Profiling programs

Daniel Lucio
Pragneshkumar Patel

XSEDE
Extreme Science and Engineering Discovery Environment

A UT/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES
NICS

Optimizing code

After your code is compiled, debugged, and capable of running to completion or planned termination, you can begin looking for ways in which to improve execution speed. In general, the opportunities for optimization fall into three categories (which require progressively more programmer effort):

- Improving overall I/O
- Improving use of compiler-generated optimizations
- Analyzing code behavior and rewriting code to optimize performance

Code Profiling

- Software profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or frequency and duration of function calls. The most common use of profiling information is to improve program optimization.

- Profiling is achieved by instrumenting either the program source code or its binary executable form using libraries or tools called profiler. A number of different techniques may be used by profilers, such as event-based, statistical, instrumented, and simulation methods.

Code Profiling

- Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing.
- This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster. It can also tell you which functions are being called more or less often than you expected. This may help you spot bugs that had otherwise been unnoticed!
- How your program is run will affect the information that shows up in the profile data!!!

What is it?

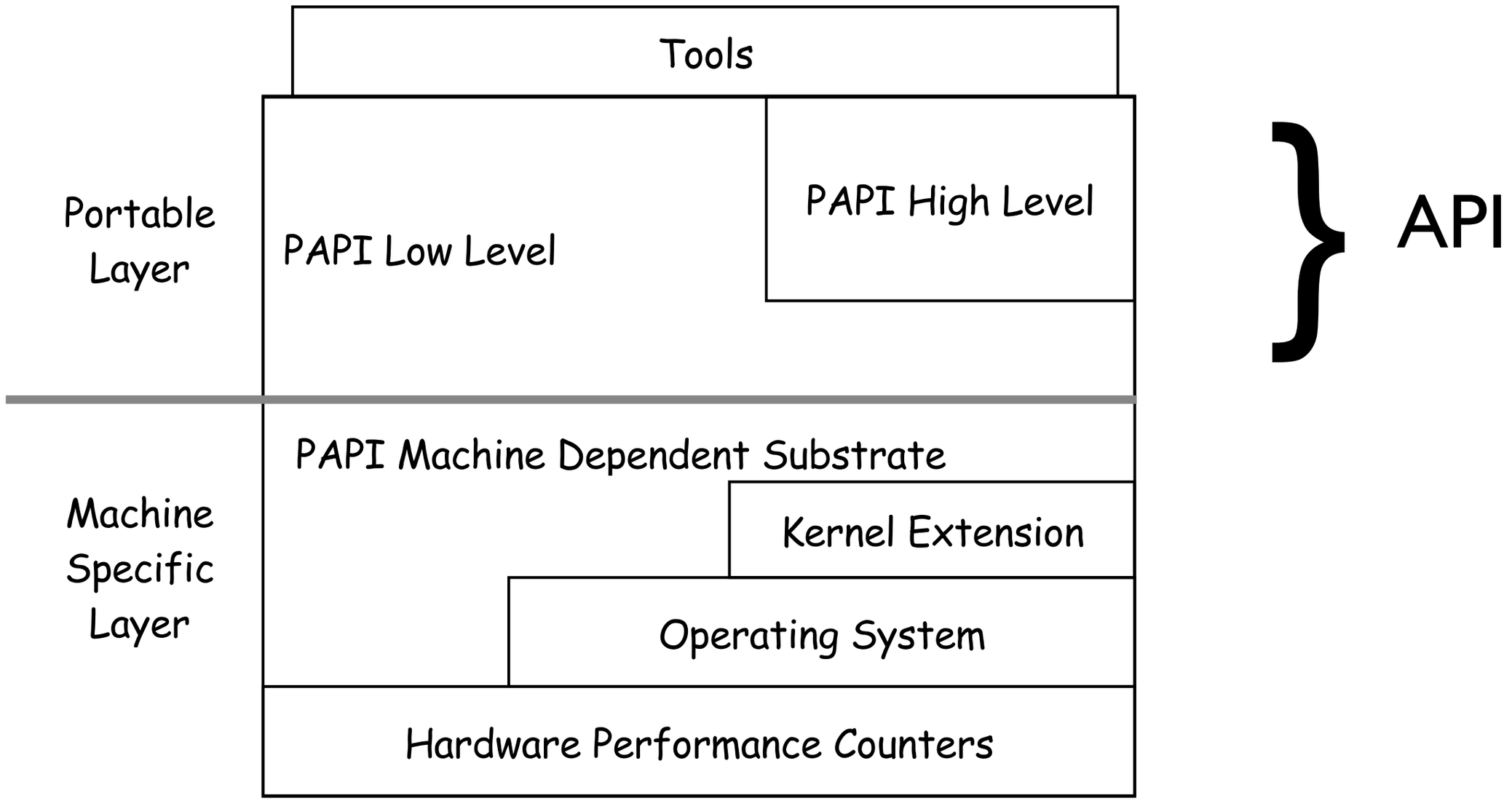
PAPI

- The Performance Application Programming Interface (PAPI) provides a standard application programming interface (in the form of a library) for accessing *hardware performance counters* found on most modern microprocessor-based computer systems.
- It is being widely used to collect low level performance metrics (e.g. instruction counts, clock cycles, cache misses) of computer systems running UNIX/Linux operating systems.
- Any of over 100 preset events can be counted through either a simple high level programming interface or a more complete low level interface from either C or Fortran.
- PAPI provides predefined high level hardware events summarized from popular processors and direct access to low level native events of one particular processor.

Architecture

PAPI

- Operating system support for accessing hardware counters is needed to use PAPI. For example, a Linux/x86 kernel has to be patched with a performance monitoring counters driver to support PAPI.



Hardware Performance Counters

PAPI

- Hardware performance counters, or hardware counters are a set of *special-purpose registers* built into modern microprocessors to store the counts of hardware-related activities within computer systems.
- The number of available hardware counters in a processor is limited while each CPU model might offer a lot of different events to measure. Each counter can be programmed with the index of an event type to be monitored, like a L1 cache miss or a branch misprediction.

CPUs and available counters

Pentium III	2
AMD Athlon	4
Pentium 4	18

- Hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units, caches and main memory etc.

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 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.

PAPI predefined named presets events

- The PAPI library names a number of predefined, or preset events.
- This set is a collection of events typically found in many CPUs that provide performance counters.
- A PAPI preset event name is mapped onto one or more of the countable native events on each hardware platform.
- On any particular platform, the preset can either be directly available as a single counter, derived using a combination of counters or unavailable.

PAPI predefined named presets events

PAPI

- The PAPI presets events can be “loosely” broken int o the following categories:
  - Conditional Branching (PAPI_BR_CN - Conditional Branch Instructions)
  - Cache Requests (PAPI_CA_CLN - Requests for exclusive access to clean cache line)
  - Conditional Store (PAPI_CSR_FAL - Failed store conditional instructions)
  - Floating Point Operations (PAPI_FP_INS - Floating point instructions)
  - Instruction Counting (PAPI_TOT_INS - Instructions issued)
  - Cache access (PAPI_L1_DCA - L1 data cache accesses)
  - Data Access (PAPI_LD__INS - Load Instructions)
  - TLB Operations (PAPI_TLB_DM - Data translation lookaside buffer misses)

PAPI native events

- In addition to the predefined PAPI preset events, the PAPI library also exposes a majority of the events native to each platform.
- Native events form the basic building blocks for PAPI presets.
- They can also be used directly to access functions specific to a given platform.
- Since all native events are specific to any given platform, the names for these events will be unique to each platform. Native events for a given platform can be discovered by combing the PAPI_enum_event() and PAPI_event_code_to_name(), or, PAPI_get_event_info() functions.

papi_avail utility program

PAPI

```
lucio@nid00453:~> aprun -n1 papi_avail
Available events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.3.0.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz (45)
CPU Revision             : 7.000000
CPUID Info               : Family: 6  Model: 45  Stepping: 7
CPU Max Megahertz        : 2601
CPU Min Megahertz        : 1200
Hdw Threads per core     : 2
Cores per Socket         : 8
Sockets                  : 2
NUMA Nodes               : 2
CPUs per Node            : 16
Total CPUs               : 32
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 64
--------------------------------------------------------------------------------

    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
PAPI_CA_SNP  0x80000009  No    No   Requests for a snoop
PAPI_CA_SHR  0x8000000a  No    No   Requests for exclusive access to shared cache line
PAPI_CA_CLN  0x8000000b  No    No   Requests for exclusive access to clean cache line
PAPI_CA_INV  0x8000000c  No    No   Requests for cache line invalidation
PAPI_CA_ITV  0x8000000d  No    No   Requests for cache line intervention
PAPI_L3_LDM  0x8000000e  No    No   Level 3 load misses
PAPI_L3_STM  0x8000000f  No    No   Level 3 store misses
PAPI_BRU_IDL 0x80000010  No    No   Cycles branch units are idle
PAPI_FXU_IDL 0x80000011  No    No   Cycles integer units are idle
PAPI_FPU_IDL 0x80000012  No    No   Cycles floating point units are idle
PAPI_LSU_IDL 0x80000013  No    No   Cycles load/store units are idle
PAPI_TLB_DM  0x80000014  Yes   Yes  Data translation lookaside buffer misses
PAPI_TLB_IM  0x80000015  Yes   No   Instruction translation lookaside buffer misses
PAPI_TLB_TL  0x80000016  No    No   Total translation lookaside buffer misses
PAPI_L1_LDM  0x80000017  Yes   No   Level 1 load misses
PAPI_L1_STM  0x80000018  Yes   No   Level 1 store misses
PAPI_L2_LDM  0x80000019  No    No   Level 2 load misses
PAPI_L2_STM  0x8000001a  Yes   No   Level 2 store misses
PAPI_BTAC_M  0x8000001b  No    No   Branch target address cache misses
PAPI_PRF_DM  0x8000001c  No    No   Data prefetch cache misses
PAPI_L3_DCH  0x8000001d  No    No   Level 3 data cache hits
PAPI_TLB_SD  0x8000001e  No    No   Translation lookaside buffer shootdowns
PAPI_CSR_FAL 0x8000001f  No    No   Failed store conditional instructions
PAPI_CSR_SUC 0x80000020  No    No   Successful store conditional instructions
PAPI_CSR_TOT 0x80000021  No    No   Total store conditional instructions
PAPI_MEM_SCY 0x80000022  No    No   Cycles Stalled Waiting for memory accesses
PAPI_MEM_RCY 0x80000023  No    No   Cycles Stalled Waiting for memory Reads
PAPI_MEM_WCY 0x80000024  No    No   Cycles Stalled Waiting for memory writes
PAPI_STL_ICY 0x80000025  Yes   No   Cycles with no instruction issue
PAPI_FUL_ICY 0x80000026  No    No   Cycles with maximum instruction issue
PAPI_STL_CCY 0x80000027  No    No   Cycles with no instructions completed
PAPI_FUL_CCY 0x80000028  No    No   Cycles with maximum instructions completed
PAPI_HW_INT  0x80000029  No    No   Hardware interrupts
PAPI_BR_UCN  0x8000002a  Yes   Yes  Unconditional branch instructions
PAPI_BR_CN   0x8000002b  Yes   No   Conditional branch instructions
PAPI_BR_TKN  0x8000002c  Yes   Yes  Conditional branch instructions taken
PAPI_BR_NTK  0x8000002d  Yes   No   Conditional branch instructions not taken
PAPI_BR_MSP  0x8000002e  Yes   No   Conditional branch instructions mispredicted
PAPI_BR_PRC  0x8000002f  Yes   Yes  Conditional branch instructions correctly predicted
PAPI_FMA_INS 0x80000030  No    No   FMA instructions completed
PAPI_TOT_IIS 0x80000031  No    No   Instructions issued
PAPI_TOT_INS 0x80000032  Yes   No   Instructions completed
PAPI_INT_INS 0x80000033  No    No   Integer instructions
PAPI_FP_INS  0x80000034  Yes   Yes  Floating point instructions
PAPI_LD_INS  0x80000035  Yes   No   Load instructions
PAPI_SR_INS  0x80000036  Yes   No   Store instructions
PAPI_BR_INS  0x80000037  Yes   No   Branch instructions
PAPI_VEC_INS 0x80000038  No    No   Vector/SIMD instructions (could include integer)
PAPI_RES_STL 0x80000039  No    No   Cycles stalled on any resource
PAPI_FP_STAL 0x8000003a  No    No   Cycles the FP unit(s) are stalled
PAPI_TOT_CYC 0x8000003b  Yes   No   Total cycles
PAPI_LST_INS 0x8000003c  No    No   Load/store instructions completed
PAPI_SYC_INS 0x8000003d  No    No   Synchronization instructions completed
PAPI_L1_DCH  0x8000003e  No    No   Level 1 data cache hits
PAPI_L2_DCH  0x8000003f  Yes   Yes  Level 2 data cache hits
PAPI_L1_DCA  0x80000040  No    No   Level 1 data cache accesses
PAPI_L2_DCA  0x80000041  Yes   No   Level 2 data cache accesses
PAPI_L3_DCA  0x80000042  Yes   Yes  Level 3 data cache accesses
PAPI_L1_DCR  0x80000043  No    No   Level 1 data cache reads
PAPI_L2_DCR  0x80000044  Yes   No   Level 2 data cache reads
PAPI_L3_DCR  0x80000045  Yes   No   Level 3 data cache reads
PAPI_L1_DCW  0x80000046  No    No   Level 1 data cache writes
PAPI_L2_DCW  0x80000047  Yes   No   Level 2 data cache writes
PAPI_L3_DCW  0x80000048  Yes   No   Level 3 data cache writes
PAPI_L1_ICH  0x80000049  No    No   Level 1 instruction cache hits
PAPI_L2_ICH  0x8000004a  Yes   No   Level 2 instruction cache hits
PAPI_L3_ICH  0x8000004b  No    No   Level 3 instruction cache hits
PAPI_L1_ICA  0x8000004c  No    No   Level 1 instruction cache accesses
PAPI_L2_ICA  0x8000004d  Yes   No   Level 2 instruction cache accesses
PAPI_L3_ICA  0x8000004e  Yes   No   Level 3 instruction cache accesses
PAPI_L1_ICR  0x8000004f  No    No   Level 1 instruction cache reads
PAPI_L2_ICR  0x80000050  Yes   No   Level 2 instruction cache reads
PAPI_L3_ICR  0x80000051  Yes   No   Level 3 instruction cache reads
PAPI_L1_ICW  0x80000052  No    No   Level 1 instruction cache writes
PAPI_L2_ICW  0x80000053  No    No   Level 2 instruction cache writes
PAPI_L3_ICW  0x80000054  No    No   Level 3 instruction cache writes
PAPI_L1_TCH  0x80000055  No    No   Level 1 total cache hits
PAPI_L2_TCH  0x80000056  No    No   Level 2 total cache hits
PAPI_L3_TCH  0x80000057  No    No   Level 3 total cache hits
PAPI_L1_TCA  0x80000058  No    No   Level 1 total cache accesses
PAPI_L2_TCA  0x80000059  Yes   Yes  Level 2 total cache accesses
PAPI_L3_TCA  0x8000005a  Yes   No   Level 3 total cache accesses
PAPI_L1_TCR  0x8000005b  No    No   Level 1 total cache reads
PAPI_L2_TCR  0x8000005c  Yes   Yes  Level 2 total cache reads
PAPI_L3_TCR  0x8000005d  Yes   Yes  Level 3 total cache reads
PAPI_L1_TCW  0x8000005e  No    No   Level 1 total cache writes
PAPI_L2_TCW  0x8000005f  Yes   No   Level 2 total cache writes
PAPI_L3_TCW  0x80000060  Yes   No   Level 3 total cache writes
PAPI_FML_INS 0x80000061  No    No   Floating point multiply instructions
PAPI_FAD_INS 0x80000062  No    No   Floating point add instructions
PAPI_FDV_INS 0x80000063  Yes   No   Floating point divide instructions
PAPI_FSQ_INS 0x80000064  No    No   Floating point square root instructions
PAPI_FNV_INS 0x80000065  No    No   Floating point inverse instructions
PAPI_FP_OPS  0x80000066  Yes   Yes  Floating point operations
PAPI_SP_OPS  0x80000067  Yes   Yes  Floating point operations; optimized to count scaled single
precision vector operations
PAPI_DP_OPS  0x80000068  Yes   Yes  Floating point operations; optimized to count scaled double
precision vector operations
PAPI_VEC_SP  0x80000069  Yes   Yes  Single precision vector/SIMD instructions
PAPI_VEC_DP  0x8000006a  Yes   Yes  Double precision vector/SIMD instructions
PAPI_REF_CYC 0x8000006b  Yes   No   Reference clock cycles
--------------------------------------------------------------------------------
Of 108 possible events, 50 are available, of which 17 are derived.

avail.c
```

The Library

PAPI

- **High Level Functions**: The high-level functions are self-initializing. You may mix high- and low-level functions, but low-level functions must follow either a high-level function call or a call to PAPI_library_init().

- **Low Level Functions**: PAPI also provides an advanced interface for instrumenting applications. The PAPI library must be initialized before calling any of these functions; initialization can be done by issuing either a high-level function call or a call to PAPI_library_init().

- **PAPI Utilities**: The PAPI utilities provide more information about the system and the PAPI events that can be examined.

Calling Interface

PAPI

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)
```

Almost all he C function calls have equivalent Fortran 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.

High-level Functions

PAPI

PAPI_num_counters() PAPIF_num_counters()	Return the number of hardware counters available on the system.
PAPI_flips() PAPIF_flips()	Return the Mflips (floating point instruction rate per second), real, and CPU time.
PAPI_flops() PAPIF_flops()	Return the Mflops (floating point operation rate per second), real, and CPU time.
PAPI_ipc() PAPIF_ipc()	Return the instructions per cycle, real, and CPU time.
PAPI_epc() PAPIF_epc()	Get arbitrary events per cycle, real and processor time, reference and core cycles.
PAPI_accum_counters() PAPIF_accum_counters()	Add current counts to array, and reset counters.
PAPI_read_counters() PAPIF_read_counters()	Copy current counts to array, and reset counters.
PAPI_start_counters() PAPIF_start_counters()	Start counting hardware events.
PAPI_stop_counters() PAPIF_stop_counters()	Stop counting events and return the current counts.

Low-level Functions

PAPI_accum(3) PAPIF_accum(3)	Accumulate and reset hardware events from an event set.
PAPI_add_event(3) PAPIF_add_event(3)	Add a single PAPI preset or a native hardware event to an event set.
PAPI_add_events(3) PAPIF_add_events(3)	Add an array of PAPI preset or native hardware events to an event set.
PAPI_add_named_event(3) PAPIF_add_named_event(3)	Add an event by name to a PAPI event set.
PAPI_assign_eventset_component(3) PAPIF_assign_eventset_component(3)	Assign a component index to an existing but empty event set.
PAPI_attach(3) PAPIF_attach(3)	Attach PAPI event set to the specified thread ID.
PAPI_cleanup_eventset(3) PAPIF_cleanup_eventset(3)	Remove all PAPI events from an event set.
PAPI_create_eventset(3) PAPIF_create_eventset(3)	Create a new, empty, PAPI event set.

…many more!

PAPI utilities

papi_avail()	Returns information regarding the availability of PAPI preset events.
papi_clockres()	Measures and reports clock latency and resolution for PAPI timers.
papi_command_line()	Execute PAPI preset or native events from the command line.
papi_component_avail()	Provides detailed information for PAPI native events.
papi_cost()	Computes execution time costs for basic PAPI operations.
papi_decode()	Decodes PAPI preset events into a csv format suitable for use with PAPI_encode_events.
papi_error_codes()	Lists all currently defined PAPI error codes.
papi_event_chooser()	Given a list of named events, lists other events that can also be counted at the same time.
papi_mem_info()	Returns information regarding the memory architecture of the current processor.
papi_multiplex_cost()	Computes the execution time costs for basic PAPI operations on multiplexed event sets.
papi_native_avail()	Returns detailed information for PAPI native events.

Example: Multiply two matrices

PAPI

```
#include "papi.h"

#define INDEX 100

static void test_fail(char *file, int line, char *call, int retval);


int main(int argc, char **argv) {
  extern void dummy(void *);
  float matrixa[INDEX][INDEX], matrixb[INDEX][INDEX], mresult[INDEX][INDEX];
  float real_time, proc_time, mflops;
  long long flpins;
  int retval;
  int i,j,k;


  /* Initialize the Matrix arrays */
  for ( i=0; i<INDEX*INDEX; i++ ){
    mresult[0][i] = 0.0;
    matrixa[0][i] = matrixb[0][i] = rand()*(float)1.1; }

  /* Setup PAPI library and begin collecting data from the counters */
  if((retval=PAPI_flops( &real_time, &proc_time, &flpins, &mflops))<PAPI_OK)
    test_fail(__FILE__, __LINE__, "PAPI_flops", retval);

  /* Matrix-Matrix multiply */
  for (i=0;i<INDEX;i++)
   for(j=0;j<INDEX;j++)
    for(k=0;k<INDEX;k++)
      mresult[i][j]=mresult[i][j] + matrixa[i][k]*matrixb[k][j];

  /* Collect the data into the variables passed in */
  if((retval=PAPI_flops( &real_time, &proc_time, &flpins, &mflops))<PAPI_OK)
    test_fail(__FILE__, __LINE__, "PAPI_flops", retval);

  printf("Real_time:\t%f\nProc_time:\t%f\nTotal flpins:\t%lld\nMFLOPS:\t\t%f\n",
  real_time, proc_time, flpins, mflops);
  printf("%s\tPASSED\n", __FILE__);
  PAPI_shutdown();
  exit(0);
}
```

How compile:

Standard cluster

```
UNIX> gcc -I/usr/local/include \
-OQ -o PAPI_flops PAPI_flops.c \
/usr/local/lib/libpapi.a
```

Cray system

```
$ cc -OQ -o PAPI_flops PAPI_flops.c
```

Darter example

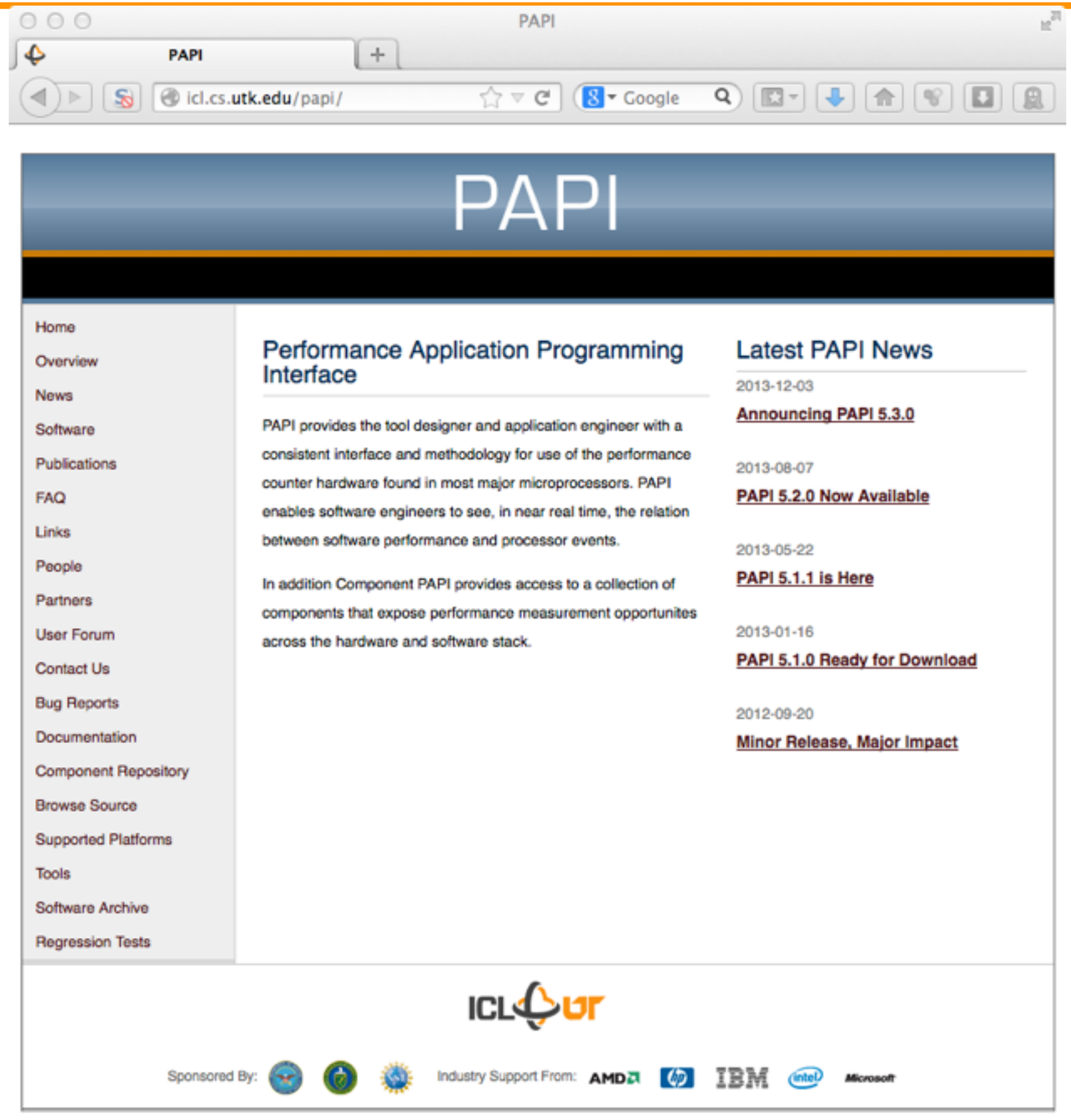
PAPI

```
lucio@darter1:~> cd $SCRATCHDIR/
lucio@darter1:/lustre/snx/lucio> wget http://web.eecs.utk.edu/~terpstra/using_papi/
PAPI_flops.c
lucio@darter1:/lustre/snx/lucio> perl -pi -e 's/PAPI_perror\(retval, errstring,
PAPI_MAX_STR_LEN \)/PAPI_perror\(errstring\)/g' PAPI_flops.c


lucio@darter1:/lustre/snx/lucio> module swap PrgEnv-cray PrgEnv-gnu
lucio@darter1:/lustre/snx/lucio> module load papi

lucio@darter1:/lustre/snx/lucio> cc -O0 -o PAPI_flops PAPI_flops.c

lucio@darter1:/lustre/snx/lucio> ./PAPI_flops
Real_time:  0.007060
Proc_time:  0.008593
Total flpins:   2028482
MFLOPS:     236.058762
PAPI_flops.c    PASSED
```

Example: Using Native events


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

main()
{
   int retval, EventSet = PAPI_NULL;
   unsigned int native = 0x0;
   PAPI_event_info_t info;
   /* 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);
   /* Find the first available native event */
   native = NATIVE_MASK | 0;
   if (PAPI_get_event_info(native, &info) !=
PAPI_OK) {
      if (PAPI_enum_event(&native, 0) != PAPI_OK)
         handle_error(1);
}
   /* Add it to the eventset */
   if (PAPI_add_event(EventSet, native) != PAPI_OK)
      handle_error(1);
}
```

References


PAPI
PAPI
icl.cs.utk.edu/papi/
Google
PAPI
Home
Overview
News
Software
Publications
FAQ
Links
People
Partners
User Forum
Contact Us
Bug Reports
Documentation
Component Repository
Browse Source
Supported Platforms
Tools
Software Archive
Regression Tests
Performance Application Programming Interface
PAPI provides the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events.
In addition Component PAPI provides access to a collection of components that expose performance measurement opportunites across the hardware and software stack.
Latest PAPI News
2013-12-03
Announcing PAPI 5.3.0
2013-08-07
PAPI 5.2.0 Now Available
2013-05-22
PAPI 5.1.1 is Here
2013-01-16
PAPI 5.1.0 Ready for Download
2012-09-20
Minor Release, Major Impact
ICL UT
Sponsored By:
Industry Support From:
AMD
IBM
intel
Microsoft
Mar 27 2014
Admin Login