CS484: Parallel Programming - Spring 2017
PAPI - Quick Reference Manual

February 11, 2017

1 Background

PAPI is an acronym for Performance Application Programming Interface. The PAPI Project is being developed at the University of Tennessees 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.

These counters provide developers with valuable information about sections of their code that can be improved. Some goals of the PAPI Project are as follows:

- To provide cross platform performance analysis tools
- To present a set of standard definitions for performance metrics on all platforms

2 Architecture

Figure 1 shows the components of the PAPI API. The Portable Layer consists of the API and machine independent support functions. The Machine Specific Layer defines and exports an 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.

3 PAPI Interface

3.1 Events

Events are occurrences of specific signals related to a processors 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

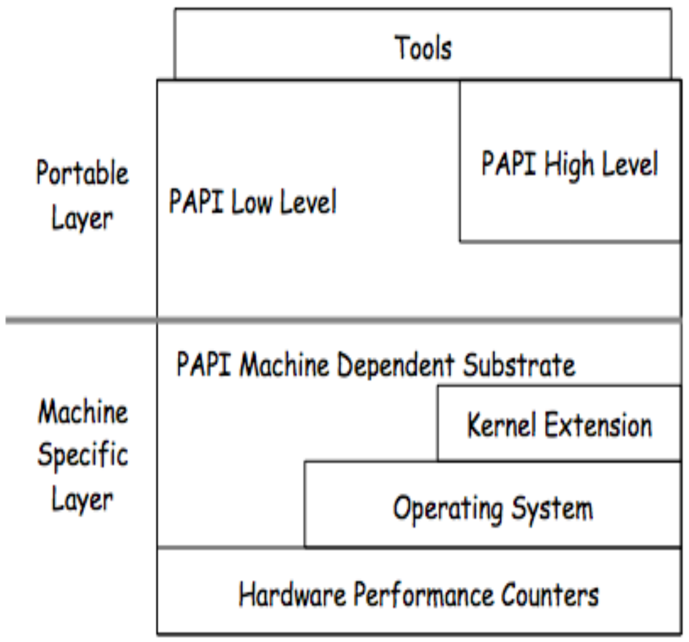

Figure 1: PAPI Software Architecture

events that are native 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.

3.1.1 Native Events

Native events comprise the set of all events that are countable by the CPU. There are generally far more native events available than can be mapped onto PAPI preset events. Even if no preset event is available that exposes a given native event, native events can still be accessed directly. To use native events effectively you should be very familiar with the particular platform in use. The utility code *util/papi_native_avail* provides insight into the names of the native events for a specific platform.

3.1.2 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, Total L1 Cache Misses (PAPI_L1_TCM) might be the sum of L1 Data Misses and L1 Instruction Misses on a given platform. A preset can be either directly available as a single counter, derived using a combination of counters, or unavailable on any particular platform. PAPI provides functions to query both types of events on a processor.

3.2 Counters

3.2.1 High-level API

The high-level API provides the ability to start, stop, and read the counters for a specified list of events. It is meant for programmers wanting simple event measurements using 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 calls). This ease of use comes with somewhat higher overhead and loss of flexibility. Some of the useful high-level API calls are listed below :

- *PAPI_num_counters()* : This function returns the number of hardware counters supported by a processor
- *PAPI_start_counters(*events, array_length)* : Initilizes the PAPI counters and starts counting for the events listed in the array *events*
- *PAPI_flips(*real_time, *proc_time, *flpins, *mflips)*
- *PAPI_flops(*real_time, *proc_time, *flpins, *mflops)*
- *PAPI_ipc(*real_time, *proc_time, *ins, *ipc)*

where *real_time – the total real (wallclock) time since the first rate call,
*proc_time – the total process time since the first rate call,
*flpins – the total floating point instructions since the first rate call,
*mflips, *mflops Millions of floating point operations or instructions per second achieved since the latest rate call,
*ins – the total instructions executed since the first PAPI_ipc call. *ipc instructions per cycle achieved since the latest PAPI_ipc call

3.2.2 Functions to access counters

The following functions can be used to access the counters.

- *PAPI_read_counters(*values, array_length)*
- *PAPI_accum_counters(*values, array_length)*
- *PAPI_stop_counters(*values, array_length)*

where **values* – an array where to put the counter values. *array_length* – the number of items in the *values array. *PAPI_read_counters*, *PAPI_accum_counters* and *PAPI_stop_counters* all capture the values of the currently running counters into the array, values.
PAPI_read_counters copies the current counts into the elements of the values array, resets the counters to zero, and leaves the counters running.
PAPI_accum_counters adds the current counts into the elements of the values array and resets the counters to zero, leaving the counters running.

A possible sequence to access and read the PAPI counters is provided below.

```
#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);
}
```

3.2.3 Low-level API

The low-level API (Application Programming Interface) manages hardware events in user-defined groups called Event Sets. It is meant for experienced application programmers and tool developers wanting fine-grained measurement and control of the PAPI interface. Unlike the high-level interface, it allows both PAPI preset and native events. Other 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.

4 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. The user specifies the events to be added to an Event Set, and other attributes, such as: the counting domain (user or kernel), whether or not the events in the Event Set are to be multiplexed, and whether the Event Set is to be used for overflow or profiling.

4.1 Creating an Event Set

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

```
PAPI_create_eventset (*EventSet)
```

EventSet – Address of an integer location to store the new EventSet handle. Once it has been created, the user may add hardware events to the EventSet by calling *PAPI_add_event* or *PAPI_add_events*. On success, this function returns *PAPI_OK*. On error, a non-zero error code is returned.

4.2 Adding Events to an EventSet

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

- *PAPI_add_event(EventSet, EventCode)*
- *PAPI_add_events(EventSet, *EventCode, number)*

A code example is provided below :

```
#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.

A full example of how to use PAPI is provided below :

```
#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.