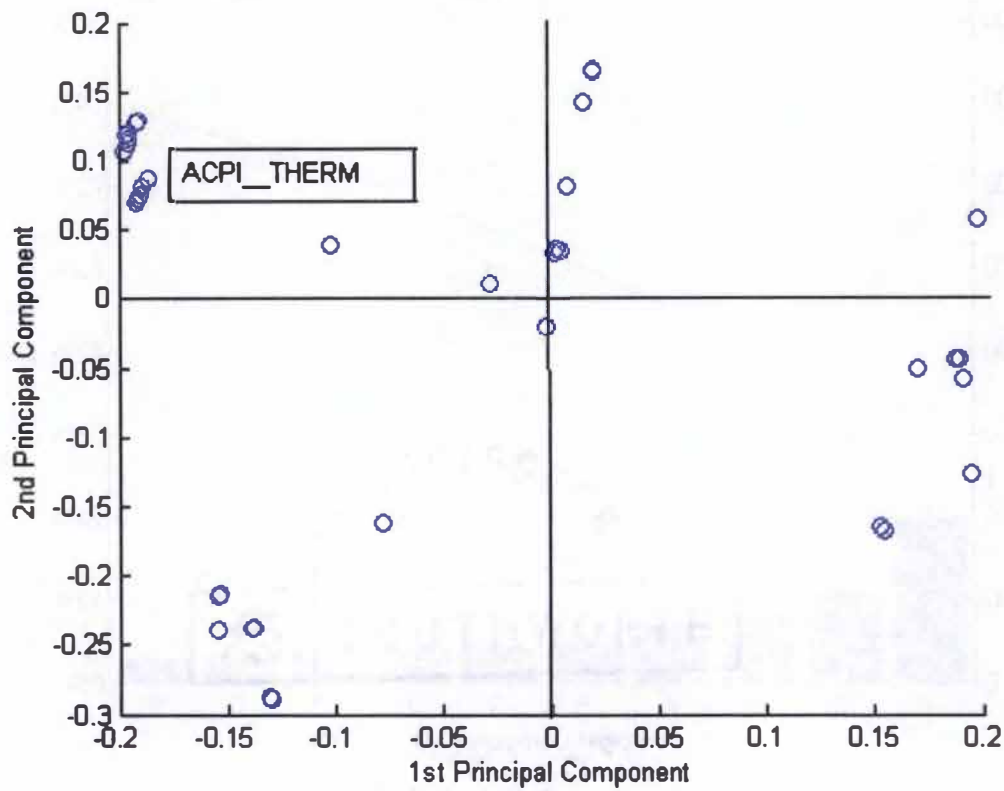


Figure 11. Scatter plot principal component 1 vs. principal component 2

event are: PAPI_L1_ICH, PAPI_L1_ICR, PAPI_L1_DCH, PAPI_L1_DCA, PAPI_TOT_INS, PAPI_VEC_INS, PAPI_FML_INS, PAPI_FP_INS and PAPI_FAD_INS. The nine events that are inversely related to the thermal event are: PAPI_FPU_IDL, PAPI_TLB_DM, PAPI_TLB_TL, PAPI_HW_INT, PAPI_RES_STL, PAPI_L2_TCM, PAPI_L2_DCM, PAPI_L1_TCM, and PAPI_L1_DCM.

Comparing the PCA results to the dendrogram results, there are six events in common: PAPI_FPU_IDL, PAPI_TLB_DM, PAPI_TLB_TL, PAPI_HW_INT, PAPI_RES_STL and PAPI_L2_DCM. In addition, it was shown in the dendrogram results that PAPI_TLB_DM and PAPI_TLB_TL could be removed as they show similar patterns to PAPI_L2_DCM. This leaves 4 events that can be counted all at the same time. While this does not provide a solution to accurately computing the temperature of the processor, the process does show

performance metrics that are strongly correlated to the temperature of the processor.

Also it narrows the events that have to be looked at for estimating the thermal output of a processor. I believe with further research there may be a way to use the extended version of PAPI to automatically determine all the events that affect the temperature of the processor. Using the information, algorithms to estimate the temperature of the processor can be much more quickly developed. There is ongoing research on determining the temperature characteristics of the processor, and the PAPI version that supports multiple substrates should prove to be a valuable tool in their research.

8. Future Work

The work in this thesis has provided a groundwork for simultaneous multiple substrate support. The basic functionality to start, stop and read counters from multiple substrates has been completely integrated into PAPI. However, much of the additional functionality is only supported by the processor substrate. Currently all hardware information comes from the processor substrate. It would make sense that users would want to know specifics about the hardware that the other substrates support as well. This would require changing the exposed API and how the hardware information is stored. These changes would be beneficial for users to determine the speed and revision numbers of network interface cards and other pertinent information. Besides the hardware information, profiling and overflow are only supported on the processor substrate. This functionality is very useful and will be added to support all substrates in the future. There will need to be some restrictions on what substrates can use overflow and profile at the same time. There will need to be design decisions made such as whether can you overflow on multiple substrates at the same time and if hardware and software overflow will be supported at the same time. Currently there is no way to select different hardware devices and hence an exposed API needs to be created to allow users to select what hardware they wish to monitor. But this also entails giving the substrates the ability to determine what hardware is installed that the substrate supports. Finally, more off-processor substrates need to be developed so they can be studied to determine how they can be used for performance optimization, system monitoring and other applications of PAPI.

9. Conclusion

The extension of PAPI to support multiple substrates allows PAPI to be used in many ways that were not possible before. As shown in the example, PAPI can now be used to correlate processor performance metrics with the temperature of the processor. This will allow researchers in the field to use PAPI to quickly access the information needed to formulate algorithms that predict the temperature of the processor. With the ability to monitor off processor counters, PAPI can now be used by MPI libraries to determine network characteristics and other information. As more off-processor substrates are written, PAPI will become usable by a variety of other applications such as system monitoring, application steering, compiler feedback, and many more.

During the course of extending PAPI to allow multiple substrates, there were a few lessons learned. First, even in overhead sensitive applications such as PAPI using function tables to interface with modules is a viable option. Second, if modules will be used in the future, many things can be done to allow multiple modules more easily. This includes allocating all memory, providing routines to determine the size of hardware-dependent data structures, and the use of function tables. Finally when using modules the application should provide interfaces to list the modules and provide information about the modules.

REFERENCES

References

- [1] Federal Plan for High-End Computing, Report of the High-End Computing Revitalization Task Force (HECRTF), Executive Office of the President, Office of Science and Technology Policy, May 2004,
http://www.itrd.gov/pubs/2004_hecrtf/20040702_hecrtf.pdf
- [2] Hennessy, John, Patterson, David, “Computer Architecture – A Quantitative Approach Third Edition”, Morgan Kaufmann, San Francisco, 2002.
- [3] Terpstra, D. “The PAPI Project: An Update”, ICL Talk, March 2004.
- [4] Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D., H. You, M. Zhou, “Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters,” PADTAD Workshop, IPDPS 2003, Nice, France, April 2003.
- [5] Lee, Kyeong-Jae, Skadron, Kevin, “Using Performance Counters for Runtime Temperature Sensing in High-Performance Processors”, Proceedings of the IEEE International Parallel and Distributed Processing Symposium, Denver, Colorado, April 2005.
- [6] Weissel, Andreas, Bellosa, Frank, “Dynamic Thermal Management for Distributed Systems”, Proceedings of the First Workshop on Temperature-Aware Computer Systems (TACS-1), München, Germany, June 2004.
- [7] Identifying Program Power Phase Behavior Using Power Vectors
- [8] Isci, Canturk, Martonosi, Margaret, “Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data”, Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, San Diego, CA, December 2003.
- [9] PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.2) website, <http://www.fz-juelich.de/zam/PCL/>
- [10] HPM ToolKit Website, <http://www.alphaworks.ibm.com/tech/hpmtoolkit>
- [11] perfmon website, <http://www.hpl.hp.com/research/linux/perfmon/>
- [12] Perfctr website, <http://user.it.uu.se/~mikpe/linux/perfctr/>
- [13] Open Speedshop for Linux website, <http://oss.sgi.com/projects/openspeedshop/>
- [14] Advanced Configuration and Power Interface website, <http://www.acpi.info/>

- [15] Squyres, Jeffery, Lumsdaine, Andrew, "The System Services Interface (SSI) to LAM/MPI SSI Version 1.0.0", Technical Report 575, Indiana University, August 2003.
- [16] Top 500 Website, <http://www.top500.org/>
- [17] Luszczek, P., Dongarra, J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., McCalpin, J., Bailey, D., Takahashi, D. "Introduction to the HPC Challenge Benchmark Suite," March, 2005.

APPENDICES

APPENDIX A

PAPI Software Specification

This software specification describes the PAPI 3.0 Release, and is current as of March 08, 2004. It consists of the following sections:

- Introduction to PAPI Constants
- Standardized Event Definitions
- Return Codes
- Low Level API
- High Level API

Introduction to PAPI Constants

The PAPI constants are defined in the header files:

papiStdEventDefs.h
papi.h

The header file **papiStdEventDefs.h** contains platform specific constants. These constants are presented in Table 1: Standardized Event Definitions below. The user should read the documentation that accompanies this table for an explanation of these constants.

The remaining header file **papi.h** contains the PAPI Return Codes, among other internal definitions.

Standardized Event Definitions

The following is a table of hardware events deemed relevant and useful in tuning application performance. These events have identical assignments in the header files on different platforms; *however they may differ in their actual semantics. In addition, all of these events are not guaranteed to be present on all platforms. Please check your platform's documentation carefully.* Note: these values should not be changed by the user.

Value	Symbol	Description
0x80000000	PAPI_L1_DCM	Level 1 data cache misses
0x80000001	PAPI_L1_ICM	Level 1 instruction cache misses
0x80000002	PAPI_L2_DCM	Level 2 data cache misses
0x80000003	PAPI_L2_ICM	Level 2 instruction cache misses
0x80000004	PAPI_L3_DCM	Level 3 data cache misses
0x80000005	PAPI_L3_ICM	Level 3 instruction cache misses
0x80000006	PAPI_L1_TCM	Level 1 total cache misses
0x80000007	PAPI_L2_TCM	Level 2 total cache misses
0x80000008	PAPI_L3_TCM	Level 3 total cache misses
0x80000009	PAPI_CA_SNP	Snoops
0x8000000A	PAPI_CA_SHR	Request for access to shared cache line (SMP)
0x8000000B	PAPI_CA_CLN	Request for access to clean cache line (SMP)
0x8000000C	PAPI_CA_INV	Cache Line Invalidation (SMP)
0x8000000D	PAPI_CA_ITV	Cache Line Intervention (SMP)
0x8000000E	PAPI_L3_LDM	Level 3 load misses
0x8000000F	PAPI_L3_STM	Level 3 store misses
0x80000010	PAPI_BRU_IDL	Cycles branch units are idle
0x80000011	PAPI_FXU_IDL	Cycles integer units are idle
0x80000012	PAPI_FPU_IDL	Cycles floating point units are idle
0x80000013	PAPI_LSU_IDL	Cycles load/store units are idle
0x80000014	PAPI_TLB_DM	Data translation lookaside buffer misses
0x80000015	PAPI_TLB_IM	Instruction translation lookaside buffer misses
0x80000016	PAPI_TLB_TL	Total translation lookaside buffer misses
0x80000017	PAPI_L1_LDM	Level 1 load misses
0x80000018	PAPI_L1_STM	Level 1 store misses
0x80000019	PAPI_L2_LDM	Level 2 load misses
0x8000001A	PAPI_L2_STM	Level 2 store misses

0x8000001B	PAPI_BTAC_M	BTAC miss
0x8000001C	PAPI_PRF_DM	Prefetch data instruction caused a miss
0x8000001D	PAPI_L3_DCH	Level 3 Data Cache Hit
0x8000001E	PAPI_TLB_SD	Translation lookaside buffer shutdowns (SMP)
0x8000001F	PAPI_CSR_FAL	Failed store conditional instructions
0x80000020	PAPI_CSR_SUC	Successful store conditional instructions
0x80000021	PAPI_CSR_TOT	Total store conditional instructions
0x80000022	PAPI_MEM_SCY	Cycles Stalled Waiting for Memory Access
0x80000023	PAPI_MEM_RCY	Cycles Stalled Waiting for Memory Read
0x80000024	PAPI_MEM_WCY	Cycles Stalled Waiting for Memory Write
0x80000025	PAPI_STL_ICY	Cycles with No Instruction Issue
0x80000026	PAPI_FUL_ICY	Cycles with Maximum Instruction Issue
0x80000027	PAPI_STL_CCY	Cycles with No Instruction Completion
0x80000028	PAPI_FUL_CCY	Cycles with Maximum Instruction Completion
0x80000029	PAPI_HW_INT	Hardware interrupts
0x8000002A	PAPI_BR_UCN	Unconditional branch instructions executed
0x8000002B	PAPI_BR_CN	Conditional branch instructions executed
0x8000002C	PAPI_BR_TKN	Conditional branch instructions taken
0x8000002D	PAPI_BR_NTK	Conditional branch instructions not taken
0x8000002E	PAPI_BR_MSP	Conditional branch instructions mispredicted
0x8000002F	PAPI_BR_PRC	Conditional branch instructions correctly predicted
0x80000030	PAPI_FMA_INS	FMA instructions completed
0x80000031	PAPI_TOT_IIS	Total instructions issued
0x80000032	PAPI_TOT_INS	Total instructions executed
0x80000033	PAPI_INT_INS	Integer instructions executed
0x80000034	PAPI_FP_INS	Floating point instructions executed
0x80000035	PAPI_LD_INS	Load instructions executed
0x80000036	PAPI_SR_INS	Store instructions executed
0x80000037	PAPI_BR_INS	Total branch instructions executed
0x80000038	PAPI_VEC_INS	Vector/SIMD instructions executed
0x8000003A	PAPI_RES_STL	Cycles processor is stalled on resource
0x8000003B	PAPI_FP_STAL	Cycles any FP units are stalled
0x8000003C	PAPI_TOT_CYC	Total cycles
0x8000003E	PAPI_LST_INS	Total load/store instructions executed

0x8000003F	PAPI_SYC_INS	Sync. instructions executed
0x80000040	PAPI_L1_DCH	L1 data cache hit
0x80000041	PAPI_L2_DCH	L2 data cache hit
0x80000042	PAPI_L1_DCA	L1 data cache access
0x80000043	PAPI_L2_DCA	L2 data cache access
0x80000044	PAPI_L3_DCA	L3 data cache access
0x80000045	PAPI_L1_DCR	L1 data cache read
0x80000046	PAPI_L2_DCR	L2 data cache read
0x80000047	PAPI_L3_DCR	L3 data cache read
0x80000048	PAPI_L1_DCW	L1 data cache write
0x80000049	PAPI_L2_DCW	L2 data cache write
0x8000004A	PAPI_L3_DCW	L3 data cache write
0x8000004B	PAPI_L1_ICH	L1 instruction cache hits
0x8000004C	PAPI_L2_ICH	L2 instruction cache hits
0x8000004D	PAPI_L3_ICH	L3 instruction cache hits
0x8000004E	PAPI_L1_ICA	L1 instruction cache accesses
0x8000004F	PAPI_L2_ICA	L2 instruction cache accesses
0x80000050	PAPI_L3_ICA	L3 instruction cache accesses
0x80000051	PAPI_L1_ICR	L1 instruction cache reads
0x80000052	PAPI_L2_ICR	L2 instruction cache reads
0x80000053	PAPI_L3_ICR	L3 instruction cache reads
0x80000054	PAPI_L1_ICW	L1 instruction cache writes
0x80000055	PAPI_L2_ICW	L2 instruction cache writes
0x80000056	PAPI_L3_ICW	L3 instruction cache writes
0x80000057	PAPI_L1_TCH	L1 total cache hits
0x80000058	PAPI_L2_TCH	L2 total cache hits
0x80000059	PAPI_L3_TCH	L3 total cache hits
0x8000005A	PAPI_L1_TCA	L1 total cache accesses
0x8000005B	PAPI_L2_TCA	L2 total cache accesses
0x8000005C	PAPI_L3_TCA	L3 total cache accesses
0x8000005D	PAPI_L1_TCR	L1 total cache reads
0x8000005E	PAPI_L2_TCR	L2 total cache reads
0x8000005F	PAPI_L3_TCR	L3 total cache reads
0x80000060	PAPI_L1_TCW	L1 total cache writes

0x80000061	PAPI_L2_TCW	L2 total cache writes
0x80000062	PAPI_L3_TCW	L3 total cache writes
0x80000063	PAPI_FML_INS	Floating Multiply instructions
0x80000064	PAPI_FAD_INS	Floating Add instructions
0x80000065	PAPI_FDV_INS	Floating Divide instructions
0x80000066	PAPI_FSQ_INS	Floating Square Root instructions
0x80000067	PAPI_FNV_INS	Floating Inverse instructions
0x80000068	PAPI_FP OPS	Floating point operations executed

Table 1: Standardized Event Definitions

Return Codes

All of the functions contained in the PAPI APIs return standardized error codes. Values greater than or equal to zero indicate success, less than zero indicates failure.

Value	Symbol	Definition
0	PAPI_OK	No error
-1	PAPI_EINVAL	Invalid argument
-2	PAPI_ENOMEM	Insufficient memory
-3	PAPI_ESYS	A System or C library call failed, please check errno
-4	PAPI_ESBSTR	Substrate returned an error, usually the result of an unimplemented feature
-5	PAPI_ECLOST	Access to the counters was lost or interrupted
-6	PAPI_EBUG	Internal error, please send mail to the developers
-7	PAPI_ENOEVT	Hardware Event does not exist
-8	PAPI_ECNFLCT	Hardware Event exists, but cannot be counted due to counter resource limitations
-9	PAPI_ENOTRUN	No Events or EventSets are currently not counting
-10	PAPI_EISRUN	EventSet is currently running
-11	PAPI_ENOEVST	No such EventSet available
-12	PAPI_ENOTPRESET	Event is not a valid preset
-13	PAPI_ENOCNTR	Hardware does not support performance counters
-14	PAPI_EMISC	'Unknown error' code
-15	PAPI_EPERM	You lack the necessary permissions

Table 2: Return Codes

The Low Level API

The following functions represent the low level portion of the PAPI API. These functions provide greatly increased efficiency and functionality over the high level API presented in the next section. As mentioned in the introduction, the low level API is only as powerful as the substrate upon which it is built. Thus some features may not be available on every platform. The converse may also be true, that more advanced features may be available and defined in the header file. The user is encouraged to read the documentation for each platform carefully.

int PAPI_accum (int EventSet, long long *values)

This function accumulates (adds) the running or stopped counters in EventSet into the values array. In addition, it re-initializes the internal counters to zero.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_add_event (int EventSet, int Event)

This function modifies an existing EventSet. In PAPI3.0, at any time at most one EventSet can be active. Returns the error code PAPI_ENOEVNT if Event cannot be counted on this platform. The addition of a conflicting event to an event set will return an error unless PAPI_SET_MPXRES has been set.

The return value is an integer that indicates whether the call succeeded (a non-negative integer corresponding to the index in the EventInfoArray where this event is stored) or failed (an error code).

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

This function modifies an existing EventSet. The Events contained in *Events will be loaded into the EventSet. In PAPI3.0, at any time at most one EventSet can be active. Returns the error code PAPI_ENOEVNT if Events cannot be counted on this platform. The addition of a conflicting event to an event set will return an error unless PAPI_SET_MPXRES has been set.

The return value is an integer that indicates whether the call succeeded (a non-negative integer corresponding to the index in the EventInfoArray where this event is stored) or failed (an error code).

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

This function adds a native programmable Event to an existing EventSet. Such EventSets can only consist of one event, namely that which is specified in this call. Its semantics are very similar to that of ioctl() system call. inout points to an opaque data structure that is specific to the value in code. Higher level macros may be provided in the header file. Please check the documentation for each substrate. This function has a C binding only. (not yet implemented)

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_cleanup_eventset (int EventSet)

This function effectively cleans the target EventSet. It removes all of the hardware events which have been added to the EventSet. It can then be removed from existence with a call to PAPI_destroy_eventset. The EventSet must be stopped in order for this call to succeed.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_create_eventset (int *EventSet)

This function creates a new EventSet for use. This call is not thread safe.????

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_destroy_eventset (int *EventSet)

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

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_enum_event (int *EventCode, int modifier)

This function updates EventCode to next valid value;

modifier can specify {all / available} for presets, or other values for native tables and may be platform specific (Major groups / all mask bits; P / M / E chip, etc) The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_event_code_to_name (int EventCode, char *out)

This function translates an EventCode from the user into the event name

used by PAPI. Return code is PAPI_OK if successful, and appropriate error code otherwise.

int PAPI_event_name_to_code (char *in, int *out)

This function translates an event name to PAPI event code. Return value is PAPI_OK if successful and appropriate error code otherwise.

long PAPI_get_dmem_info(int option)

This function returns page size in bytes, Resident set size in pages or Size of process image in pages based on the option passed by the user. If the option is illegal, the function returns PAPI_EINVAL.

int PAPI_get_event_info(int EventCode, PAPI_event_info_t * info)

This function fills into a structure of type PAPI_event_info_t, which contains the description about an event, from the eventcode passed by the user if successful, and error if otherwise.

const PAPI_exe_info_t *PAPI_get_executable_info (void)

This function returns a pointer to a structure of type PAPI_exe_info_t, which contains path, name, start and end addresses for the program's text, data, and bss segments. For the definition of the structure, see papi.h Returns pointer to structure of type PAPI_exe_info_t if successful, and NULL if otherwise.

const PAPI_hw_info_t *PAPI_get_hardware_info (void)

This function returns a pointer to a structure of type PAPI_hw_info_t, which contains number of CPUs, nodes, vendor number/name for CPU, CPU revision, and clock speed. For the definition of the structure, see papi.h Returns pointer to structure of type PAPI_hw_info_t if successful, and NULL if otherwise.

int PAPI_get_multiplex(int EventSet)

This function queries the multiplex status of the EventSet. This function returns true (non zero) if the EventSet's status is multiplexing, and false (zero) if the EventSet's status is non-multiplexing.

int PAPI_get_opt (int option, PAPI_option_t *ptr)

This function queries the status of tunable options in the PerfAPI Library. "option" is an input/output parameter. The "ptr" structure is for input and output. Not all options fill the PAPI_option_t structure. This function has a C binding only.

The reader is urged to carefully read the PerfAPI Draft for a complete discussion of PAPI_get_opt. The file papi.h contains definitions for the structures unioned in the PAPI_option_t structure.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

long long PAPI_get_real_cyc (void)

This function returns a value in cycles, and can be used at the beginning and end of a section of code to calculate number of total cycles elapsed while the section executed. Returns time of type long long if successful, appropriate error code otherwise.

long long PAPI_get_real_usec (void)

This function returns a real (wall) time in microseconds, and can be used at the beginning and end of a section of code to calculate real time in microseconds for the section. Returns time of type long long if successful, appropriate error code otherwise.

const PAPI_shlib_info_t *PAPI_get_shared_lib_info(void)

This function returns a pointer to a structure of type PAPI_shlib_info_t, which contains path, name, start and end addresses for the shared libraries used by the program. For the definition of the structure, see papi.h Returns pointer to structure of type PAPI_shlib_info_t if successful, and NULL if otherwise.

int PAPI_get_thr_specific(int tag, void **ptr)

This function returns PAPI_OK if successful, and appropriate error code otherwise. Each thread can store up to PAPI_MAX_THREAD_STORAGE pointers and query them through this function. The pointer is stored in the address pointed by ptr. Tag is used to identify which pointer you want to retrieve.

long long PAPI_get_virt_cyc (void)

This function returns a value in cycles, and can be used at the beginning and end of a section of code to calculate number of virtual (process or thread) cycles elapsed in the section. Returns time of type long long if successful, appropriate error code otherwise.

long long PAPI_get_virt_usec (void)

This function returns a virtual time in microseconds, and can be used at the beginning and end of a section of code to calculate user time in microseconds for the section. Returns time of type long long if successful, appropriate error code otherwise.

int PAPI_is_initialized(void)

This function returns the initialization status of PAPI. It returns 0 if PAPI_library_init has not yet been called, or returns PAPI_LOW_LEVEL_INITED if PAPI_library_init is called by low-level papi functions, or returns PAPI_HIGH_LEVEL_INITED if PAPI_library_init is called by high-level papi functions.

int PAPI_library_init (int version)

This function initializes the PAPI library and has to be called before the low level PAPI can be used. The argument should always be set to PAPI_VER_CURRENT. The reason for this is that you may be linked with a shared library, so this will detect version skew. Don't forget PAPI_thread_init also has to be called before low-level PAPI calls can be used in a threaded application.

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

This function decomposes EventSet into the hardware Events it contains. number is both an input and output parameter.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

void PAPI_lock (int lock)

Grabs access to the PAPI mutex variable. This function is provided to the user to have a platform independent call to an (hopefully) efficiently implemented mutex. This function has no return value. The user can only set lock to PAPI_USR1_LOCK or PAPI_USR2_LOCK. These two locks are equivalent.

int PAPI_multiplex_init (void)

This function enables and initializes multiplex support in the PAPI library. This allows a user to count more events than total physical counters by time-sharing the existing counters at some loss in precision. Applications that make no use of multiplexing do not need to call this routine.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_set_multiplex (int EventSet)

This function converts a standard EventSet created by a call to PAPI_create_eventset() into an event set capable of handling multiplexed

events. This must be done after calling `PAPI_multiplex_init()` , but prior to calling `PAPI_start()`. Events can be added to an `EventSet` either before or after converting it into a multiplexed set, but the conversion must be done prior to using it as a multiplexed set.

The return value is an integer that indicates whether the call succeeded (`PAPI_OK`) or failed (not `PAPI_OK`).

int PAPI_num_hwctrs(void)

This function returns the total physical performance counters.

int PAPI_num_events(int EventSet)

This function returns the total events in the `EventSet` if successful, appropriate error code otherwise.

int PAPI_overflow (int EventSet, int EventCode, int threshold, int flags, PAPI_overflow_handler_t handler)

This function sets up an `EventSet` such that when it is `PAPI_start()`'ed, it begins to register overflows. This `EventSet` may have multiple events as overflow triggers. To turn off overflow, set the threshold to zero.

The return value is an integer that indicates whether the call succeeded (`PAPI_OK`) or failed (not `PAPI_OK`).

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

This function copies length worth of the error description string corresponding to code into destination. The resulting string is always null terminated. If length is 0, then the string is printed on stderr. The return value is an integer that indicates whether the call succeeded (`PAPI_OK`) or failed (not `PAPI_OK`).

int PAPI_profil (unsigned short *buf, unsigned bufsiz, unsigned long offset, unsigned scale, int EventSet, int EventCode, int threshold, int flags)

This function sets the values in the `PAPI_sprofil_t` structure, if profiling is to be enabled for this `EventSet`. The `EventSet` must be in the stopped state for this call to succeed.

The return value is an integer that indicates whether the call succeeded (`PAPI_OK`) or failed (not `PAPI_OK`).

int PAPI_query_event (int EventCode)

This function tests if the event designated by EventCode is supported by the current substrate.

If the answer is yes, the function returns PAPI_OK. If the answer is no, the function returns an error code.

int PAPI_read (int EventSet, long long *values)

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

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_register_thread(void)

This function notifies PAPI that a thread has 'appeared'. We lookup the thread list, if it does not exist we add it to the list.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_remove_event (int EventSet, int EventCode)

This function removes an Event from EventSet.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_remove_events (int EventSet, int *Events, int number)

This function removes the events listed in the Events array from EventSet.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_reset (int EventSet)

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

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_restore (void)

PAPI_save and PAPI_restore are for use with external libraries that wish to preserve the state of PAPI and the hardware counters. For instance a C++ instrumentation library will probably want to call PAPI_save() upon entry to it's functions and PAPI_restore() upon exit. These function calls map to whatever is the most efficient on the underlying platform for

saving and restoring. Returns PAPI_OK if successful, and appropriate error code otherwise. (not yet implemented)

int PAPI_save (void)

see PAPI_restore description (not yet implemented)

int PAPI_set_debug (int level)

This function sets the default debug level for the PAPI library to one of three debug levels as defined in the papi.h header file. The current debug level is internally stored in the PAPI library and is used by the default internal PAPI error handler subroutine. The error handler is called by library routines on the occurrence of recoverable errors.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_set_domain (int domain)

This function sets the execution domain in which events are counted. Here domain is one of the constants PAPI_DOM_USER, PAPI_DOM_MIN, PAPI_DOM_KERNEL, PAPI_DOM_OTHER, PAPI_DOM_ALL, PAPI_DOM_MAX, or PAPI_DOM_HWSPEC as defined in the header file. These constants are listed below and can also be found in the online PerfAPI Draft. The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_set_granularity (int granularity)

This function sets the measurement granularity in which the counters function. By default, the granularity is set to the most restrictive supported by the substrate. Returns PAPI_OK if successful, and appropriate error code otherwise. Granularity settings include per thread, process, process group, current cpu, and all cpus. For more information, see papi.h

int PAPI_set_opt (int option, PAPI_option_t *ptr)

This function sets specific options of the PerfAPI Library, its substrate, or specific EventSets. The PAPI_option_t structure represents a union of all the structures that can be arguments to the different options. In addition, there may exist machine specific options so please check the header file for documentation. This function has a C binding only.

The reader is urged to carefully read the PerfAPI Draft for a complete discussion of PAPI_set_opt. The file papi.h contains definitions for the structures unioned in the PAPI_option_t structure.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_set_thr_specific(int tag, void *ptr)

This function returns PAPI_OK if successful, and appropriate error code otherwise. The ptr is stored in an array with index tag. The ptr can be read late by PAPI_get_thr_specific function with the same tag parameter.

void PAPI_shutdown (void)

This is an exit function used by the PAPI Library to free resources and shut down when certain error conditions arise. This call is not necessary, but allows the user the capability to free memory and resources used by the PAPI Library.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_sprofil (PAPI_sprofil_t *prof, int profcnt, int EventSet, int EventCode, int threshold, int flags)

This function assumes a pre-initialized PAPI_sprofil_t structure and enables profiling for this EventSet. The EventSet must be in the stopped state for this call to succeed.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_start (int EventSet)

This function starts counting all of the hardware events contained in EventSet. All counters are implicitly set to zero before counting.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_state (int EventSet, int *status)

This function returns the state of the entire EventSet in status. If the call succeeds, then status is either PAPI_RUNNING or PAPI_STOPPED.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_stop (int EventSet, long long *values)

This function terminates the counting of all hardware events contained in EventSet. In addition, the counters contained in that EventSet are copied into the values array.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

char *PAPI_strerror (int code)

This function returns the corresponding English error string from the passed code. Returns NULL if code is invalid.

int PAPI_thread_init (unsigned long int (*id_fn)(void))

This function initializes thread support. The argument is a pointer to a function that returns the Thread ID of the currently running thread.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

unsigned long int PAPI_thread_id (void)

This function calls the thread id function registered by PAPI_thread_init().

The return value is an unsigned long integer representing the thread id or (unsigned long int)-1.

void PAPI_unlock (int lock)

Unlocks the mutex acquired by a call to PAPI_lock() . This function has no return value.

int PAPI_write (int EventSet, long long *values)

This function assigns the values contained in the values array to the internal counters of the Events contained in the EventSet. Returns PAPI_OK if successful, and appropriate error code otherwise.

The High Level API

The simple interface implemented by the following eight routines allows the user to access and count specific hardware events. It should be noted that this API can be used *in conjunction with the low level API*. However, the high level API by itself is only able to access those events countable simultaneously by the underlying hardware. Note that the high level interface performs initialization implicitly and is *not thread safe*. Under the covers it calls `PAPI_library_init(PAPI_VER_CURRENT)` and `PAPI_thread_init(NULL)`.

`int PAPI_accum_counters(long long *values, int array_len)`

Add the running counter values to the values in the values array. This call implicitly re-initializes the counters to zero and lets them continue to run upon return. Returns `PAPI_OK` if successful, and an appropriate error code otherwise.

`int PAPI_num_counters(void)`

This function returns the optimal length of the values array for the high level functions. This value corresponds to the number of hardware counters supported by the current substrate.

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

Read the running counter values into the values array. This call implicitly re-initializes the counters to zero and lets them continue to run upon return. Returns `PAPI_OK` if successful, and an appropriate error code otherwise.

`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 a 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. The length of this array should be no longer than `PAPI_MAX_EVENTS`. Returns `PAPI_OK` if successful, and an appropriate error code otherwise.

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

Stop the running counters and copy the counts into the values array. This is to be used in conjunction with

PAPI_start_counters. Returns PAPI_OK if successful, and an appropriate error code otherwise.

```
int PAPI_flips(float *rtime, float *ptime, long_long *flpins, float *mflips )
```

Simplified single call to measure the number of floating point instructions executed and the MegaFlip rate, defined as the number of floating point **instructions** per microsecond. Note that not all floating point instructions are created equal: Some platforms implement a floating point multiply/add (FMA) as a single instruction; and a floating point *square root* is often more costly than a floating point *add*. Caution must be used in comparing measurements on different platforms. Each call to PAPI_flips returns the real (clock) time and process time, floating point instructions executed, and MFlips for the period since the last call. A call to PAPI_flips with flpins = -1 resets the counters. Returns PAPI_OK if successful, and an appropriate error code otherwise.

```
int PAPI_flops(float *rtime, float *ptime, long_long *flpops, float *mflops )
```

Simplified single call to measure the theoretical floating point operations executed rather than simple instructions, and the MegaFlop rate, defined as the number of floating point **operations** per microsecond. It uses the PAPI_FP_OPS event, which attempts to 'correctly' account for, e.g., FMA undercounts and FP Store overcounts, etc. Caution must be used in comparing measurements on different platforms. Each call to PAPI_flops returns the real (clock) time and process time, floating point operations executed, and MFlops for the period since the last call. A call to PAPI_flops with flpops = -1 resets the counters. Returns PAPI_OK if successful, and an appropriate error code otherwise.

```
int PAPI_ipc(float *rtime, float *ptime, long_long *ins, float *ipc)
```

Simplified single call to measure the information on the instruction rate using the PAPI_TOT_INS event. Returns PAPI_OK if successful, and an appropriate error code otherwise.

APPENDIX B

vector_test.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "papi.h"

typedef void (*func_t) (int *i);
extern void tester(int *i);

void vector_version();
void direct_version();

func_t func_table[10];

#define RUNS    100000000

int dummy_var;
long long start, end;

int main(int argc, char **argv){
    int vector=0;
    dummy_var = rand();

    if ( argc > 1 && !strcmp(argv[1], "vector") )    vector = 1;

    if ( PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT ){
        fprintf(stderr, "PAPI failed to initialized\n");
        exit(1);
    }

    if ( vector ){
        vector_version();
        printf("Vectorized Version:\n");
    }
    else {
        direct_version();
        printf("Direct Version:\n");
    }
    printf("Cycles per function call: %lf cycles\n", ((double)(end-
start))/((double)RUNS));
    printf("Dummy variable: %d\n", dummy_var);
    exit(0);
}

void vector_version(){
    int i;

    func_table[5] = tester;
    start = PAPI_get_real_cyc();
    for(i=0;i<RUNS;i++){
        func_table[5](&dummy_var);
```

```

    }
    end = PAPI_get_real_cyc();
}

void direct_version(){
    int i;

    start = PAPI_get_real_cyc();
    for(i=0;i<RUNS;i++){
        tester(&dummy_var);
    }
    end = PAPI_get_real_cyc();
}

```

dummy.c:

```

#include <stdlib.h>

void tester(int *la){
}

```

APPENDIX C

papi_setup.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <signal.h>
#include <errno.h>
#include "papi.h"
#define NUM_CNTRS      4
#define NUM_EVENTSETS  11
#define NUM_RANDOM     10
#define LOG_FILE "papi_log_info"
#define RAND_INTERVAL  5

const int event_list[NUM_EVENTSETS][NUM_CNTRS]={
{ PAPI_BR_TKN, PAPI_BR_MSP, PAPI_TOT_INS, PAPI_FP_INS },
{ PAPI_L1_DCM, PAPI_L1_ICM, PAPI_L2_DCM, PAPI_L2_ICM },
{ PAPI_L1_TCM, PAPI_L2_TCM, PAPI_L2_TCA, PAPI_L2_TCH },
{ PAPI_TLB_TL, PAPI_L1_LDM, PAPI_L1_STM, PAPI_TLB_IM },
{ PAPI_L2_LDM, PAPI_L2_STM, PAPI_STL_ICY, PAPI_HW_INT },
{ PAPI_BR_INS, PAPI_VEC_INS, PAPI_RES_STL, PAPI_TOT_CYC },
{ PAPI_L2_DCR, PAPI_L2_DCW, PAPI_L2_ICH, PAPI_L2_ICA },
{ PAPI_L1_TCA, PAPI_L1_TCH, PAPI_L1_ICR, PAPI_L1_ICA },
{ PAPI_L1_DCH, PAPI_FML_INS, PAPI_L2_DCH, PAPI_L2_DCA },
{ PAPI_L1_DCA, PAPI_FAD_INS, PAPI_FP_OPS, PAPI_TLB_DM },
{ PAPI_FPU_IDL, PAPI_L1_ICH, PAPI_NULL, PAPI_NULL }
};

/* Proto-types */
static void papi_log_counters(int sig, siginfo_t *info, void *ptr);
int papi_setup( );
static void log_header();

/* Global Variables */
static int eventset[NUM_EVENTSETS];
static int counts[NUM_EVENTSETS];
static int tot_counts;
static int ext_eventset;
static struct itimerval time_slice;
static FILE *fp;
static void randomize_events();
static long_long values[NUM_CNTRS*NUM_EVENTSETS];
static long_long ext_values;
static int idx;
static int ridx;
static int eset[NUM_RANDOM][NUM_EVENTSETS];
static int randomize;
static sigset_t set;

/*
 * This function takes care of setting up PAPI, adding the
```

```

* events, creating the timer and signal handler and then
* starting the counters
*/
int papi_setup( ){
    int retval;
    int i,j;
    char buf[PAPI_HUGE_STR_LEN];
    struct sigaction act;
    time_t tv;
    struct tm *ltm;

    idx = 0;
    ridx = 0;
    randomize = 0;

    time(&tv);
    ltm = localtime(&tv);

    srandom(ltm->tm_sec);
    randomize_events();
    sigemptyset( &set );
    sigaddset(&set, SIGVTALRM);

    if ( !ltm ) {
        perror("local_time()");
        return(-1);
    }
    memset(values, 0, sizeof(long_long)*NUM_CNTRS*NUM_EVENTSETS);
    memset(counts, 0, sizeof(int)*NUM_EVENTSETS);
    tot_counts = 0;
    sprintf(buf, "%s_%d_%d_%d_%d_%d", LOG_FILE, ltm->tm_mon, ltm-
>tm_mday,
        ltm->tm_hour, ltm->tm_min, ltm->tm_sec);
    if ( (fp=fopen(buf,"w")) == NULL ){
        perror("fopen()");
        return(-1);
    }

    /*Initialize the PAPI library, create the eventsets and add events */
    if ((retval=PAPI_library_init(PAPI_VER_CURRENT)) !=
PAPI_VER_CURRENT){
        fprintf(stderr, "Error initializing PAPI library: %d\n", retval);
        return(-1);
    }

    for(i=0;i<NUM_EVENTSETS;i++){
        eventset[i] = PAPI_NULL;
        if ((retval=PAPI_create_eventset(&eventset[i])) != PAPI_OK){
            fprintf(stderr, "Error creating eventset #%d\n", i);
            goto FAILURE_RETURN;
        }
    }
    for(i=0;i<NUM_EVENTSETS;i++){
        for(j=0;j<NUM_CNTRS;j++){
            if ( event_list[i][j] == PAPI_NULL ) continue;

```

```

if ((retval=PAPI_add_event(eventset[i],event_list[i][j]))!=PAPI_OK){
    if ( PAPI_event_code_to_name(event_list[i][j], buf)<0 ){
        fprintf(stderr, "Error adding Event #%d (%d) name lookup
failed.\n", j, event_list[
i][j]);
    }
    else {
        fprintf(stderr, "Error adding Event #%d (%s) to eventset
%d\n",
            j, buf, i);
    }
    goto FAILURE_RETURN;
}
PAPI_event_code_to_name(event_list[i][j], buf);
}
}

ext_eventset = PAPI_NULL;

if ((retval=PAPI_create_ext_eventset(&ext_eventset, 1)) != PAPI_OK ){
    fprintf(stderr, "Error creating extended eventset\n");
    goto FAILURE_RETURN;
}

if ((retval=PAPI_event_name_to_code("ACPI_TEMP", &i)) != PAPI_OK){
    fprintf(stderr, "Error looking up event name ACPI_TEMP\n");
    goto FAILURE_RETURN;
}

if ((retval=PAPI_add_event(ext_eventset, i)) != PAPI_OK ){
    fprintf(stderr, "Error adding extended event: ACPI_TEMP\n");
    goto FAILURE_RETURN;
}

/* Setup the signal handler */
memset(&act, 0x00, sizeof(struct sigaction));
act.sa_sigaction = papi_log_counters;
act.sa_flags = SA_RESTART | SA_SIGINFO;
if ( sigaction(SIGVTALRM, &act, NULL)<0 ){
    perror("sigaction()");
    goto FAILURE_RETURN;
}

log_header();

/* Setup the timer 10 millisecond time slices */
time_slice.it_interval.tv_sec = 0;
time_slice.it_interval.tv_usec = 20000;
time_slice.it_value.tv_sec = 0;
time_slice.it_value.tv_usec = 20000;

if ( PAPI_start(ext_eventset) != PAPI_OK ){
    fprintf(stderr, "Error starting extended eventset\n");
    goto FAILURE_RETURN;
}

```

```

    }

    counts[eset[ridx][idx]]++;
    tot_counts++;
    if ( PAPI_start(eventset[eset[ridx][idx]]) != PAPI_OK ){
        fprintf(stderr, "Error starting eventset %d\n", eset[ridx][idx]);
        goto FAILURE_RETURN;
    }

    if ( setitimer(ITIMER_VIRTUAL,&time_slice,NULL) == -1 ){
        perror("setitimer()");
        goto FAILURE_RETURN;
    }

    return(0);

FAILURE_RETURN:
    PAPI_shutdown();
    return(-1);
}

/* Close the fp and shutdown PAPI */
int papi_finalize(){
    PAPI_shutdown();
    fflush(fp);
    fclose(fp);
    return(0);
}

/*
 * This handles writing out the log information
 * reading counters and re-randomizing the counters when necessary
 */
static void papi_log_counters(int sig, siginfo_t *info, void *ptr){
    int i,j;
    long_long new_values[4];

    sigprocmask(SIG_BLOCK,&set, NULL);

    if ( PAPI_stop(eventset[eset[ridx][idx]], new_values)!=PAPI_OK){
        fprintf(stderr, "Error stopping eventset: %d\n", eset[ridx][idx]);
    }
    for(j=0;j<4;j++){
        values[(eset[ridx][idx]*NUM_CNTRS)+j] += new_values[j];
    }

    idx++;

    if ( idx == NUM_EVENTSETS ){
        /* Simplified scaling */
        randomize++;
        if ( randomize == RAND_INTERVAL ){
            PAPI_read(ext_eventset, &ext_values);
            fprintf(fp, "%lld ", ext_values);
            for(i=0;i<NUM_EVENTSETS; i++){
                for(j=0;j<NUM_CNTRS;j++){

```

```

        if ( event_list[i][j] == PAPI_NULL ) continue;
        if ( i==(NUM_EVENTSETS-1) && (j==(NUM_CNTRS-1) ||
(j<(NUM_CNTRS-1) && event_list[i][j
+1] == PAPI_NULL)))
            fprintf(fp, "%lld", values[(i*NUM_CNTRS)+j]*(tot_counts-
counts[i]));
        else
            fprintf(fp, "%lld,", values[(i*NUM_CNTRS)+j]*(tot_counts-
counts[i]));
    }
}
memset(values, 0, sizeof(long_long)*NUM_CNTRS*NUM_EVENTSETS);
ridx++;
if ( ridx == NUM_RANDOM ) ridx = 0;
randomize = 0;
fprintf(fp, "\n");
memset(counts, 0, sizeof(int)*NUM_EVENTSETS);
tot_counts=0;
}

idx = 0;
}

counts[eset[ridx][idx]]++;
tot_counts++;
if ( PAPI_start(eventset[eset[ridx][idx]]) != PAPI_OK )
    fprintf(stderr, "Error starting eventset: %d\n", eset[ridx][idx]);
sigprocmask(SIG_UNBLOCK,&set, NULL);
}

void papi_log_message(char *buf){
    fprintf(fp, "%s\n", buf);
}

/*
 * Print the events we are using
 */
static void log_header(){
    char buf[PAPI_HUGE_STR_LEN];
    int i,j;

    fprintf(fp, "ACPI_THERM ");
    for(i=0;i<NUM_EVENTSETS;i++){
        for(j=0;j<NUM_CNTRS;j++){
            if ( event_list[i][j] == PAPI_NULL ) continue;
            PAPI_event_code_to_name(event_list[i][j], buf);
            fprintf(fp, "%s,", buf);
        }
    }
    fprintf(fp, "\n");
    fflush(fp);
    return;
}

static void randomize_events(){

```

```

int i,j,nidx,picked[NUM_EVENTSETS];
int tmp=0;

for( j = 0; j< NUM_RANDOM; j++ ){
DO_LOOP:
    memset(picked, 0, sizeof(int)*NUM_EVENTSETS);
    for(i=0;i<NUM_EVENTSETS;i++){
        nidx = (random()%NUM_EVENTSETS);

        for(tmp=0;picked[nidx];tmp++) {
            if (++nidx >= NUM_EVENTSETS ) nidx=0;
            if ( tmp >= NUM_EVENTSETS ) {
                for(i=0;i<NUM_EVENTSETS;i++)
                    eset[j][i] = i;
                randomize = 0;
                j++;
                goto DO_LOOP;
            }
        }

        picked[nidx] = 1;
        eset[j][i] = nidx;
    }
}
randomize = 0;
}

```

APPENDIX D

1. ACPI_THERM
2. PAPI_BR_TKN
3. PAPI_BR_MSP
4. PAPI_TOT_INS
5. PAPI_FP_INS
6. PAPI_L1_DCM
7. PAPI_L1_ICM
8. PAPI_L2_DCM
9. PAPI_L2_ICM
10. PAPI_L1_TCM
11. PAPI_L2_TCM
12. PAPI_L2_TCA
13. PAPI_L2_TCH
14. PAPI_TLB_TL
15. PAPI_L1_LDM
16. PAPI_L1_STM
17. PAPI_TLB_IM
18. PAPI_L2_LDM
19. PAPI_L2_STM
20. PAPI_STL_ICY
21. PAPI_HW_INT
22. PAPI_BR_INS
23. PAPI_VEC_INS
24. PAPI_RES_STL
25. PAPI_TOT_CYC
26. PAPI_L2_DCR
27. PAPI_L2_DCW
28. PAPI_L2_ICH
29. PAPI_L2_ICA
30. PAPI_L1_TCA
31. PAPI_L1_TCH
32. PAPI_L1_ICR
33. PAPI_L1_ICA
34. PAPI_L1_DCH
35. PAPI_FML_INS
36. PAPI_L2_DCH
37. PAPI_L2_DCA
38. PAPI_L1_DCA
39. PAPI_FAD_INS
40. PAPI_FP_OPS
41. PAPI_TLB_DM
42. PAPI_FPU_IDL
43. PAPI_L1_ICH

VITA

Kevin Scott London was born in Charlotte, North Carolina on March 8, 1971. He was raised in Signal Mountain, Tennessee and graduated from Baylor High School in Chattanooga, TN in 1989. He graduated from The University of Tennessee at Knoxville with a Bachelor of Science Degree in Computer Science in 2003. He continued his education at The University of Tennessee at Knoxville where he graduated with a Master of Science Degree in Computer Science in 2005.

...the ... of ...
...the ... of ...
...the ... of ...
...the ... of ...
...the ... of ...

UNIVERSITY OF
SOUTHWORTH

4533 0543 5

11/09/05

MAB

