PAPI USER'S GUIDE

TABLE OF CONTENTS

PREFACE

INTENDED AUDIENCE

This document is intended to provide the PAPI user with a discussion of how to use the different components and functions of PAPI . The intended users are application developers and performance tool writers who need to access performance data to tune and model application performance. The user is expected to have some level of familiarity with either the C or Fortran programming language.

ORGANIZATION OF THIS DOCUMENT

II. INTRODUCTION TO PAPI

This section provides an introduction to PAPI by describing the project, its motivation, and its architecture.

III. HOW TO INSTALL PAPI ONTO YOUR SYSTEM

This section provides an installation guide for PAPI. It states the necessary steps in order to install PAPI on the various supported operating systems.

IV. C AND FORTRAN CALLING INTERFACES

This section states the header files in which function calls are defined and the form of the function calls for both the C and Fortran calling interfaces. Also, it provides a table that shows the relation between certain pseudo-types and Fortran variable types.

V. EVENTS

This section provides an explanation of events as well as an explanation of native and preset events. The preset query and translation functions are also discussed in this section. There are code examples using native events, preset query, and preset translation with the corresponding output.

VI. PAPI COUNTER INTERFACES

This section discusses the high-level and low-level interfaces in detail. The initialization and functions of these interfaces are also discussed. Code examples along with the corresponding output are included as well.

VII. PAPI TIMERS

This section explains the PAPI functions associated with obtaining real and virtual time from the platform's timers. Code examples along with the corresponding output are included as well.

VIII. PAPI SYSTEM INFORMATION

This section explains the PAPI functions associated with obtaining hardware and executable information. Code examples along with the corresponding output are included as well.

IX. ADVANCED PAPI FEATURES

This section discusses the advanced features of PAPI, which includes multiplexing, threads, MPI, overflows, and statistical profiling. The functions that are use to implement these features are also discussed. Code examples along with the corresponding output are included as well.

X. PAPI ERROR HANDLING

This section discusses the various negative error codes that are returned by the PAPI functions. A table with the names, values, and descriptions of the return codes are given as well as a discussion of the PAPI function that can be used to convert error codes to error messages along with a code example with the corresponding output.

XI. PAPI MAILING LISTS

This section provides information on PAPI two mailing lists for the users to ask various questions about the project.

XII. APPENDICES

These appendices provide various listings and tables, such as: a table of preset events and the platforms on which they are supported, a table of PAPI supported tools, more information on native events, multiplexing, overflow, and etc.

DOCUMENT CONVENTION

handle_error(1)

A function that passes the argument of 1 that the user should write to handle errors.

INTRODUCTION TO PAPI

WHAT IS PAPI?

PAPI is an acronym for **P**erformance **A**pplication **P**rogramming **I**nterface. The PAPI Project is being developed at the University of Tennessee's Innovative Computing Laboratory in the Computer Science Department. This project was created to design, standardize, and implement a portable and efficient API (Application Programming Interface) to access the hardware performance counters found on most modern microprocessors.

BACKGROUND

Hardware counters exist on every major processor today, such as Intel Pentium, IA-64, AMD Athlon, and IBM POWER series. These counters can provide performance tool developers with a basis for tool development and application developers with valuable information about sections of their code that can be improved. However, there are only a few APIs that allow access to these counters, and most of them are poorly documented, unstable, or unavailable. In addition, performance metrics may have different definitions and different programming interfaces on different platforms.

These considerations motivated the development of the PAPI Project. Some goals of the PAPI Project are as follows:

- To provide a solid foundation for cross platform performance analysis tools
- To present a set of standard definitions for performance metrics on all platforms
- To provide a standardize API among users, vendors, and academics
- To be easy to use, well documented, and freely available

ARCHITECTURE

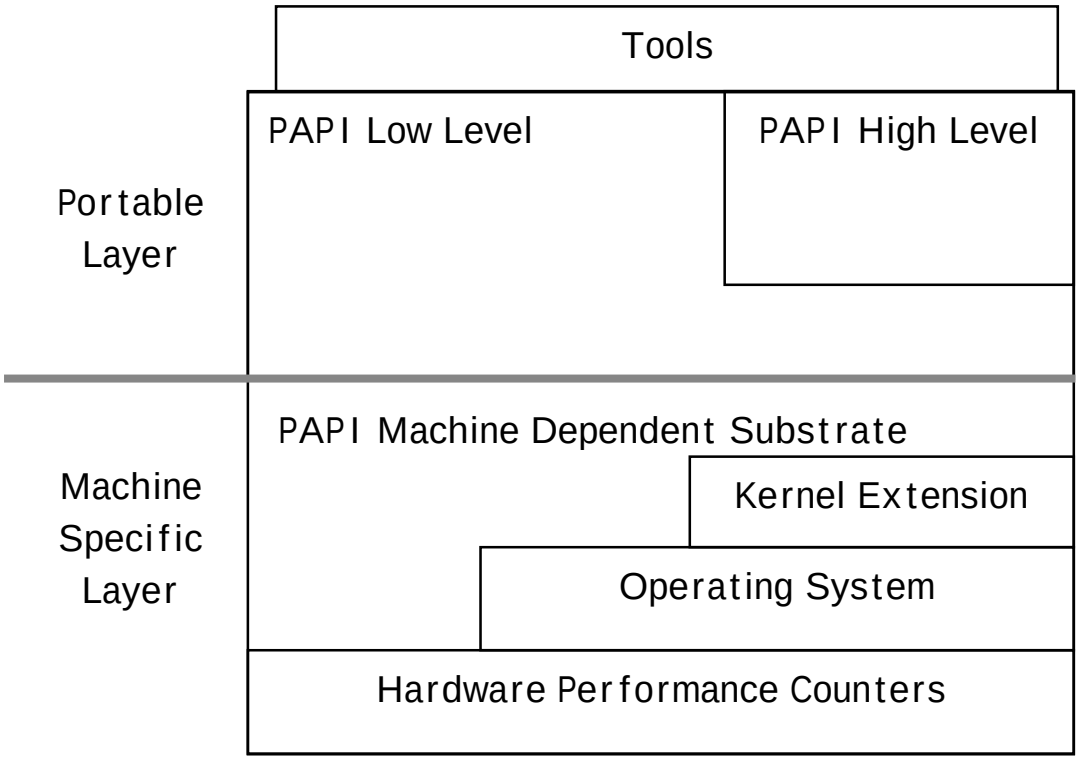
The **Figure** below shows the internal design of the PAPI architecture. In this figure, we can see the two layers of the architecture:

The **Portable Layer** consists of the API (low level and high level) and machine independent support functions.

The **Machine Specific Layer** defines and exports a machine independent interface to machine dependent functions and data structures. These functions are defined in the substrate layer, which uses kernel extensions, operating system calls, or assembly language to access the hardware performance counters. PAPI uses the most efficient and flexible of the three, depending on what is available.

PAPI strives to provide a uniform environment across platforms. However, this is not always possible. Where hardware support for features, such as overflows and multiplexing is not supported, PAPI implements the features in software where possible. Also, processors do not support the same metrics, thus you can monitor different events depending on the processor in use. Therefore, the interface

remains constant, but how it is implemented can vary. Throughout this guide, implementation decisions will be documented where it can make a difference to the user, such as overhead costs, sampling, and etc.



On some of the systems that PAPI supports (see Appendix D), you can install PAPI right out of the box without any additional setup. Others require drivers or patches to be installed first.

The general installation steps are below, but first find your particular Operating System's section of the /papi/INSTALL file for current information on any additional steps that may be necessary.

General Installation

1. Pick the appropriate Makefile.<arch> for your system in the papi source distribution, edit it (if necessary) and compile.

   % make -f Makefile.<arch>

2. Check for errors. Look for the libpapi.a and libpapi.so in the current directory. Optionally, run the test programs in the 'ftests' and 'tests' directories.
   Not all tests will succeed on all platforms.

   % ./run_tests.sh

   This will run the tests in quiet mode, which will print PASSED, FAILED, or SKIPPED. Tests are SKIPPED if the functionality being tested is not supported by that platform.

3. Create a PAPI binary distribution or install PAPI directly.

   To directly install PAPI from the build tree:

   % make -f Makefile.<arch> DESTDIR=<install-dir> install

   Please use an absolute pathname for <install-dir>, not a relative pathname.

   To create a binary kit, papi-<arch>.tgz:

   % make -f Makefile.<arch> dist

C AND FORTRAN CALLING INTERFACES

PAPI is written in C. The function calls in the C interface are defined in the header file, **papi.h** and consist of the following form:

<returned data type> PAPI_*function_name*(*arg1, arg2,*…)

The function calls in the Fortran interface are defined in the header file, **fpapi.h** and consist of the following form:

PAPIF_*function_name*(*arg1, arg2,* …, check)

As you can probably see, the C function calls have equivalent Fortran function calls (PAPI_<call> becomes PAPIF_<call>). Well, this is true for most function calls, except for the functions that return C pointers to structures, such as PAPI_get_opt and PAPI_get_executable_info, which are either not implemented in the Fortran interface, or implemented with different calling semantics. **In the function calls of the Fortran interface, the return code of the corresponding C routine is returned in the argument, *check*.**

For most architectures, the following relation holds between the pseudo-types listed and Fortran variable types:

Pseudo-type	Fortran type	Description
C_INT	INTEGER	Default Integer type
C_FLOAT	REAL	Default Real type
C_LONG_LONG	INTEGER*8	Extended size integer
C_STRING	CHARACTER*(PAPI_MAX_STR_LEN)	Fortran string
C_INT FUNCTION	EXTERNAL INTEGER FUNCTION	Fortran function returning integer result

Array arguments must be of sufficient size to hold the input/output from/to the subroutine for predictable behavior. The array length is indicated either by the accompanying argument or by internal PAPI definitions.

Subroutines accepting **C_STRING** as an argument are on most implementations capable of reading the character string length as provided by Fortran. In these implementations, the string is truncated or space padded as necessary. For other implementations, the length of the character array is assumed to be of sufficient size. No character string longer than **PAPI_MAX_STR_LEN** is returned by the PAPIF interface.

For more information on all of the function calls and their job descriptions, see Appendix B for the high-level functions and Appendix C for the low-level functions.

EVENTS

WHAT ARE EVENTS?

Events are occurrences of specific signals related to a processor's function. Hardware performance counters exist as a small set of registers that count events, such as cache misses and floating point operations while the program executes on the processor. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. Each processor has a number of events that are *native* to and often to that architecture. PAPI provides a software abstraction of these architecture-dependent native events into a collection of *preset* events that are accessible through the PAPI interface.

NATIVE EVENTS

WHAT ARE NATIVE EVENTS?

Native events comprise the set of all events that are countable by the CPU. In many cases, these events will be available through a matching preset PAPI event. Even if no preset event is available native events can still be accessed directly. These events are intended to be used by people who are very familiar with the particular platform in use. PAPI provides access to native events on all supported platforms through the low-level interface. Native events use the same interface as used when setting up a preset event, but a CPU-specific bit pattern is used instead of the PAPI event definition.

Native encoding is usually:

((register code & 0xffffff) << 8 | (register number & 0xff))

Native encodings are platform dependent, so the above native encoding may or may not work with your platform. To determine the native encoding for your platform, see Appendix F or the README file for your platform in the PAPI source distribution. In addition, the native event lists for the various platforms can be found in the processor architecture manual.

Native events are specified as arguments to the low-level function, PAPI_add_event. In the following code example, a native event is added by using PAPI_add_event with the register code = 0x800000 and the register number = 0x01:

```
#include <papi.h>
#include<stdio.h>

main()
{
     int retval, EventSet = PAPI_NULL;
     unsigned int native = 0x0;

     /* Initialize the library */
     retval = PAPI_library_init(PAPI_VER_CURRENT);

     if  (retval != PAPI_VER_CURRENT) {
       printf("PAPI library init error!\n");
       exit(1);
     }

     if  (PAPI_create_eventset(&EventSet) != PAPI_OK)
        handle_error(1);

     /* Add the native event */
     native = ((0x800000  & 0xffffff) << 8 | (0x01 & 0xff));

     if (PAPI_add_event(&EventSet, native) != PAPI_OK)
        handle_error(1);
}
```

For more code examples, see tests/native.c in the papi source distribution.

PRESET EVENTS

WHAT ARE PRESET EVENTS?

Preset events, also known as predefined events, are a common set of events deemed relevant and useful for application performance tuning. These events are typically found in many CPUs that provide performance counters and give access to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit, and pipeline status. Furthermore, preset events are mappings from symbolic names (PAPI preset name) to machine specific definitions (native countable events) for a particular hardware resource. For example, Total Cycles (in user mode) is PAPI_TOT_CYC. Also, PAPI supports presets that may be derived from the underlying hardware metrics. For example, Floating Point Instructions per Second is PAPI_FLOPS. A preset can be either directly available as a single counter, derived using a combination of counters, or unavailable on any particular platform.

The PAPI library names approximately 100 preset events, which are defined in the header file, **papiStdEventDefs.h**. For a given platform, a subset of these preset events can be counted though either

a simple high-level programming interface or a more complete C or Fortran low-level interface. For a list and a job description of all the preset events, see Appendix A.

The exact semantics of an event counter are platform dependent. PAPI preset names are mapped onto available events in a way, so it can count as many similar types of events as possible on different platforms. Due to hardware implementation differences, it is not necessarily feasible to directly compare the counts of a particular PAPI event obtained on different hardware platforms. To determine which preset events are available on a specific platform, see Appendix E or run tests/avail.c in the papi source distribution.

PRESET QUERY

The following low-level functions can be called to query about the existence of a preset (in other words, if the hardware supports that certain preset), to query details about a PAPI event, or to acquire details about all PAPI events, respectively:

C:

PAPI_query_event(*EventCode*)
PAPI_query_event_verbose(*EventCode, info*)
PAPI_query_all_events_verbose()

Fortran:

PAPIF_query_event(*EventCode, check*)
PAPIF_query_event_verbose(*EventCode*, *EventName, EventDescr*, *EventLabel*, *avail*, *EventNote, flags*, *check*)

ARGUMENTS

EventCode -- a defined event, such as PAPI_TOT_INS.

EventName -- the event name, such as the preset name, PAPI_BR_CN.

EventDescr -- a descriptive string for the event of length less than **PAPI_MAX_STR_LEN**.

EventLabel -- a short descriptive label for the event of length less than 18 characters.

avail -- zero if the event CANNOT be counted.

EventNote -- additional text information about an event (if available).

flags -- provides additional information about an event, e.g., **PAPI_DERIVED** for an event derived from 2 or more other events.

Note that PAPI_query_all_events_verbose is not implemented in Fortran because it returns a C pointer to an array of C structures.

PAPI_query_event asks the PAPI library if the PAPI Preset event can be counted on this architecture. If the event CAN be counted, the function returns PAPI_OK. If the event CANNOT be counted, the function returns an error code. On some platforms, this function also can be used to check the syntax of a native event.
PAPI_query_event_verbose asks the PAPI library for a copy of an event descriptor. This descriptor can then be used to investigate the details about the event. In Fortran, the individual fields in the descriptor are returned as parameters.

PAPI_query_all_events_verbose asks the PAPI library to return a pointer to an array of event descriptors. The number of objects in the array is **PAPI_MAX_PRESET_EVENTS** and each object is a descriptor as returned by **PAPI_query_event_verbose().**

```
#include <papi.h>
#include <stdio.h>

main()
{
      int EventSet = PAPI_NULL;
      unsigned int native = 0x0;
      int retval, i;
      PAPI_preset_info_t info;
      PAPI_preset_info_t *infostructs;

      /* Initialize the library */
      retval = PAPI_library_init(PAPI_VER_CURRENT);

      if (retval != PAPI_VER_CURRENT) {
      fprintf(stderr,"PAPI library init error!\n");
      exit(1); }

      /* Check to see if the preset, PAPI_TOT_INS, exists */
      if (PAPI_query_event (PAPI_TOT_INS) != PAPI_OK) {
      fprintf (stderr,"No instruction counter? How lame.\n");
      exit(1);
      }

      /* Query details about the preset, PAPI_TOT_INS */
      if (PAPI_query_event_verbose(PAPI_TOT_INS,&info) != PAPI_OK) {
      fprintf (stderr,"No instruction counter? How lame.\n");
      exit(1);
      }

      if (info.avail)
      printf ("This event is available on this hardware.\n");

      if (info.flags & PAPI_DERIVED)
      printf ("This event is a derived event on this hardware.\n");

      retval = 0;

      /* Acquire details of all PAPI events */
      infostructs = PAPI_query_all_events_verbose();
      if (infostructs)
      for (i = 0; i < PAPI_MAX_PRESET_EVENTS; i++)
      if (infostructs[i].avail)
      retval += 1;
}
```

OUTPUT (IF THE EVENT, PAPI_TOT_INS, IS AVAILABLE ON YOUR SYSTEM):

```
This event is available on this hardware.
```

In the above code example, PAPI_query_event is used to see if a preset ("PAPI_TOT_INS") exists, PAPI_query_event_verbose is used to query details about the event, and PAPI_query_all_events_verbose is used to acquire details about all PAPI events:

On success, PAPI_query_event and PAPI_query_event_verbose return PAPI_OK, and on error, a non-zero error code is returned.
On success, PAPI_query_all_events_verbose returns a pointer to an array of PAPI_preset_info_t structures and on error, a null pointer is returned.

For more information about the preset query functions, see Appendix C.

PRESET TRANSLATION

A preset event can be translated to a description, label, number, and string by calling the following low-level functions, respectively:

C:

PAPI_describe_event(*EventName, EventCode, EventDescr*)
PAPI_label_event(*EventCode, EventLabel*)
PAPI_event_name_to_code(*EventName, EventCode*)
PAPI_event_code_to_name(*EventCode, EventName*)

Fortran:

PAPIF_describe_event(*EventName, EventCode, EventDesc, check*)
PAPIF_label_event(*EventCode, EventLabel, check*)
PAPIF_event_name_to_code(*EventName, EventCode, check*)
PAPIF_event_code_to_name(*EventCode, EventName, check*)

ARGUMENTS

EventCode -- a defined event of integer type, such as PAPI_TOT_INS.

EventName -- the event name, such as the preset name, PAPI_BR_CN.

EventDescr -- a descriptive string for the event of length less than PAPI_MAX_STR_LEN.

EventLabel -- a short descriptive label for the event of length less than 18 characters.

Note that the preset does not actually have to exist to call these functions.

PAPI_describe_event is used to translate either an ASCII PAPI preset name or an integer PAPI preset event code into the corresponding event code or name as well as an ASCII description of that event. If the *EventName* argument is a string of length > 0 it is assumed to contain the name to look up and the corresponding event code is returned in the argument, *EventCode*. Otherwise, the *EventCode* argument is used to look up the event name, which is stored in the *EventName* argument. Finally, a descriptive string of length less than **PAPI_MAX_STR_LEN** is copied to the argument, *EventDescr*. **Note that the functionality of this call is a superset of the PAPI_event_name_to_code and PAPI_event_code_to_name calls.**

PAPI_label_event is used to translate an integer PAPI event code into a short (<=18 character) ASCII label that is more descriptive than the preset name but shorter than the description. These labels can be used as event identifiers in third party tools.

PAPI_event_name_to_code is used to translate an ASCII PAPI preset name into an integer PAPI event code.

PAPI_event_code_to_name is used to translate an integer PAPI event code into an ASCII PAPI preset name.

In the following code example, PAPI_event_name_to_code is used to translate a string into an integer (EventCode) and PAPI_label_event is used to label an event from an event code:

```
#include <papi.h>
#include <stdio.h>

main()
{
      int EventCode, retval;
      char EventCodeStr[PAPI_MAX_STR_LEN] = "PAPI_TOT_INS";
      char EventDescr[PAPI_MAX_STR_LEN];
      char EventLabel[20];

      /* Initialize the library */
      retval = PAPI_library_init(PAPI_VER_CURRENT);

      if  (retval != PAPI_VER_CURRENT) {
        fprintf(stderr, "PAPI library init error!\n");
        exit(1);
       }

      /* Translate the string to an integer code */
      if (PAPI_event_name_to_code(EventCodeStr, &EventCode) != PAPI_OK)
        handle_error(1);

      printf("Name: %s\nCode: %x\n", EventCodeStr, EventCode);

      /* Label event from EventCode */
      if (PAPI_label_event(EventCode, EventLabel) != PAPI_OK)
         handle_error(1);

      printf("Label: %s\n", EventLabel);

      /* Describe the event from EventCodeStr (The Event Name) */
       if (PAPI_describe_event(EventCodeStr,&EventCode,EventDescr) !=
      PAPI_OK)
          handle_error(1);

        printf("Description: %s\n", EventDescr);
}
```

OUTPUT:

```
Name: PAPI_TOT_INS
Code: 80000032
Label: Instr completed
Description: Instructions completed
```

Note that the event code is in hexadecimal, which is consistent with all the preset translation functions. The hexadecimal values of each preset are specified in the header file, papiStdEventDefs.h.

On success, all the functions return PAPI_OK and on error, a non-zero error code is returned.

For more information about the preset translation functions, see Appendix C.

PAPI'S COUNTER INTERFACES

HIGH-LEVEL API

WHAT IS A HIGH-LEVEL API?

The high-level API (**A**pplication **P**rogramming **I**nterface) simply provides the ability to start, stop, and read the counters for a specified list of events. It is meant for single thread applications and for programmers wanting simple and coarse-grained measurements. In addition, *it is not thread safe and allows only PAPI preset events.* Some of the benefits of using the high-level API rather than the low-level API are that it is easier to use and requires less setup (additional code).

It should also be noted that the high-level API could be used in conjunction with the low-level API and in fact does call the low-level API. However, the high-level API by itself is only able to access those events countable simultaneously by the underlying hardware.

There are six functions that represent the high-level API that allow the user to access and count specific hardware events. *Note that these functions can be implemented in both C and Fortran.* For a list and job description of all the high-level functions, see Appendix B. Also, for a code example of using the high-level interface, see Simple Code Examples: High Level API or tests/high-level.c in the PAPI source distribution.

INITIALIZATION OF A HIGH-LEVEL API

The PAPI library can be initialized implicitly by calling one of the following three high-level functions:

C:
PAPI_num_counters()
PAPI_start_counters(**events*, *array_length*)
PAPI_flops*(*real_time, *proc_time, *flpins, *mflops*)

Fortran:
PAPIF_num_counters(check)
PAPIF_start_counters*(*events*, *array_length*, check)
PAPIF_flops(*real_time, proc_time, flpins, mflops,* check)

ARGUMENTS
**events* -- an array of codes for events such as PAPI_INT_INS or a native event code.

array_length -- the number of items in the *events* array.

**real_time* -- the total real time since the first PAPI_flops call.

**proc_time* -- the total process time since the first PAPI_flops call.

**flpins* -- the total floating point instructions since the first PAPI_flops call.

**mflops* – Mflops/s achieved since the latest PAPI_flops call.

Note that one of the above functions must be called before calling any other PAPI function.

PAPI_num_counters returns the optimal length of the *values* array for high-level functions. This value corresponds to the number of hardware counters supported by the current substrate. PAPI_num_counters initializes the PAPI library using PAPI_library_init if necessary.

PAPI_start_counters initializes the PAPI library (if necessary) and starts 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 the *events* array should be no longer than the value returned by PAPI_num_counters.

The first call to PAPI_flops only initializes the library. For more information on PAPI_flops, see the following section:
Mflops/s, Real Time, and Processor Time.

In the following code example, PAPI_num_counters is used to initialize the library and to get the number of hardware counters available on the system. Also, PAPI_start_counters is used to start counting events:

```
#include <papi.h>

main()
{
     int Events[2] = { PAPI_TOT_CYC, PAPI_TOT_INS };
     int num_hwcntrs = 0;

     /* Initialize the PAPI library and get the number of
     counters available */
     if ((num_hwcntrs = PAPI_num_counters()) <= PAPI_OK)
         handle_error(1);

     printf("This system has %d available counters.",
        num_hwcntrs);

     if (num_hwcntrs > 2)
         num_hwcntrs = 2;

     /* Start counting events */
     if (PAPI_start_counters(Events, num_hwcntrs) != PAPI_OK)
         handle_error(1);
}
```

POSSIBLE OUTPUT (VARIES ON DIFFERENT SYSTEMS):

```
This system has 4 available counters.
```

On success, PAPI_num_counters returns the number of hardware counters available on the system and on error, a non-zero error code is returned.

***Optionally,* the PAPI library can be initialize explicitly by using PAPI_library_init.**

For more information on these functions, see Appendix B.

READING, ADDING, AND STOPPING COUNTERS

Counters can be read, added, and stopped by calling the following high-level functions, respectively:

C:
PAPI_read_counters(**values*, *array_length*)
PAPI_accum_counters(**values*, *array_length*)
PAPI_stop_counters(**values*, *array_length*)

Fortran:
PAPIF_read_counters(**values*, *array_length*, check)
PAPIF_accum_counters(**values*, *array_length*, check)
PAPIF_stop_counters(**values*, *array_length*, check)

ARGUMENTS

**values* -- an array where to put the counter values.

array_length -- the number of items in the *values array.

PAPI_read_counters and PAPI_accum_counters read (copy) and add the event counters into the array, *values*, respectively. The counters are reset and left running after the call of these functions.

PAPI_stop_counters stops the counters started by the function, PAPI_start_counters and return their values.

In the following code example, PAPI_read_counters and PAPI_stop_counters are used to copy and stop event counters in an array, respectively:

```
#include <papi.h>

#define NUM_EVENTS 2

main()
{
     int Events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_TOT_CYC};
     long_long values[NUM_EVENTS];

     /* Start counting events */
     if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
        handle_error(1);

     /* Do some computation here*/

     /* Read the counters */
     if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);

     /* Do some computation here */

     /* Stop counting events */
     if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
}
```

On success, all of these functions return PAPI_OK and on error, a non-zero error code is returned.

For more information on these functions, see Appendix B.

MFLOPS/S, REAL TIME, AND PROCESSOR TIME

Mflops/s, real time, and processor time can be obtained by calling the following high-level function:

C:
PAPI_flops(**real_time*, **proc_time*, **flpins*, **mflops*)

Fortran:
PAPIF_flops(*real_time*, *proc_time*, *flpins*, *mflops*, check)

ARGUMENTS

**real_time* -- the total real time since the first PAPI_flops call.

**proc_time* -- the total process time since the first PAPI_flops call.

**flpins* -- the total floating point instructions since the first PAPI_flops call.

**mflops* – Mflops/s achieved since the latest PAPI_flops call.

The first call to PAPI_flops initializes the PAPI library, set up the counters to monitor PAPI_FP_INS and PAPI_TOT_CYC events, and start the counters. Subsequent calls will read the counters and return total real time, total process time, total floating point instructions, and the Mflops/s rate since the last call to PAPI_flops. Any call with flpins = -1 will reinitialize all counters to 0.

Note that most platforms are only capable of counting the number of floating point instructions completed. This may or may not translate to your definition of floating point operations. The measured rate is thus Mflops/s, and will in some circumstances count FMA instructions as one operation. Consult the hardware documentation for your system for more details.

PAPI_flops may be called by the user's application program and contains calls to the following functions:
PAPI_perror, PAPI_library_init, PAPI_get_hardware_info, PAPI_create_eventset, PAPI_add_event, PAPI_start, PAPI_get_real_usec, PAPI_accum, and PAPI_shutdown.

On success, it returns PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix B. Also, for a code example, see test/flops.c in the papi source distribution.

LOW-LEVEL API

WHAT IS A LOW-LEVEL API?

The low-level API (**A**pplication **P**rogramming **I**nterface) manages hardware events in user-defined groups called Event Sets. It is meant for experienced application programmers and tool developers wanting more fine-grained measurements. *Unlike the high-level interface, it is thread safe and allows both PAPI preset and native events.* Another features of the low-level API are the ability to obtain information about the executable and the hardware as well as to set options for multiplexing and overflow handling. Some of the benefits of using the low-level API rather than the high-level API are that it increases efficiency and functionality.

It should also be noted that the low-level interface could be used in conjunction with the high-level interface, but the user would have to be careful about initialization and threads.

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 on every platform and defined in the header file. Therefore, the user is encouraged to read the documentation for each platform carefully. *There are approximately 40 functions that represent the low-level API, where some of these functions are implemented only in C or Fortran.* For more information on these function and their job descriptions, see Appendix C. Also, for a code example of using the low-level interface, see Simple Code Examples: Low-Level API or tests/low_level.c in the PAPI source distribution.

INITIALIZATION OF A LOW-LEVEL API

The PAPI library can be initialized explicitly by calling the following low-level function:

C:
PAPI_library_init(*version*)

Fortran:
PAPIF_library_init(check)

ARGUMENT

version -- upon initialization, PAPI checks the argument against the internal value of **PAPI_VER_CURRENT** when the library was compiled. This guards against portability problems when updating the PAPI shared libraries on your system.

Note that this function must be called before calling any other PAPI function.

The following is a code example of using PAPI_library_init to initialize the PAPI library:

```
#include <papi.h>
#include <stdio.h>
int retval;

main()
{
     /* Initialize the PAPI library */
     retval = PAPI_library_init(PAPI_VER_CURRENT);

     if (retval != PAPI_VER_CURRENT && retval > 0) {
       fprintf(stderr,"PAPI library version mismatch!\n");
       exit(1); }

     if (retval < 0) {
       fprintf(stderr, "Initialization error!\n");
       exit(1); }
}
```

On success, this function returns PAPI_VER_CURRENT.
On error, a *positive* return code other than PAPI_VER_CURRENT indicates a library version mismatch and a *negative* return code indicates an initialization error.

For more information on this function, see Appendix C.

EVENT SETS

WHAT ARE EVENT SETS?

Event Sets are user-defined groups of hardware events (preset or native), which are used in conjunction with one another to provide meaningful information, such as: what low-level hardware counters to use, the most recently read counter values, the state of the Event Set (running/not running), and optional settings (e.g., overflow, profiling). Therefore, Event Sets allow a highly efficient implementation and as a result, users can have more detailed and accurate measurements. In addition, Event Sets are managed by the user through the use of integer handles, which helps simplify inter-language calling conventions. There are no real programming restrictions on the use of Event Sets. The user is free to allocate and use any number of them provided the substrate can provide the required resources. They may be used simultaneously and in fact may even share counter values.

CREATING AN EVENT SET

An event set can be created by calling the following the low-level function:

C:
PAPI_create_eventset (**EventSet*)

Fortran:
PAPIF_create_eventset(*EventSet*, check)

ARGUMENT

EventSet -- Address of an integer location to store the new EventSet handle.

Note that *EventSet* must be initialized to PAPI_NULL before calling this function. Then, the user may add hardware events to the *EventSet* by calling PAPI_add_event or similar functions.

On success, this function returns PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix C. Also, for a code example, see the next section.

ADDING EVENTS TO AN EVENT SET

Hardware events can be added to an event set by calling the following the low-level functions:

C:
PAPI_add_event(**EventSet*, *EventCode*)
PAPI_add_events(**EventSet*, **EventCode*, *number*)

Fortran:
PAPIF_add_event(*EventSet*, *EventCode*, check)
PAPIF_add_events(*EventSet*, *EventCode*, *number*, check)

ARGUMENTS

**EventSet* -- an integer handle for a PAPI Event Set as created by PAPI_create_eventset.

EventCode -- a defined event such as PAPI_TOT_INS.

**EventCode* -- an array of defined events.

number -- an integer indicating the number of events in the array **EventCode*.

PAPI_add_event adds a single hardware event to a PAPI event set.

PAPI_add_events does the same as PAPI_add_event, but for an array of hardware event codes.

In the following code example, the preset event, PAPI_TOT_INS is added to an event set:

```
#include <papi.h>
#include <stdio.h>

main()
{
     int EventSet = PAPI_NULL;
     int retval;

     /* Initialize the PAPI library */
     retval = PAPI_library_init(PAPI_VER_CURRENT);

     if (retval != PAPI_VER_CURRENT) {
       fprintf(stderr, "PAPI library init error!\n");
       exit(1); }

     /* Create an EventSet */
     if (PAPI_create_eventset(&EventSet) != PAPI_OK)
       handle_error(1);

     /* Add Total Instructions Executed to our EventSet */
     if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
       handle_error(1);
}
```

On success, both of these functions return PAPI_OK and on error, a non-zero error code is returned.

For more information on these functions, see Appendix C.

STARTING, READING, ADDING, AND STOPPING EVENTS IN AN EVENT SET

Hardware events in an event set can be started, read, added, and stopped by calling the following low-level functions, respectively:

C:
PAPI_start(*EventSet*)
PAPI_read(*EventSet, *values*)
PAPI_accum(*EventSet, *values*)
PAPI_stop(*EventSet, *values*)

Fortran:
PAPIF_start(*EventSet,* check)
PAPIF_read(*EventSet, values,* check)
PAPIF_accum(*EventSet, values*, check)
PAPIF_stop(*EventSet, values,* check)

ARGUMENTS

EventSet -- an integer handle for a PAPI Event Set as created by PAPI_create_eventset.

**values* -- an array to hold the counter values of the counting events.

PAPI_start starts the counting events in a previously defined event set.

PAPI_read reads (copies) the counters of the indicated event set into the array, *values*. The counters are left counting after the read without resetting.

PAPI_accum adds the counters of the indicated event set into the array, *values*. The counters are reset and left counting after the call of this function.

PAPI_stop stops the counting events in a previously defined event set and return the current events. The following is a code example of using PAPI_start to start the counting of events in an event set, PAPI_read to read the counters of the same event set into the array *values*, and PAPI_stop to stop the counting of events in the event set:

```
#include <papi.h>
#include <stdio.h>

main()
{
     int retval, EventSet = PAPI_NULL;
     long_long values[1];

     /* Initialize the PAPI library */
     retval = PAPI_library_init(PAPI_VER_CURRENT);

     if (retval != PAPI_VER_CURRENT) {
       fprintf(stderr, "PAPI library init error!\n");
       exit(1); }

     /* Create the Event Set */
     if (PAPI_create_eventset(&EventSet) != PAPI_OK)
         handle_error(1);

     /* Add Total Instructions Executed to our EventSet */
     if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
         handle_error(1);

     /* Start counting */
     if (PAPI_start(EventSet) != PAPI_OK)
         handle_error(1);

     /* Do some computation here */

     if (PAPI_read(EventSet, values) != PAPI_OK)
         handle_error(1);

     /* Do some computation here */

     if (PAPI_stop(EventSet, values) != PAPI_OK)
         handle_error(1);
}
```

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

For more information on these functions, see Appendix C.

RESETTING EVENTS IN AN EVENT SET

The hardware event counts in an event set can be reset to zero by calling the following low-level function:

C:
PAPI_reset(*EventSet*)

Fortran:
PAPI_reset(*EventSet*, check)

ARGUMENT

EventSet -- an integer handle for a PAPI event set as created by PAPI_create_eventset.

Note that the event set must be running or stopped in order to call PAPI_reset.

For example, the *EventSet* in the code example of the previous section could have been reset to zero by adding the following lines:

```
if (PAPI_reset(EventSet) != PAPI_OK)
  handle_error(1);
```

On success, this function returns PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix C.

REMOVING EVENTS IN AN EVENT SET

A hardware event and an array of hardware events can be removed from an event set by calling the following low-level functions, respectively:

C:
PAPI_rem_event(*EventSet*, *EventCode*)
PAPI_rem_events(*EventSet*, *EventCode*, number)

Fortran:
PAPIF_rem_event(*EventSet*, *EventCode*, check)
PAPIF_rem_events(*EventSet*, *EventCode*, *number*, check)

ARGUMENTS

EventSet -- an integer handle for a PAPI event set as created by PAPI_create_eventset.

EventCode -- a defined event such as PAPI_TOT_INS or a native event.

**EventCode* -- an array of defined events.

number -- an integer indicating the number of events in the array **EventCode*.

PAPI_rem_event removes a single hardware event from a PAPI event set.

PAPI_rem_events, does the same as PAPI_rem_event, but for an array of hardware event codes.

```
#include <papi.h>
#include <stdio.h>
main()
{
     int retval, EventSet = PAPI_NULL;

     /* Initialize the PAPI library */
     retval = PAPI_library_init(PAPI_VER_CURRENT);

     if (retval != PAPI_VER_CURRENT) {
       fprintf(stderr, "PAPI library init error!\n");
       exit(1); }

     /* Create an EventSet */
     if (PAPI_create_eventset(&EventSet) != PAPI_OK)
       handle_error(1);

     /* Add Total Instructions Executed to our EventSet */
     if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
       handle_error(1);

     /* Remove event */
     if (PAPI_rem_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
       handle_error(1);
}
```

In the following code example, PAPI_rem_event is used to removed the event, PAPI_TOT_INS, from an event set:

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

For more information on these functions, see Appendix C.

EMPTYING AND DESTROYING AN EVENT SET

All the events in an event set can be emptied and destroyed by calling the following low-level functions, respectively:

C:
PAPI_cleanup_eventset(*EventSet*)
PAPI_destroy_eventset(*EventSet*)

Fortran:
PAPIF_cleanup_eventset(*EventSet*, check)
PAPIF_destroy_eventset(*EventSet*, check)

ARGUMENT

EventSet -- an integer handle for a PAPI event set as created by PAPI_create_eventset.

Note that the event set must be empty in order to use PAPI_destroy_eventset.

In the following code example, PAPI_cleanup_eventset is used to empty all the events from an event set and PAPI_remove_eventset is used to deallocate the memory associated with the empty event set:

```
#include <papi.h>
#include <stdio.h>

main()
{
     int retval, EventSet = PAPI_NULL;

     /* Initialize the PAPI library */
     retval = PAPI_library_init(PAPI_VER_CURRENT);

     if (retval != PAPI_VER_CURRENT) {
       fprintf(stderr, "PAPI library init error!\n");
       exit(1); }

     /* Create the EventSet */
     if (PAPI_create_eventset(&EventSet) != PAPI_OK)
         handle_error(1);

     /* Add Total Instructions Executed to our EventSet */
     if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
         handle_error(1);

     /* Remove all events in the eventset */
     if (PAPI_cleanup_eventset(&EventSet) != PAPI_OK)
       handle_error(1);

     /* Free all memory and data structures, EventSet must be
     empty. */
     if (PAPI_destroy_eventset(&EventSet) != PAPI_OK)
       handle_error(1);
}
```

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

For more information on these functions, see Appendix C.

THE STATE OF AN EVENT SET

The counting state of an Event Set can be obtained by calling the following low-level function:

C:
PAPI_state(*EventSet*, **status*)

Fortran:
PAPIF_state(*EventSet*, *status*, check)

ARGUMENTS

EventSet -- an integer handle for a PAPI event set as created by PAPI_create_eventset.

status -- an integer containing a Boolean combination of one or more of the following nonzero constants as defined in the PAPI header file, papi.h:

PAPI_STOPPED	*EventSet* is stopped
PAPI_RUNNING	*EventSet* is running
PAPI_PAUSED	*EventSet* temporarily disabled by the library
PAPI_NOT_INIT	*EventSet* defined, but not initialized
PAPI_OVERFLOWING	*EventSet* has overflow enabled
PAPI_PROFILING	*EventSet* has profiling enabled
PAPI_MULTIPLEXING	*EventSet* has multiplexing enabled
PAPI_ACCUMULATING	*EventSet* has accumulating enabled

In the following code example, PAPI_state is used to return the counting state of an *EventSet*:

```
#include <papi.h>
#include <stdio.h>

main ()
{
     int retval, status = 0, EventSet = PAPI_NULL;

     /* Initialize the PAPI library */
     retval = PAPI_library_init(PAPI_VER_CURRENT);

     if (retval != PAPI_VER_CURRENT) {
       fprintf(stderr, "PAPI library init error!\n");
       exit(1); }

     /* Create the EventSet */
     if (PAPI_create_eventset(&EventSet) != PAPI_OK)
       handle_error(1);

     /* Add Total Instructions Executed to our EventSet */
     if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
       handle_error(1);

     /* Start counting */
     if (PAPI_state(EventSet, &status) != PAPI_OK)
       handle_error(1);

     printf("State is now %d\n", status);

     if (PAPI_start(EventSet) != PAPI_OK)
       handle_error(1);

     if (PAPI_state(EventSet, &status) != PAPI_OK)
       handle_error(1);

     printf("State is now %d\n", status);
}
```

OUTPUT:

```
State is now 1
State is now 2
```

On success, this function returns PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix C.

GETTING AND SETTING OPTIONS

The options of the PAPI library or a specific event set can be obtained and set by calling the following low-level functions, respectively:

C:
PAPI_get_opt(*option, ptr*)
PAPI_set_opt(*option, ptr*)

Fortran:
PAPIF_get_clockrate(*clockrate*)
PAPIF_get_domain(*EventSet, domain, mode,* check)
PAPIF_get_granularity(*EventSet,* granularity, *mode,* check)
PAPIF_get_preload(*preload,* check)

ARGUMENTS

option -- is an input parameter describing the course of action. The Fortran calls are implementations of specific options. Possible values are defined in **papi.h** and briefly described below:

Predefined name	**Explanation**
General information requests	
PAPI_GET_CLOCKRATE	Return clockrate in MHz.
PAPI_GET_MAX_CPUS	Return number of CPUs.
PAPI_GET_MAX_HWCTRS	Return number of counters.
PAPI_GET_EXEINFO	Addresses for text/data/bss.
PAPI_GET_HWINFO	Info. about hardware.
PAPI_GET_PRELOAD	Get ''LD_PRELOAD'' environment equivalent.
Defaults for the global library	
PAPI_GET_DEFDOM	Return the default counting domain for newly created event sets.
PAPI_SET_DEFDOM	Set the default counting domain.
PAPI_GET_DEFGRN	Return the default counting granularity.
PAPI_SET_DEFGRN	Set the default counting granularity.
PAPI_GET_DEBUG	Get the PAPI debug state. The available debug states are defined in papi.h. The debug state is available in ptr->debug
PAPI_SET_DEBUG	Set the PAPI debug state
Multiplexing control	
PAPI_GET_MULTIPLEX	Get options for multiplexing. Currently not implemented.
PAPI_SET_MULTIPLEX	Set options for multiplexing
Manipulating individual event sets	
PAPI_GET_DOMAIN	Get domain for a single event set. The event set is specified in ptr-

	>domain.eventset
PAPI_SET_DOMAIN	Set the domain for a single event set.
PAPI_GET_GRANUL	Get granularity for a single event set. The event set is specified in ptr->granularity.eventset
PAPI_SET_GRANUL	Set the granularity for a single event set.

ptr -- is a pointer to a structure that acts as both an input and output parameter. It is defined in **papi.h** and below.

EventSet -- input; a reference to an EventSetInfo structure

clockrate -- output; cycle time of this CPU in MHz; *may* be an estimate generated at init time with a quick timing routine

domain -- output; execution domain for which events are counted

granularity -- output; execution granularity for which events are counted

mode -- input; determines if domain or granularity are default or for the current event set

preload -- output; environment variable string for preloading libraries

PAPI_get_opt and PAPI_set_opt query or change the options of the PAPI library or a specific event set created by PAPI_create_eventset. In C interface, these functions pass a pointer to the *PAPI_option_t* structure. Not all options require or return information in this structure. The Fortran interface is a series of calls implementing various subsets of the C interface. **Not all options in C are available in Fortran.** Note that some options, such as PAPI_SET_DOMAIN, are also available as separate entry points in both C and Fortran.

The file, **papi.h**, contains definitions for the structures combined in the *PAPI_option_t* structure. Users should use the definitions in **papi.h** that correspond with the library used.

```
#include <papi.h>
#include <stdio.h>

main()
{
int num, retval, EventSet = PAPI_NULL;
PAPI_option_t options;

/* Initialize the PAPI library */
retval = PAPI_library_init(PAPI_VER_CURRENT);

if (retval != PAPI_VER_CURRENT) {
  fprintf(stderr, "PAPI library init error!\n");
  exit(1); }

if ((num = PAPI_get_opt(PAPI_GET_MAX_HWCTRS,NULL)) <= 0)
  handle_error();

printf("This machine has %d counters.0,num);

if (PAPI_create_eventset(&EventSet) != PAPI_OK)
  handle_error();

/* Set the domain of this EventSet
   to counter user and kernel modes for this
   process */

memset(&options,0x0,sizeof(options));

options.domain.eventset = EventSet;
options.domain.domain = PAPI_DOM_ALL;
if (PAPI_set_opt(PAPI_SET_DOMAIN, &options) != PAPI_OK)
  handle_error();
}
```

In the following code example, PAPI_get_opt is used to acquire the option, PAPI_GET_MAX_HWCTRS, of an event set and PAPI_set_opt is used to set the option, PAPI_SET_DOMAIN, to the same event set:

POSSIBLE OUTPUT (VARIES ON DIFFERENT PLATFORMS):

```
This machine has 4 counters.
```

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

For more information on these functions, see Appendix C and for more code examples, see tests/second.c or tests/third.c in the PAPI source distribution.

SIMPLE CODE EXAMPLES

HIGH-LEVEL API

The following is a simple code example of using the high-level API:

```
#include <papi.h>

#define NUM_FLOPS 10000
#define NUM_EVENTS 1

main()
{
     int Events[NUM_EVENTS] = {PAPI_TOT_INS};
     long_long values[NUM_EVENTS];

     /* Start counting events */
     if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
        handle_error(1);

     /* Defined in tests/do_loops.c in the PAPI source distribution
     */
     do_flops(NUM_FLOPS);

     /* Read the counters */
     if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);

     printf("After reading the counters: %lld\n",values[0]);

     do_flops(NUM_FLOPS);

     /* Add the counters */
     if (PAPI_accum_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
     printf("After adding the counters: %lld\n", values[0]);

     do_flops(NUM_FLOPS);

     /* Stop counting events */
     if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);

     printf("After stopping the counters: %lld\n", values[0]);
}
```

POSSIBLE OUTPUT:

```
After reading the counters: 441027
After adding the counters: 891959
After stopping the counters: 443994
```

Notice that on the second line (after adding the counters) the value is approximately twice as large as the first line (after reading the counters) because PAPI_read_counters resets and leaves the counters running, then PAPI_accum_counters adds the value of the current counter into the *values* array.

LOW-LEVEL API

The following is a simple code example that does the same technique as the above example, except it uses the Low-Level API:

```
#include <papi.h>
#include <stdio.h>

#define NUM_FLOPS 10000

main()
{
      int retval, EventSet=PAPI_NULL;
      long_long values[1];

      /* Initialize the PAPI library */
      retval = PAPI_library_init(PAPI_VER_CURRENT);
      if (retval != PAPI_VER_CURRENT) {
        fprintf(stderr, "PAPI library init error!\n");
        exit(1); }

      /* Create the Event Set */
      if (PAPI_create_eventset(&EventSet) != PAPI_OK)
          handle_error(1);

      /* Add Total Instructions Executed to our Event Set */
      if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
          handle_error(1);

      /* Start counting events in the Event Set */
      if (PAPI_start(EventSet) != PAPI_OK)
          handle_error(1);

      /* Defined in tests/do_loops.c in the PAPI source distribution */
      do_flops(NUM_FLOPS);

      /* Read the counting events in the Event Set */
      if (PAPI_read(EventSet, values) != PAPI_OK)
          handle_error(1);

      printf("After reading the counters: %lld\n",values[0]);

      /* Reset the counting events in the Event Set */
      if (PAPI_reset(EventSet) != PAPI_OK)
        handle_error(1);

      do_flops(NUM_FLOPS);

      /* Add the counters in the Event Set */
      if (PAPI_accum(EventSet, values) != PAPI_OK)
         handle_error(1);
      printf("After adding the counters: %lld\n",values[0]);

      do_flops(NUM_FLOPS);

      /* Stop the counting of events in the Event Set */
      if (PAPI_stop(EventSet, values) != PAPI_OK)
          handle_error(1);

      printf("After stopping the counters: %lld\n",values[0]);
}
```

POSSIBLE OUTPUT:

```
After reading the counters: 440973
After adding the counters: 882256
After stopping the counters: 443913
```

Notice that in order to get the desired results (the second line approximately twice as large as the first line), PAPI_reset was called to reset the counters, since PAPI_read did not reset the counters.

PAPI TIMERS

PAPI timers use the most accurate timers available on the platform in use. These timers can be use to obtain both real and virtual time on each supported platform. The real time clock runs all the time (e.g. a wall clock) and the virtual time clock runs only when the processor is running in user mode.

REAL TIME

Real time can be acquired in clock cycles and microseconds by calling the following low-level functions, respectively:

C:
PAPI_get_real_cyc()
PAPI_get_real_usec()

Fortran:
PAPIF_get_real_cyc(check)
PAPIF_get_real_usec(check)

Both of these functions return the total real time passed since some arbitrary starting point and are equivalent to wall clock time. Also, *these functions always succeed (error-free) since they are guaranteed to exist on every PAPI supported platform.*

In the following code example, PAPI_get_real_cyc() and PAPI_get_real_usec() are used to obtain the real time it takes to create an event set in clock cycles and microseconds, respectively:

```
#include <papi.h>

 main()
{
     long_long start_cycles, end_cycles, start_usec, end_usec;
     int EventSet = PAPI_NULL;

     if (PAPI_library_init(PAPI_VER_CURRENT) !=
     PAPI_VER_CURRENT)
       exit(1);

     /* Gets the starting time in clock cycles */
     start_cycles = PAPI_get_real_cyc();

     /* Gets the starting time in microseconds */
     start_usec = PAPI_get_real_usec();

     /*Create an EventSet */
     if (PAPI_create_eventset(&EventSet) != PAPI_OK)
       exit(1);

     /* Gets the ending time in clock cycles */
     end_cycles = PAPI_get_real_cyc();

     /* Gets the ending time in microseconds */
     end_usec = PAPI_get_real_usec();

     printf("Wall clock cycles: %lld\n", end_cycles -
     start_cycles);
     prinf("Wall clock time in microseconds: %lld\n", end_usec
     - start_usec);
}
```

POSSIBLE OUTPUT:

```
Wall clock cycles: 100173
Wall clock time in microseconds: 136
```

For more information on these functions, see Appendix C.

VIRTUAL TIME

Virtual time can be acquired in clock cycles and microseconds by calling the following low-level functions, respectively:

C:
PAPI_get_virt_cyc()
PAPI_get_virt_usec()

Fortran:
PAPIF_get_virt_cyc(check)
PAPIF_get_virt_usec(check)

Both of these functions return the total number of virtual units from some arbitrary starting point. Virtual units accrue every time a process is running in user-mode. *Like the real time counters, these functions always succeed (error-free) since they are guaranteed to exist on every PAPI supported platform.* However, the resolution can be as bad as 1/Hz as defined by the operating system on some platforms.

In the following code example, PAPI_get_virt_cyc() and PAPI_get_virt_usec() are used to obtain the virtual time it takes to create an event set in clock cycles and microseconds, respectively:

```
#include <papi.h>

 main()
{
     long_long start_cycles, end_cycles, start_usec, end_usec;
     int EventSet = PAPI_NULL;

     if (PAPI_library_init(PAPI_VER_CURRENT) !=
     PAPI_VER_CURRENT)
       exit(1);

     /* Gets the starting time in clock cycles */
     start_cycles = PAPI_get_virt_cyc();

     /* Gets the starting time in microseconds */
     start_usec = PAPI_get_virt_usec();

     /*Create an EventSet */
     if (PAPI_create_eventset(&EventSet) != PAPI_OK)
       exit(1);

     /* Gets the ending time in clock cycles */
     end_cycles = PAPI_get_virt_cyc();

     /* Gets the ending time in microseconds */
     end_usec = PAPI_get_virt_usec();

     printf("Virtual clock cycles: %lld\n", end_cycles -
     start_cycles);
     prinf("Virtual clock time in microseconds: %lld\n",
     end_usec - start_usec);
```

POSSIBLE OUTPUT:

```
Virtual clock cycles: 715408
Virtual clock time in microseconds: 976
```

For more information on these functions, see Appendix C.

PAPI SYSTEM INFORMATION

EXECUTABLE INFORMATION

Information about the executable's address space can be obtained by using the following low-level function:

C:
PAPI_get_executable_info()

Fortran:
PAPIF_get_exe_info(*fullname, name, text_start, text_end, data_start, data_end, bss_start, bss_end, lib_preload_env*, check)

ARGUMENTS
The following arguments are implicit in the structure returned by the C function, or explicitly returned by Fortran:

fullname -- fully qualified path + filename of the executable

name -- filename of the executable with no path information

text_start, text_end -- Start and End addresses of program text segment

data_start, data_end -- Start and End addresses of program data segment

bss_start, bss_end -- Start and End addresses of program bss segment

lib_preload_env -- environment variable for preloading libraries

Note that the arguments, *text_start* and *text_end*, are the only fields that are filled on every architecture.

In C, this function returns a pointer to a structure containing information about the current program, such as the start and end addresses of the text, data, and bss segments.
In Fortran, the fields of the structure are returned explicitly.

In the following code example, PAPI_get_executable_info() is used to acquire information about the start and end addresses of the program's text segment:

```
#include <papi.h>
#include <stdio.h>

main()
{
     const PAPI_exe_info_t *prginfo = NULL;

     if (PAPI_library_init(PAPI_VER_CURRENT) !=
     PAPI_VER_CURRENT)
       exit(1);

     if ((prginfo = PAPI_get_executable_info()) == NULL)
       exit(1);

     printf("Start of user program is at %p\n",prginfo-
     >text_start);
     printf("End of user program is at %p\n",prginfo-
     >text_end);
}
```

POSSIBLE OUTPUT:

```
Start of user program is at 0x4000000000000f20
End of user program is at 0x4000000000034e00
```

In C, on success, the function returns a non-NULL pointer and on error, NULL is returned.
In Fortran, on success, the function returns PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix C.

HARDWARE INFORMATION

Information about the system hardware can be obtained by using the following low-level function:

C:
PAPI_get_hardware_info()

Fortan:
PAPIF_get_hardware_info (nc*pu*, *nnodes*, *totalcpus*, *vendor*, *vendor_string*, *model*, *model_string*, *revision*, *mhz*)

ARGUMENTS
The following arguments are implicit in the structure returned by the C function, or explicitly returned by Fortran.

ncpu -- number of CPUs in an SMP Node

nnodes -- number of Nodes in the entire system

totalcpus -- total number of CPUs in the entire system

vendor -- vendor id number of CPU

vendor_string -- vendor id string of CPU

model -- model number of CPU

model_string -- model string of CPU

revision -- Revision number of CPU

mhz -- Cycle time of this CPU; *may* be an estimate generated at initial time with a quick timing routine

In C, this function returns a pointer to a structure containing information about the hardware on which the program runs, such as: the number of CPUs, CPU model information, and the cycle time of the CPU.
In Fortran, the values of the structure are returned explicitly.

Note that if this function were called before PAPI_library_init, it would be undefined.

In the following code example, PAPI_get_hardware_info is used to acquire hardware information about the total number of CPUs and the cycle time of the CPU:

```
#include <papi.h>
#include <stdio.h>

main()
{
     const PAPI_hw_info_t *hwinfo = NULL;

     if (PAPI_library_init(PAPI_VER_CURRENT) !=
     PAPI_VER_CURRENT)
       exit(1);

     if ((hwinfo = PAPI_get_hardware_info()) == NULL)
       exit(1);

     printf("%d CPU's at %f Mhz.\n",hwinfo->totalcpus,hwinfo-
     >mhz);
}
```

POSSIBLE OUTPUT:

```
1 CPUs at 733.000000 Mhz.
```

In C, on success, this function returns a non-NULL pointer and on error, NULL is returned.
In Fortran, on success, this function returns PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix C.

ADVANCED PAPI FEATURES

MULTIPLEXING

WHAT IS MULTIPLEXING?

Multiplexing allows more counters to be used than what is supported by the hardware, thus allowing a larger number of events to be counted simultaneously. When a microprocessor has a very limited number of events that can be counted simultaneously, a large application with many hours of run time may require days or weeks of profiling in order to gather enough information to base a performance analysis. Therefore, multiplexing overcomes this limitation by subdividing the usage of the counter hardware over time (timesharing).

USING PAPI WITH MULTIPLEXING

INITIALIZATION OF MULTIPLEX SUPPORT

Multiplex support in the PAPI library can be enabled and initialized by calling the following low-level function:

C:
PAPI_muliplex_init()

Fortran:
PAPIF_multiplex_init(check)

The above function allows more events to be counted than there are physical counters by timesharing the existing counters at some loss in precision. **This function should be used after calling PAPI_library_init.** After this function is called, the user can proceed to use the normal PAPI routines. *It should be also noted that applications that make no use of multiplexing should not call this function.*

On success, this function returns PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix C and for a code example, see the next section.

CONVERTING AN EVENT SET INTO A MULTIPLEXED EVENT SET

In addition, a standard event set can be converted to a multiplexed event set by the calling the following low-level function:

C:
PAPI_set_multiplex(*EventSet*)

Fortran:
PAPIF_set_multiplex(*EventSet*)

ARGUMENT
**EventSet* -- a pointer to an integer handle for a PAPI event set as created by PAPI_create_eventset.

The above function converts a standard PAPI event set created by a call to PAPI_create_eventset into an event set capable of handling multiplexed events. **This function must be used after calling PAPI_multiplex_init and PAPI_create_eventset, but prior to calling PAPI_start.** Events can be added to an event set either before or after converting it into a multiplexed set, but the conversion must be done prior to using it as a multiplexed set.

In the following code example, PAPI_set_multiplex is used to convert a standard event set into a multiplexed event set:

```
int retval, i, EventSet = PAPI_NULL, max_to_add = 6, j = 0;
long_long *values;
const PAPI_preset_info_t *pset;

main()
{
       /* Initialize the PAPI library */
       retval = PAPI_library_init(PAPI_VER_CURRENT);
       if (retval != PAPI_VER_CURRENT)
         handle_error(1);

       pset = PAPI_query_all_events_verbose();
       if (pset == NULL)
         handle_error(1);

       /* Enable and initialize multiplex support */
       if (PAPI_multiplex_init() != PAPI_OK)
         handle_error(1);

      /* Create an EventSet */
       if (PAPI_create_eventset(&EventSet) != PAPI_OK)
         handle_error(1);

      /* Convert the EventSet to a multiplexed event set */
       if (PAPI_set_multiplex(&EventSet) != PAPI_OK)
         handle_error(1);

       for (i=0;i<PAPI_MAX_PRESET_EVENTS;i++)
       {
         if ((pset->avail) && (pset->event_code != PAPI_TOT_CYC))
             {
                if (PAPI_add_event(&EventSet, pset->event_code) !=
     PAPI_OK)
                 handle_error(1);

                if (++j >= max_to_add)
                 break;
             }
             pset++;
       }

       values = (long_long *)malloc(max_to_add*sizeof(long_long));
       if (values == NULL)
         handle_error(1);

       /* Start counting events */
       if (PAPI_start(EventSet) != PAPI_OK)
         handle_error(1);
}
```

On success, both functions return PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix C. Also, for more code examples, see tests/multiplex1.c in the papi source distribution.

ISSUES OF MULTIPLEXING

The following are some issues concerning multiplexing that the PAPI user should be aware of:

- **Multiplexing is not supported by all platforms** and therefore, PAPI implements software multiplexing on those platforms that do not support multiplexing through the use of a high-resolution interval timer. For more information on which platforms support hardware or software multiplexing, see Appendix H.

- Multiplexing unavoidably incurs a small amount of overhead and can adversely affect the accuracy of reported counter values. In other words, the more events that are multiplexed, the more likely that the results will be incorrect. The granularity of the measured regions must be increased in order to get acceptable results.

- To prevent naïve use of multiplexing by the novice user, the high level API can only access those events countable simultaneously by the underlying hardware, unless a low level function has been called to explicitly enable multiplexing.

USING PAPI WITH PARALLEL PROGRAMS

THREADS

WHAT ARE THREADS?

A thread is an independent flow of instructions that can be scheduled to run by the operating system. Multi-threaded programming is a form of parallel programming where several controlled threads are executing concurrently in the program. All threads execute in the same memory space, and can therefore work concurrently on shared data. Threads can run parallel on several processors, allowing a single program to divide its work between several processors, thus running faster than a single-threaded program, which runs on only one processor at a time.

PAPI only supports thread level measurements with *kernel or bound threads*, which are threads that have a scheduling entity known and handled by the operating system's kernel. In most cases like with SMP or OpenMP complier directives, *bound* threads will be the default. Each thread is responsible for the creation, start, stop, and read of its own counters. When a thread is created, it inherits no PAPI information from the calling thread. There are some threading packages or APIs that can be used to manipulate threads with PAPI, particularly Pthreads and OpenMP. For those using Pthreads, the user should take care to set the scope of each thread to PTHREAD_SCOPE_SYSTEM attribute, unless the system is known to have a non-hybrid thread library implementation.

In addition, PAPI does support *unbound or non-kernel threads*, but the counts will reflect the total events for the process. Measurements that are done in other threads will get all the same values, namely the counts for the total process. For *unbound* threads, it is not necessary to call PAPI_thread_init, which will be discussed in the next section.

When threads are in use, PAPI allows the user to provide a routine to its library that returns the thread ID of the currently running thread (for example, pthreads_self for Pthreads) and this thread ID is used as a lookup function for the internal data structures.

INITIALIZATION OF THREAD SUPPORT

Thread support in the PAPI library can be initialized by calling the following low-level function:

C:
PAPI_thread_init(*handle*, *flag*)

Fortran:
PAPIF_thread_init(*handle*, *flag*, check)

ARGUMENTS

handle -- Pointer to a routine that returns the current thread ID.

flag -- This is reserved for future use and should be set to zero.

This function should be called only once, just after PAPI_library_init, and before any other PAPI calls. If the function is called more than once, the application will exit. Also, *applications that make no use of threads do not need to call this function.*

The following example shows the correct syntax for using PAPI_thread_init with **OpenMP**:

```
C:
#include <papi.h>
#include <omp.h>
if (PAPI_thread_init(omp_get_thread_num,0) != PAPI_OK)
  handle_error(1);

Fortran:
#include “fpapi.h”
#include “omp.h”
EXTERNAL omp_get_thread_num
C Fortran dictates that in order to a pass a subroutine
C as an argument, the subroutine must be
C declared external!
call PAPIF_thread_init(omp_get_thread_num, 0, error)
```

On success, the function, PAPI_thread_init, returns PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix C and for a code example of using PAPI_thread_init with Pthreads, see the next section.

THREAD ID

The identifier of the current thread can be obtained by calling the following low-level function:

C:
PAPI_thread_id()

Fortran:
PAPIF_thread_id(check)

This function calls the thread id function registered by PAPI_thread_init and returns an unsigned long integer containing the thread identifier.

In the following code example, PAPI_thread_init and PAPI_thread_id are used to initialize thread support in the PAPI library and to acquire the identifier of the current thread, respectively, with **Pthreads**:

```
#include <papi.h>
#include <pthread.h>

main()
{
     unsigned long int tid;

     if (PAPI_library_init(PAPI_VER_CURRENT) !=
     PAPI_VER_CURRENT)
       exit(1);

     if (PAPI_thread_init(pthread_self, 0) != PAPI_OK)
       exit(1);

     if ((tid = PAPI_thread_id()) == (unsigned long int)-1)
       exit(1);

     printf("Initial thread id is: %lu\n",tid);
}
```

OUTPUT:

```
Initial thread id is: 0
```

On success, this function returns a valid thread identifier and on error, (unsigned long int) –1 is returned.

More information on this function can be found in Appendix C.

For more code examples of using Pthreads and OpenMP with PAPI, see tests/zero_pthreads.c and tests/zero_omp.c in the papi source distribution, respectively. Also, for a code example of using SMP with PAPI, see tests/zero_smp.c in the papi source distribution.

MPI

MPI is an acronym for **M**essage **P**assing **I**nterface. MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementers, and users. MPI was designed for high performance on both massively parallel machines and on workstation clusters. More information on MPI can be found at http://www-unix.mcs.anl.gov/mpi.

PAPI does support MPI. When using timers in applications that contain multiplexing, profiling, and overflow, MPI uses a default virtual timer and must be converted to a real timer in order to for the application to work properly. Otherwise, the application will exit.

Optionally, the supported tools, Tau and SvPablo, can be used to implement PAPI with MPI.

The following is a code example of using MPI's PI program with PAPI:

```
#include <papi.h>
#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];
{
  int done = 0, n, myid, numprocs, i, rc, retval, EventSet = PAPI_NULL;
  double PI25DT = 3.14159265358979323846264 3;
  double mypi, pi, h, sum, x, a;
  long_long values[1] = {(long_long) 0};

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /*Initialize the PAPI library */
  retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT) {
    fprintf(stderr, "PAPI library init error!\n");
    exit(1); }

  /* Create an EventSet */
  if (PAPI_create_eventset(&EventSet) != PAPI_OK)
    handle_error(1);

/* Add Total Instructions Executed to our EventSet */
  if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
    handle_error(1);

  /* Start counting */
  if (PAPI_start(EventSet) != PAPI_OK)
    handle_error(1);

  while (!done)
  {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);

    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
    }

   /* Read the counters */
   if (PAPI_read(EventSet, values) != PAPI_OK)
     handle_error(1);

   printf("After reading counters: %lld\n",values[0]);

   /* Start the counters */
   if (PAPI_stop(EventSet, values) != PAPI_OK)
     handle_error(1);

   printf("After stopping counters: %lld\n",values[0]);

   MPI_Finalize();
}
```

POSSIBLE OUTPUT (AFTER ENTERING 50, 75, AND 100 AS INPUT):

```
Enter the number of intervals: (0 quits) 50
pi is approximately 3.1416259869230028, Error is 0.0000333333332097
Enter the number of intervals: (0 quits) 75
pi is approximately 3.1416074684045965, Error is 0.0000148148148034
Enter the number of intervals: (0 quits) 100
pi is approximately 3.1416009869231254, Error is 0.0000083333333323
Enter the number of intervals: (0 quits) 0
After reading counters: 117393
After stopping counters: 122921
```

OVERFLOW

WHAT IS AN OVERFLOW?

An overflow is when a particular hardware event exceeds a specified threshold. PAPI provides the ability to call user-defined handlers when an overflow occurs, which is accomplished by setting up a high-resolution interval timer and installing a timer interrupt handler. For the systems that do not support counter overflow at the operating system level, PAPI uses the signal, SIGPROF, by comparing the current counter value against the threshold. If the current value exceeds the threshold, then the user's handler is called from within the signal context with some additional arguments. These arguments allow the user to determine which event overflowed, how much it overflowed, and at what location in the source code.

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

For more information on which platforms support hardware or software overflow, see Appendix I.

BEGINNING OVERFLOWS IN EVENT SETS

An event set can begin registering overflows by calling the following low-level function:

C:
PAPI_overflow(*EventSet, EventCode, threshold, flags, handler*)

Fortran:
NOT IMPLEMENTED

ARGUMENTS

EventSet -- a reference to the event set to use

EventCode -- the counter to be used for overflow detection

threshold -- the overflow threshold value to use

flags -- bit map that controls the overflow mode of operation. **This is currently not used and should be set to 0**.

handler -- the handler function to call upon overflow

This function marks a specific *EventCode* in an *EventSet* to generate an overflow signal after every threshold events are counted. **Only one event in an event set can be used as an overflow trigger.** Subsequent calls to PAPI_overflow replace earlier calls. *To turn off overflow, set the handler to NULL.*

In the following code example, PAPI_overflow is used to mark PAPI_TOT_INS in order to generate an overflow signal after every 100,000 counted events:

```
#include <papi.h>
#include <stdio.h>

#define THRESHOLD  100000

int total = 0;      /* total overflows */

void handler(int EventSet, int EventCode, int EventIndex, long_long
*values, int *threshold, void *context)
{
     fprintf(stderr,"Value %lld at
%p\n",values[EventIndex],PAPI_get_overflow_addres
     s(context));
     total++;
}

main()
{
     int retval, EventSet = PAPI_NULL;

     /* Initialize the PAPI library */
     retval = PAPI_library_init(PAPI_VER_CURRENT);
     if (retval != PAPI_VER_CURRENT)
        handle_error(1);

     /* Create the EventSet */
     if (PAPI_create_eventset(&EventSet) != PAPI_OK)
        handle_error(1);

     /* Add Total Instructions Executed to our EventSet */
     if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
        handle_error(1);

     /* Call handler every 100000 instructions */
     retval = PAPI_overflow(EventSet, PAPI_TOT_INS, THRESHOLD, 0,
handler);
     if (retval != PAPI_OK)
        handle_error(1);

     /* Start counting */
     if (PAPI_start(EventSet) != PAPI_OK)
        handle_error(1);
}
```

On success, this function returns PAPI_OK and on error, a non-zero error code is returned.

For more information on this function, see Appendix C and for more code examples, see the tests/overflow.c or tests/overflow_pthreads.c in the papi source distribution.

ADDRESS OF THE OVERFLOW

The address where an overflow occurred can be obtained by calling the low-level function:

C:
PAPI_get_overflow_address(*context*)

Fortran:
NOT IMPLEMENTED

ARGUMENT

context -- a platform dependent structure containing information about the overflow event. Typically, the signal handler returns this structure automatically.

This function returns the instruction pointer where an overflow occurred and it is often used as part of the overflow handler routine. PAPI_get_overflow_address always returns the value at the offset in the *context* structure where the instruction pointer should be. No validity testing of this structure is done. If an invalid context pointer is passed to this function, the results will be undefined.

For more information on this function, see Appendix C and for code examples, see the above section as well as tests/overflow.c and tests/overflow_pthreads.c in the papi source distribution.

STATISTICAL PROFILING

WHAT IS STATISTICAL PROFILING?

Statistical Profiling is built upon the method of installing and emulating arbitrary callbacks on overflow. Profiling work as follows: when an event exceeds a threshold, the signal, SIGPROF, is delivered with a number of arguments. Among those arguments is the interrupted thread's stack pointer and register set. The register set contains the program counter and the address at which the process was interrupted when the signal was delivered. Performance tools like UNIX prof extract this address and hashes the value into a histogram. At program completion, the histogram is analyzed and associated with symbolic information contained in the executable. GNU prof in conjunction with the –p option of the GCC compiler performs exactly this analysis using the process time as the overflow trigger. PAPI aims to generalize this functionality so that a histogram can be generated using any countable event as the basis for analysis.

GENERATING A PC HISTOGRAM

A PC histogram can be generated on any countable event by calling the following low-level functions:

C:
PAPI_profil(*buf, bufsiz, offset, scale, EventSet, EventCode, threshold, flags*)
PAPI_sprofil(*prof, profcnt, EventSet, EventCode, threshold, flags*)

Fortran:
PAPI_profil(*buf, bufsiz, offset, scale, EventSet, EventCode, threshold, flags,* check)

AGRUMENTS

**buf* -- pointer to profile buffer array.

bufsiz -- number of entries in *buf.

offset -- starting value of lowest memory address to profile.

scale -- scaling factor for bin values.

EventSet -- The PAPI EventSet to profile when it is started.

EventCode -- Code of the Event in the EventSet to profile.

threshold -- threshold value for the Event triggers the handler.

flags -- bit pattern to control profiling behavior. The defined bit values for the flags variable are shown in the table below:

Defined bit	Description
PAPI_PROFIL_POSIX	Default type of profiling.
PAPI_PROFIL_RANDOM	Drop a random 25% of the samples.
PAPI_PROFIL_WEIGHTED	Weight the samples by their value.
PAPI_PROFIL_COMPRESS	Ignore samples if hash buckets get big.

**prof* -- pointer to PAPI_sprofil_t structure.

profcnt -- number of buffers for hardware profiling (**reserved**).

PAPI_profil creates a histogram of overflow counts for a specified region of the application code by using its first four parameters to create the data structures needed by PAPI_sprofil and then calls PAPI_sprofil to do the work. PAPI_sprofil assumes a pre-initialized PAPI_sprofil_t structure and enables profiling for the *EventSet* based on its value. **Note that the EventSet must be in the stopped state in order for both calls to succeed.**

In the following code example, PAPI_profil is used to generate a PC histogram:

```
#include <papi.h>
#include <stdio.h>

main()
{
        int retval;
        int EventSet = PAPI_NULL;
        unsigned long start, end, length;
        PAPI_exe_info_t *prginfo;
        unsigned short *profbuf;

        /* Initialize the PAPI library */
        retval = PAPI_library_init(PAPI_VER_CURRENT);
        if (retval != PAPI_VER_CURRENT & retval > 0) {
          fprintf(stderr,"PAPI library version mismatch!0);
          exit(1); }

        if (retval < 0)
          handle_error(retval);

        if ((prginfo = PAPI_get_executable_info()) == NULL)
          handle_error(1);

        start = (unsigned long)prginfo->text_start;
        end = (unsigned long)prginfo->text_end;
        length = end - start;

        profbuf = (unsigned short *)malloc(length*sizeof(unsigned short));
        if (profbuf == NULL)
          handle_error(1);
        memset(profbuf,0x00,length*sizeof(unsigned short));

        if (PAPI_create_eventset(&EventSet) != PAPI_OK)
          handle_error(retval);

        /* Add Total FP Instructions Executed to our EventSet */
        if (PAPI_add_event(&EventSet, PAPI_FP_INS) != PAPI_OK)
          handle_error(retval);

        if (PAPI_profil(profbuf, length, start, 65536, EventSet, PAPI_FP_INS, 1000000,
        PAPI_PROFIL_POSIX) != PAPI_OK)
          handle_error(1);

        /* Start counting */
        if (PAPI_start(EventSet) != PAPI_OK)
          handle_error(1);
}
```

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

For more information on these functions, see Appendix C and for more code examples, see profile.c and sprofile.c in the PAPI source distribution.

PAPI ERROR HANDLING

ERROR CODES

All of the functions contained in the PAPI library return standardized error codes in which the values that are greater than or equal to zero indicate success and those that are less than zero indicate failure, as shown in the table below:

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_ENOEVNT	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 event sets are currently not counting
-10	PAPI_EISRUN	Event Set is currently running
-11	PAPI_ENOEVST	No such event set 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

CONVERTING ERROR CODES TO ERROR MESSAGES

Error codes can be converted to error messages by calling the following low-level functions:

C:
PAPI_perror(*code*, *destination*, *length*)
PAPI_strerror(*code*)

Fortan:
PAPIF_perror(*code*, *destination*, check)

ARGUMENTS

code -- the error code to interpret

**destination* -- "the error message in quotes"

length -- either 0 or strlen(destination)

PAPI_perror fills the string, *destination,* with the error message corresponding to the error code (*code).* The 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 to stderr.

PAPI_strerror returns a pointer to the error message corresponding to the error code (*code).* If the call fails, the function returns a NULL pointer. Otherwise, a non-NULL pointer is returned. **Note that this function is not implemented in Fortran.**

In the following code example, PAPI_perror is used to convert error codes to error messages:

```
#include <papi.h>
#include <stdio.h>

main()
{
     int EventSet = PAPI_NULL;
     int native = 0x0;
     char error_str[PAPI_MAX_STR_LEN];

     /* Initialize the PAPI library */
     retval = PAPI_library_init(PAPI_VER_CURRENT);

     if (retval != PAPI_VER_CURRENT && retval > 0) {
       fprintf(stderr,"PAPI library version mismatch!\n");
       exit(1); }

     if ((retval = PAPI_create_eventset(&EventSet)) !=
     PAPI_OK)
       {
          fprintf(stderr, "PAPI error %d:
     %s\n",retval,PAPI_strerror(retval));
          exit(1);
       }

     /* Add Total Instructions Executed to our EventSet */
     if ((retval = PAPI_add_event(&EventSet, PAPI_TOT_INS)) !=
     PAPI_OK)
       {
          PAPI_perror(retval,error_str,PAPI_MAX_STR_LEN);
          fprintf(stderr,"PAPI_error %d:
     %s\n",retval,error_str);
          exit(1);
       }

     /* Add native event (0xc1 on hardware counter 1) */
     native = (0xc1 << 8) | 1;
     if ((retval = PAPI_add_event(&EventSet, native)) !=
     PAPI_OK)
       {
          /* Dump error string directly to stderr. */
           PAPI_perror(retval,NULL,NULL);
          exit(1);
       }

     /* Start counting */
     if ((retval = PAPI_start(EventSet)) != PAPI_OK)
       handle_error(retval);
}
```

OUTPUT:

```
Invalid argument
```

Notice that the above output was generated from the last call to PAPI_perror.

On success, PAPI_perror returns PAPI_OK and on error, a non-zero error code is returned.

For more information on these functions, see Appendix C.

PAPI MAILING LISTS

PAPI has the two following mailing lists for users to ask any questions about the project:

To contact a general users' discussion list for PAPI software:

Send mail to ptools-perfapi@ptools.org

This list is a good place for newbie questions and general conversation about how to use PAPI or tools that use PAPI.

To contact a list of developers of PAPI, performance tools and kernel patches:

Send mail to perfapi-devel@ptools.org

This list is intended for more technical discussions about PAPI. It is intended for developers of PAPI and other performance tools and kernel patches to share observations and insights. Interested hackers are welcomed. All the CVS log messages go here.

To subscribe to either of these mailing lists:

Send a message with blank subject to majordomo@ptools.org. In the body of the message, include 'subscribe <mailing_list>' without the single quotes. If you're having trouble, try sending 'help' in the body to the same address. Should you become hopelessly confused, send mail to the administrator.

APPENDICES

APPENDIX A. TABLE OF PRESET EVENTS AND THEIR AVAILABILITY ON SOME PLATFORMS

The following is a table of hardware events that are defined in the header file, *papiStdEventDefs.h*, which are 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. Therefore, all of these events are not guaranteed to be present on all platforms. The table indicates which events are available on some platforms. Please check your platform's documentation or run tests/avail.c in the papi source distribution to determine the preset events that are available on your platform. Note that these values should not be changed by the user.**

Key:
- indicates that the preset event is available and derived by using a combination of counters.
- indicates that the preset event is available and not derived.

PRESET NAME	DESCRIPTION	AMD ATHLON K7	IBM POWER3	INTEL/HP ITANIUM	INTEL PENTIUM III	MIPS R12K	ULTRA SPARC I
PAPI_L1_DCM	Level 1 data cache misses	•	•	•	•	•	
PAPI_L1_ICM	Level 1 instruction cache misses	•	•	•	•	•	•
PAPI_L2_DCM	Level 2 data cache misses	•		•	•	•	
PAPI_L2_ICM	Level 2 instruction cache misses	•		•	•	•	
PAPI_L3_DCM	Level 3 data cache misses			•			
PAPI_L3_ICM	Level 3 instruction cache misses			•			
PAPI_L1_TCM	Level 1 total cache misses	•	•	•	•	•	
PAPI_L2_TCM	Level 2 total cache misses	•		•	•	•	•
PAPI_L3_TCM	Level 3 total cache misses			•			
PAPI_CA_SNP	Requests for a Snoop		•				•
PAPI_CA_SHR	Requests for access to shared cache line (SMP)		•		•	•	
PAPI_CA_CLN	Requests for access to clean cache line (SMP)				•		
PAPI_CA_INV	Cache Line Invalidation (SMP)				•	•	•
PAPI_CA_ITV	Cache Line Intervention (SMP)		•		•	•	
PAPI_L3_LDM	Level 3 load misses			•			
PAPI_L3_STM	Level 3 store misses			•			
PAPI_BRU_IDL	Cycles branch units are idle		•				
PAPI_FXU_IDL	Cycles integer units are idle		•				
PAPI_FPU_IDL	Cycles floating point units are idle		•				
PAPI_LSU_IDL	Cycles load/store units are idle		•				
PAPI_TLB_DM	Data translation lookaside buffer misses	•		•			
PAPI_TLB_IM	Instruction translation lookaside buffer misses	•		•	•		
PAPI_TLB_TL	Total translation lookaside buffer misses	•	•			•	
PAPI_L1_LDM	Level 1 load misses	•	•	•	•		•

PRESET NAME	DESCRIPTION	AMD ATHLON K7	IBM POWER3	INTEL/HP ITANIUM	INTEL PENTIUM III	MIPS R12K	ULTRA SPARC I
PAPI_L1_STM	Level 1 store misses	●	●		●		●
PAPI_L2_LDM	Level 2 load misses	●	●	●			
PAPI_L2_STM	Level 2 store misses	●	●	●			
PAPI_BTAC_M	Branch target address cache (BTAC) misses		●		●		
PAPI_PRF_DM	Pre-fetch data instruction caused a miss		●			●	
PAPI_L3_DCH	Level 3 Data Cache Hit			●			
PAPI_TLB_SD	Translation lookaside buffer shootdowns (SMP)		●				
PAPI_CSR_FAL	Failed store conditional instructions		●			●	
PAPI_CSR_SUC	Successful store conditional instructions		●			●	
PAPI_CSR_TOT	Total store conditional instructions		●			●	
PAPI_MEM_SCY	Cycles Stalled Waiting for Memory Access		●	●		●	
PAPI_MEM_RCY	Cycles Stalled Waiting for Memory Read		●				
PAPI_MEM_WCY	Cycles Stalled Waiting for Memory Write		●				
PAPI_STL_ICY	Cycles with No Instruction Issue		●	●			
PAPI_FUL_ICY	Cycles with Maximum Instruction Issue						
PAPI_STL_CCY	Cycles with No Instruction Completion		●				
PAPI_FUL_CCY	Cycles with Maximum Instruction Completion						
PAPI_HW_INT	Hardware interrupts	●			●		
PAPI_BR_UCN	Unconditional branch instructions executed	●					
PAPI_BR_CN	Conditional branch instructions executed	●	●		●		
PAPI_BR_TKN	Conditional branch instructions taken	●		●	●		
PAPI_BR_NTK	Conditional branch instructions not taken	●		●	●		
PAPI_BR_MSP	Conditional branch instructions mispredicted	●	●	●	●	●	●
PAPI_BR_PRC	Conditional branch instructions correctly predicted	●	●	●	●		
PAPI_FMA_INS	FMA instructions completed		●				
PAPI_TOT_IIS	Total instructions issued		●		●	●	●
PAPI_TOT_INS	Total instructions executed	●	●	●	●	●	●

PRESET NAME	DESCRIPTION	AMD ATHLON K7	IBM POWER3	INTEL/HP ITANIUM	INTEL PENTIUM III	MIPS R12K	ULTRA SPARC I
PAPI_INT_INS	Integer instructions executed		●				
PAPI_FP_INS	Floating point instructions executed		●	●	●	●	●
PAPI_LD_INS	Load instructions executed		●	●		●	●
PAPI_SR_INS	Store instructions executed		●	●		●	●
PAPI_BR_INS	Total branch instructions executed	●	●	●	●	●	
PAPI_VEC_INS	Vector/SIMD instructions executed	●			●		
PAPI_FLOPS	Floating Point Instructions executed per second		●	●	●	●	●
PAPI_RES_STL	Cycles processor is stalled on resource	●			●		
PAPI_FP_STAL	Cycles any FP units are stalled						
PAPI_TOT_CYC	Total cycles	●	●	●	●	●	●
PAPI_IPS	Instructions executed per second	●	●		●	●	●
PAPI_LST_INS	Total load/store instructions executed		●	●		●	
PAPI_SYC_INS	Synchronization instructions executed		●				
PAPI_L1_DCH	L1 data cache hits	●		●	●		
PAPI_L2_DCH	L2 data cache hits	●					
PAPI_L1_DCA	L1 data cache accesses	●		●	●		
PAPI_L2_DCA	L2 data cache accesses	●		●	●		
PAPI_L3_DCA	L3 data cache accesses			●			
PAPI_L1_DCR	L1 data cache reads						●
PAPI_L2_DCR	L2 data cache reads	●		●	●		
PAPI_L3_DCR	L3 data cache reads			●			
PAPI_L1_DCW	L1 data cache writes						●
PAPI_L2_DCW	L2 data cache writes	●		●	●		
PAPI_L3_DCW	L3 data cache writes			●			
PAPI_L1_ICH	L1 instruction cache hits				●		●
PAPI_L2_ICH	L2 instruction cache hits				●		●
PAPI_L3_ICH	L3 instruction cache hits			●			
PAPI_L1_ICA	L1 instruction cache accesses	●			●		●

PRESET NAME	DESCRIPTION	AMD ATHLON K7	IBM POWER3	INTEL/HP ITANIUM	INTEL PENTIUM III	MIPS R12K	ULTRA SPARC I
PAPI_L2_ICA	L2 instruction cache accesses	•			•		
PAPI_L3_ICA	L3 instruction cache accesses						
PAPI_L1_ICR	L1 instruction cache reads	•		•	•		
PAPI_L2_ICR	L2 instruction cache reads			•	•		
PAPI_L3_ICR	L3 instruction cache reads			•			
PAPI_L1_ICW	L1 instruction cache writes						
PAPI_L2_ICW	L2 instruction cache writes						
PAPI_L3_ICW	L3 instruction cache writes						
PAPI_L1_TCH	L1 total cache hits						
PAPI_L2_TCH	L2 total cache hits				•		•
PAPI_L3_TCH	L3 total cache hits						
PAPI_L1_TCA	L1 total cache accesses	•			•		
PAPI_L2_TCA	L2 total cache accesses				•		•
PAPI_L3_TCA	L3 total cache accesses						
PAPI_L1_TCR	L1 total cache reads						
PAPI_L2_TCR	L2 total cache reads				•		
PAPI_L3_TCR	L3 total cache reads						
PAPI_L1_TCW	L1 total cache writes						
PAPI_L2_TCW	L2 total cache writes				•		
PAPI_L3_TCW	L3 total cache writes						
PAPI_FML_INS	Floating Multiply instructions				•		
PAPI_FAD_INS	Floating Add instructions						
PAPI_FDV_INS	Floating Divide instructions		•		•		
PAPI_FSQ_INS	Floating Square Root instructions		•				
PAPI_FNV_INS	Floating Inverse instructions						

APPENDIX B. HIGH-LEVEL API

The simple interface implemented by the six routines of the High-Level PAPI API allows the user to access and count specific hardware events. It should be noted that this API could 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, 0). **Note that the High-Level API fully supports both C and Fortran. For full details on the calling semantics of these functions, please refer to the PAPI Programmer's Reference.**

APPENDIX C. LOW-LEVEL API

The functions of the Low Level PAPI API provide greatly increased efficiency and functionality over the high level API presented in the previous appendix. 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. **Note that most functions are implemented in both C and Fortran, but some are implemented in only one of these two languages. For full details on the calling semantics of these functions, please refer to the PAPI Programmer's Reference.**

APPENDIX D. PAPI SUPPORTED PLATFORMS

HARDWARE	OPERATING SYSTEM	REQUIREMENTS
Alpha EV6 & EV67	Tru64 Unix	Contact dcpi@hp.com for required system software
Alpha EV6 & EV67	Linux	IProbe patch (included)
AMD Athlon	Linux 2.2, 2.4	Mikael Pettersson's Perfctr kernel patch for Linux on web site
Cray SV1, SV2, & T3E	Unicos	None
IBM POWER3, 604, & 604e	AIX 4.3.3	Pmtoolkit from IBM alphaWorks(More information on web site)
IBM POWER4, POWER3, 604, & 604e	AIX 5.1	bos.pmapi must be installed
Intel/HP Itanium I & II	Linux 2.4	None
Intel Pentium Series through Pentium III	Linux 2.2, 2.4	Mikael Pettersson's Perfctr kernel patch for Linux on web site
Intel Pentium Series through Pentium III	Windows NT, 2000, XP	Administrator privilege for installation
MIPS R10K & R12K	Irix 6.5	None
UltraSparc I, II, & III	Solaris 2.8 or newer	None

More information about the various supported platforms can be found at http://icl.cs.utk.edu/projects/papi/links.

APPENDIX E. TABLE OF NATIVE ENCODING FOR THE VARIOUS PLATFORMS

PLATFORM	**NATIVE ENCODING**
Alpha EV6 & EV67	((register_code & 0xffffff) << 8 | (register_number & 0xff))
AMD Athlon	(*event_code* << 8) | (*hw_counter_num*) *event_code* = 16 bit event selector code and unit mask. *hw_counter_num* = Event register number 0 through 1.
Cray SV1, SV2, & T3E	(*mask* & 0x7) (sel2 & 0xf) (sel1 & 0xf) (sel0 & 0x1) The *mask* indicates which of the three counters you want counted. If more than one bit is set in the mask, then the counters will be summed into a single event when read. The mask must be non-zero!
IBM POWER4, POWER3, 604, 604e	Low 8 bits indicate which counter number: 0 - 7 Bits 8-16 indicate which event number: 0 - 94
Intel/HP Itanium	((register_code & 0xffffff) << 8 | (register_number & 0xff))
Intel Pentium Series	((register_code & 0xffffff) << 8 | (register_number & 0xff))
MIPS R10K & R12K	Low 8 bits indicate which event number: 0 - 31
UltraSparc I, II, & III	8 bit event code in bits 8-16, counter number in bits 0-7

For more information on the native encoding for your platform, please see the README file for your platform in the papi source distribution.

APPENDIX F. TABLE OF OVERHEAD FOR THE VARIOUS PLATFORMS

< under development >

APPENDIX G. TABLE FOR MULTIPLEXING

< under development >

APPENDIX H. TABLE FOR OVERFLOW

< under development >

APPENDIX I. PAPI SUPPORTED TOOLS

TOOLS	**LINKS**
Cactus	http://www.cactuscode.org
DEEP/PAPI	http://www.psrv.com/deep_papi_top.html
DynaProf	http://www.cs.utk.edu/~mucci/dynaprof/
GProf	http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html
Paradyn	http://www.cs.wisc.edu/paradyn
Perfometer	http://icl.cs.utk.edu/projects/papi/tools
SCALEA	http://www.par.univie.ac.at/project/scalea/
SvPablo	http://vibes.cs.uiuc.edu/Software/SvPablo/svPablo.htm
TAU	http://www.cs.uoregon.edu/research/paracomp/tau
Vampir	http://www.pallas.com/e/products/vampir/index.htm
Vprof	http://aros.ca.sandia.gov/~cljanss/perf/vprof

BIBLIOGRAPHY

Browne, S., J. Dongarra J., Garner N., London K., and Mucci, P., "A Portable Programming Interface for Performance Evaluation on Modern Processors," University of Tennessee Technical Report, Knoxville, Tennessee, July 2000. http://icl.cs.utk.edu/papi/documents

Browne, S., Dongarra J., Garner N., London K., and Mucci, P., "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters," Proc. *SC'2000*, November 2000. http://icl.cs.utk.edu/papi/documents

Dongarra, J., London, K., Moore, S., Mucci, P., and Terpstra, D. "Using PAPI for Hardware Performance Monitoring on Linux Systems," *Conference on Linux Clusters: The HPC Revolution*, Urbana, Illinois, June 25-27, 2001. http://icl.cs.utk.edu/papi/documents

London, K., Moore, S., Mucci, P., Seymour, K., and Luczak, R. "The PAPI Cross-Platform Interface to Hardware Performance Counters," Department of Defense Users' Group Conference Proceedings, Biloxi, Mississippi, June 18-21, 2001. http://icl.cs.utk.edu/papi/documens

London, K., Dongarra, J., Moore, S., Mucci, P., Seymour, K., and Spencer, T. "End-user Tools for Application Performance Analysis Using Hardware Counters," *International Conference on Parallel and Distributed Computing Systems*, Dallas, TX, August 8-10, 2001. http://icl.cs.utk.edu/papi/documents

Mucci, P., Moore, S., and Smeds, Nils. "Performance Tuning Using Hardware Counter Data," Proc. *SC'2001*, November 2001. http://icl.cs.utk.edu/papi/documents

Mucci, P. "The IA64 Hardware Performance Monitor and PAPI", *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2001. http://icl.cs.utk.edu/papi/documents

Mucci, P. "PAPI -- The Performance Application Programming Interface," April 2000. http://icl.cs.utk.edu/papi/documents

"PAPI Programmer Reference." December 2001. http://icl.cs.utk.edu/papi/files/html_man/papi.html

"PAPI Software Specification." December 2001. http://icl.cs.utk.edu/papi/files/papispec21.html

"POSIX Threads Programming." http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html