Performance Measurement with PAPI

Dan Terpstra
<terpstra@cs.utk.edu>
April 17, 2002
CS 594-29622

Who are you?

PAPI
The Performance Application Programming Interface

Motivation
- To increase application performance and system throughput
- To characterize application and system workload on the CPU
- To stimulate performance tool development and research
- To stimulate research on more sophisticated feedback driven compilation techniques
## Goals

- Solid foundation for cross platform performance analysis tools
- Free tool developers from re-implementing counter access
- Standardization between vendors, academics and users
- Encourage vendors to provide hardware and OS support for counter access
- Reference implementations for a number of HPC architectures
- Well documented and easy to use

---

## Review of Modern High Performance Computing Architectures

### Modern HPC Architectures

- RISC or super-scalar architecture
  - Pipelined functional units
  - Multiple functional units in the CPU
  - Speculative execution
  - Several levels of cache memory
  - Cache lines shared between CPUs

### Pipelined Functional Units

- The circuitry on a chip that performs a given operation is called a **functional unit**.
- Most integer and floating point units are pipelined
  - Each stage of a pipelined unit working simultaneously on different sets of operands
  - After initial startup latency, goal is to generate one result every clock cycle

### Speculative Execution

- The CPU attempts to predict which way a branch will go and continues executing instructions speculatively along that path.
  - If the prediction is wrong, instructions executed down the incorrect path must be canceled.
  - On many processors, hardware counters keep counts of branch prediction hits and misses.

### Out of Order Execution

- CPU dynamically executes instructions as their operands become available, out of order if necessary
  - Any result generated out of order is temporary until all previous instructions have successfully completed.
  - Queues are used to select which instructions to issue dynamically to the execution units.
  - Relevant hardware counter metrics: instructions issued, instructions completed
**The Super-Scalar CPU (1)**

1) Predecode group of 4 instructions to I-cache
   - For each group:
     * Insert flow history bits
   - For each instruction:
     * Insert bits to tell what type it is: Branch, Exe. Unit, Mem, Ref
     * Insert branch history bits

**The Super-Scalar CPU (2)**

2) Fetch/Flow prediction
   - Guessing wrong is very costly with long pipelines
   - Branch prediction occurs late in the pipeline because we must wait for operands to be available
   - Bits point to the next group of 4 instructions to fetch. If wrong, update based on heuristic.

**The Super-Scalar Pipeline**

- Post-Risc Pipeline
- Instruction to color Buffer
- Issue
- Complete Instruction Buffer
- Fetch Decode
- Execute
- Memory
- Write back

**Instruction Predecoding**

- Four Instructions in Cache
- Instruction
- Pre-Decode
- Branch History
- Flow History

**The Super-Scalar CPU (3,4)**

3) Decode/Branch
   - Color instructions based on the branch path.
   - If the path is found to be in error, the results of all colored instructions will be discarded.

4) Instruction Dispatch/Reorder
   - Instructions are dispatched when functional unit and input/output registers become available.
   - Rename registers: Colored registers that hold results until the instructions are retired, at which point they’re committed.
The Super-Scalar CPU (5)

- 5) Execution Units
  - Some units like the Load/Store units may have a number of outstanding instructions executing at any given time.
  - Branch unit must communicate any resolved branches to all other units.

The Super-Scalar CPU (6)

- 6) Completed Instruction Buffer and Retire Unit
  - The CIB holds instructions that have been speculatively executed, including rename registers and flags.
  - Problem: We cannot retire instructions until we are sure all instructions before it have not caused an exception, therefore we retire in program order.

Hardware Counters

- Small set of registers that count events, which are occurrences of specific signals related to the processor’s function
- Monitoring these events facilitates correlation between the structure of the source/object code and the efficiency of the mapping of that code to the underlying architecture.

Instruction Counts and Functional Unit Status

- Relevant hardware counter data:
  - Total cycles
  - Total instructions
  - Floating point operations
  - Load/store instructions
  - Cycles functional units are idle
  - Cycles stalled
    - waiting for memory access
    - waiting for resource
  - Conditional branch instructions
    - executed
    - mispredicted

Cache and Memory Hierarchy

- Registers: On-chip circuitry used to hold operands and results of calculations
- L1 (primary) data cache: Small on-chip cache used to hold data about to be operated on
- L2 (secondary) cache: Larger (on- or off-chip) cache used to hold data and instructions retrieved from local memory.
- Some systems have L3 and even L4 caches.
Cache and Memory Hierarchy

- Local memory: Memory on the same node as the processor
- Remote memory: Memory on another node but accessible over an interconnect network.
- Each level of the memory hierarchy introduces approximately an order of magnitude more latency than the previous level.

Cache Structure

- Memory on a node is organized as an array of cache lines which are typically 4 or 8 words long. When a data item is fetched from a higher level cache or from local memory, an entire cache line is fetched.
- Caches can be either
  - direct mapped
  - N-way set associative
- A cache miss occurs when the program refers to a data item that is not present in the cache.

Cache Contention

- When two or more CPUs alternately and repeatedly update the same cache line
  - memory contention
    - when two or more CPUs update the same variable
    - correcting it involves an algorithm change
  - false sharing
    - when CPUs update distinct variables that occupy the same cache line
    - correcting it involves modification of data structure layout.
- Relevant hardware counter metrics
  - Cache misses and hit ratios
  - Cache line invalidations

TLB and Virtual Memory

- Memory is divided into pages.
- The operating system translates the virtual page addresses used by a program into physical addresses used by the hardware.
  - The most recently used addresses are cached in the translation lookaside buffer (TLB).
  - When the program refers to a virtual address that is not in the TLB, a TLB miss occurs.
- Relevant hardware counter metric: TLB misses

Memory Latencies

- CPU register: 0 cycles
- L1 cache hit: 2-3 cycles
- L1 cache miss satisfied by L2 cache hit: 8-12 cycles
- L2 cache miss satisfied from main memory, no TLB miss: 75-250 cycles
- TLB miss requiring only reload of the TLB: ~2000 cycles
- TLB miss requiring reload of virtual page – page fault: hundreds of millions of cycles

Steps of Optimization

- Optimize compiler switches
- Integrate libraries
- Profile
- Optimize blocks of code that dominate execution time by using hardware counter data to determine why the bottlenecks exist
- Always examine correctness at every stage!
**Example: MIPS R12K**

- MIPS R12K
  - 2 floating point units (1 multiply-add, 1 add)
  - 2 integer units
  - 2 load/store units

**Example: Intel Pentium**

- Intel Pentium Pro, II and III
  - In order x86 instructions decode to out of order uOPs
  - Up to 20 uOPs active at any given time
  - 2 ALUs, 2 LSUs, 1 FPU
  - 2 40 bit hardware counters, 1 64 bit cycle counter

**Example: Intel Pentium**

- Intel Pentium 4
  - In order x86 instructions decode to out of order uOPs
  - Up to 126 uOPs in flight
  - Up to 48 loads and 24 stores in flight
  - 3 ALUs, 2 LSUs, 2 FPU
  - 18 40 bit hardware counters, 1 64 bit cycle counter
Example: Intel Itanium
- Intel Itanium
  - EPIC (Explicitly Parallel Instruction Computing) design
  - 4 integer units
  - 4 multimedia units
  - 2 load/store units
  - 3 branch units
  - 2 extended precision floating point units
  - 2 single precision floating point units

Example: IBM POWER3
- IBM Power 3
  - 2 floating point units (multiply-add)
  - 3 fixed point units
  - 2 load/store units
  - 1 branch/dispatch unit

Example: IBM POWER4
- IBM Power 4
  - 2 floating point units (multiply-add)
  - 2 fixed point units
  - 2 load/store units
  - 1 branch execution unit
  - 1 conditional register unit
  - 2 processor cores per chip (!)
POWER4 Block Diagram

- Fabric Controller
- L1 Cache
- L2 Cache
- L3 Cache
- JTAG

POWER4 Processor Core

- L1 Cache
- L2 Cache
- L3 Cache
- Memory

POWER4 Memory Caching

- L1 ICache: 64KB / core; on-chip
- L1 DCache: 32KB / core; on-chip
- L2 Cache: 1.5 MB unified; on-chip
- L3 Cache: 32MB off-chip
- Memory: 0 – 16 GB

POWER4 Multi-Chip Module

- To other modules
- From other modules

POWER4 Multi-Module Node

General Design of PAPI
**PAPI Counter Interfaces**

- PAPI provides three interfaces to the underlying counter hardware:
  1. The low level interface manages hardware events in user defined groups called EventSets.
  2. The high level interface simply provides the ability to start, stop and read the counters for a specified list of events.
  3. Graphical tools to visualize information.

---

**PAPI Implementation**

- **Tools**
  - Portable Layer
    - PAPI Low Level
    - PAPI High Level
  - Machine Specific Layer
    - PAPI Machine Dependent Substrate
    - Kernel Extension
    - Operating System
    - Hardware Performance Counter

---

**PAPI 2.1 Release**

- Platforms
  - Linux/x86, Windows 2000
    - Requires patch to Linux kernel, driver for Windows
  - Linux/IA-64
  - Sun Solaris/Ultra 2.8
  - IBM AIX/Power3
    - Contact IBM for pmltoolkit
  - SGI IRIX/MIPS
    - Compaq Tru64/Alpha Ev6 & Ev67
    - Requires OS device driver from Compaq
    - Cray T3E/Unicos

---

**PAPI 2.1 Interfaces**

- C and Fortran bindings and Matlab wrappers (Windows only)

---

**PAPI Preset Events**

- Proposed standard set of events deemed most relevant for application performance tuning
- Defined in `papiStdEventDefs.h`
- Mapped to native events on a given platform
  - Run tests/avail to see list of PAPI preset events available on a platform

---

**Preset Events**

- PAPI supports 92 preset events and native events.
- Preset events are mappings from symbolic names to machine specific definitions for a particular hardware resource.
- Example: Total Cycles (in user mode) is `PAPI_TOT_CYC`
- PAPI also supports preset that may be derived from the underlying hardware metrics
  - Example: Floating Point Instructions per Second is `PAPI_FLOPS`
Preset Listing from tests/avail

Test case 8: Available events and hardware information.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Level</th>
<th>Count</th>
<th>Description (Note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>Yes</td>
<td>No</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x80000001</td>
<td>Yes</td>
<td>No</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>0x80000002</td>
<td>No</td>
<td>No</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>0x80000003</td>
<td>No</td>
<td>No</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L3_DCM</td>
<td>0x80000004</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 3 data cache misses</td>
</tr>
<tr>
<td>PAPI_L3_ICM</td>
<td>0x80000005</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 3 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L1_TCM</td>
<td>0x80000006</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 1 cache misses</td>
</tr>
<tr>
<td>PAPI_L2_TCM</td>
<td>0x80000007</td>
<td>Yes</td>
<td>No</td>
<td>Level 2 cache misses</td>
</tr>
<tr>
<td>PAPI_L3_TCM</td>
<td>0x80000008</td>
<td>No</td>
<td>No</td>
<td>Level 3 cache misses</td>
</tr>
<tr>
<td>PAPI_CA_SN</td>
<td>0x80000009</td>
<td>No</td>
<td>No</td>
<td>Requests for a snoo event</td>
</tr>
<tr>
<td>PAPI_CA_SH</td>
<td>0x8000000a</td>
<td>No</td>
<td>No</td>
<td>Requests for a shared cache line</td>
</tr>
<tr>
<td>PAPI_CA_CL</td>
<td>0x8000000b</td>
<td>No</td>
<td>No</td>
<td>Requests for a clean cache line</td>
</tr>
</tbody>
</table>
| PAPI_CA_INV| 0x8000000c | No    | No    | Requests for a cache line in.

PAPI High-Level Interface

High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- Not thread safe
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (additional code) than low-level

High-level API

- C interface
  - PAPI_start_counters
  - PAPI_read_counters
  - PAPI_stop_counters
  - PAPI_num_counters
  - PAPI_flops
- Fortran interface
  - PAPIF_start_counters
  - PAPIF_read_counters
  - PAPIF_stop_counters
  - PAPIF_num_counters
  - PAPIF_flops

High-level Interface Setup

- int PAPI_num_counters(void)
  - Initializes PAPI (if needed)
  - Returns number of hardware counters
- int PAPI_start_counters(int *events, int len)
  - Initializes PAPI (if needed)
  - Sets up an event set with the given counters
  - Starts counting in the event set
- int PAPI_library_init(int version)
  - Low-level routine implicitly called by above

Controlling the Counters

- PAPI_start_counters(long_long *vals, int alen)
  - Stop counters and put counter values in array
- PAPI_accum_counters(long_long *vals, int alen)
  - Accumulate counters into array and reset
- PAPI_read_counters(long_long *vals, int alen)
  - Copy counter values into array and reset counters
- PAPI_flops(float *time, float *ftime,
  long_long *flips, float *mflops)
  - Wallclock time, process time, FP ins since start,
  - MFlops/s since last call

Title goes here
PAPI_flops

- `int PAPI_flops(float *real_time, float *proc_time, long_long *flpins, float *mflops)`
  - Only two calls needed, PAPI_flops before and after the code you want to monitor
  - `real_time` is the wall-clocktime between the two calls
  - `proc_time` is the "virtual" time or time the process was actually executing between the two calls (not as fine grained as `real_time` but better for longer measurements)
  - `flpins` is the total floating point instructions executed between the two calls
  - `mflops` is the MFlop/s rating between the two calls
  - If `*flpins` == -1 the counters are reset

PAPI High-level Example

```c
long long values[NUM_EVENTS];
unsigned int
Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};
/* Start the counters */
PAPI_start_counters((int*)Events,NUM_EVENTS);
/* What we are monitoring… */
do_work();
/* Stop the counters and store the results in values */
retval = PAPI_stop_counters(values,NUM_EVENTS);
```

Return codes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_OK</td>
<td>No error</td>
</tr>
<tr>
<td>PAPI EINVAL</td>
<td>Invalid argument</td>
</tr>
<tr>
<td>PAPIENOMEM</td>
<td>Insufficient memory</td>
</tr>
<tr>
<td>PAPI EINVAL</td>
<td>A system/Library call failed. Check return variable</td>
</tr>
<tr>
<td>PAPI EBBSS</td>
<td>Subscript too small: E.g. unimplemented feature</td>
</tr>
<tr>
<td>PAPI EUCD</td>
<td>Access to the counter was lost or interrupted</td>
</tr>
<tr>
<td>PAPI EBUG</td>
<td>Internal error</td>
</tr>
<tr>
<td>PAPI EJNEVT</td>
<td>Hardware event does not exist</td>
</tr>
<tr>
<td>PAPI ECNMCT</td>
<td>Hardware event exists, but resources are exhausted</td>
</tr>
<tr>
<td>PAPI EJRUN</td>
<td>Event or event set is currently counting</td>
</tr>
<tr>
<td>PAPI EJRUN</td>
<td>Event or event set is currently running</td>
</tr>
<tr>
<td>PAPI EJNEVT</td>
<td>Event set is not available</td>
</tr>
<tr>
<td>PAPI EJNEVT</td>
<td>Argument is not a preset</td>
</tr>
<tr>
<td>PAPI EJNEVT</td>
<td>Hardware does not support events</td>
</tr>
<tr>
<td>PAPI EJNEVT</td>
<td>Any other error occurred</td>
</tr>
</tbody>
</table>

PAPI Low-Level Interface

Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- About 40 functions
- Obtain information about the executable and the hardware
- Thread-safe
- Fully programmable
- Multiplexing
- Callbacks on counter overflow
- Profiling

Low-level Functionality

- Library initialization
  PAPI_library_init, PAPI_thread_init, PAPI_multiplex_init, PAPI_shutdown
- Timing functions
  PAPI_get_real_usec, PAPI_get_virt_usec
  PAPI_get_real_cyc, PAPI_get_virt_cyc
- Inquiry functions
- Management functions
- Simple lock
  PAPI_lock/PAPI_unlock
How an OS Kernel may Handle Hardware Counters

- Problems:
  - The hardware only has a small number of bits.
  - The hardware can count USER and KERNEL modes.
  - The hardware needs to be accessed by multiple users at the same time.
  - Multiple processes, threads and CPUs.
- Solution: Modify the scheduler
  - Save and accumulate the counter hardware into 64 bit virtual registers when thread/process suspends or blocks.
  - Restore the counter control register and zero the counter value when thread/process resumes.
  - Read semantics: reading the hardware counters and add them to the 64 bit quantities handled by the kernel.

Event sets

- The event set contains key information
  - What low-level hardware counters to use
  - Most recently read counter values
  - The state of the event set (running/not running)
  - Option settings (e.g., domain, granularity, overflow, profiling)
- Event sets can overlap if they map to the same hardware counter set-up.
  - Allows inclusive/exclusive measurements

Simple Example

```c
#include "papi.h"
define NUM_EVENTS 2
int Events[NUM_EVENTS] = {PAPI_FP_INS, PAPI_DUP_CYC}, EventSet;
long long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);
/* Add Flips and total cycles to the eventset */
retval = PAPI_add_events(EventSet, Events, NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);
do_work(); /* What we want to monitor*/
/*Stop counters and store results in values */
retval = PAPI_stop(EventSet, values);
```

Creating an EventSet

```c
Event set: PAPI_STOPPED
```

The Cost of Calling PAPI

- Instrumentation introduces overhead
  - Reading hardware counters can be cheap
  - No overhead while counting is taking place
  - Must count the cost of setup and reading counters
  - These costs should be small compared to execution time for reasonable results
- Use `cost.exe` to measure latencies

Event set Operations

- Event set management:
  - PAPI_create_eventset,
  - PAPI_add_event[s], PAPI_rem_event[s],
  - PAPI_destroy_eventset
- Event set control:
  - PAPI_start, PAPI_stop, PAPI_read,
  - PAPI_accum
- Event set inquiry:
  - PAPI_query_event, PAPI_list_events,...
Adding events to an EventSet

```fortran
integer evset, status
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
```

1. PAPI_TOT_CYC

**evset state:** PAPI_STOPPED

---

Adding events to an EventSet

```fortran
integer evset, status
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
```

1. PAPI_TOT_CYC
2. PAPI_FP_INS

**evset state:** PAPI_STOPPED

---

Starting an EventSet

```fortran
integer evset, status
integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
```

C do 1000000 flops in 500000 cycles

call papif_read(evset, values, status)

C values contains the metrics in order of addition
C values(1) = 500000
C values(2) = 1000000

**evset state:** PAPI_RUNNING

---

Reading an EventSet

```fortran
integer evset, status
integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
call papif_read(evset, values, status)
```

C do 100000 flops in 500000 cycles

call papif_stop(evset, values, status)

C state can be either RUNNING or STOPPED
C to call reset

call papif_reset(evset, status)

1. PAPI_TOT_CYC
500000
2. PAPI_FP_INS
1000000

**evset state:** PAPI_STOPPED

---

Stopping an EventSet

```fortran
integer evset, status
integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
call papif_read(evset, values, status)
```

C do 1000000 flops in 500000 cycles

call papif_stop(evset, values, status)

C state can be either RUNNING or STOPPED
C to call reset

call papif_reset(evset, status)

1. PAPI_TOT_CYC
500000
2. PAPI_FP_INS
1000000

**evset state:** PAPI_STOPPED

---

Resetting an EventSet

```fortran
integer evset, status
integer*8 values(2)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
call papif_read(evset, values, status)
```

C do 100000 flops in 500000 cycles

call papif_stop(evset, values, status)

C state can be either RUNNING or STOPPED
C to call reset

call papif_reset(evset, status)

1. PAPI_TOT_CYC
0
2. PAPI_FP_INS
0

**evset state:** PAPI_STOPPED
Emptying an EventSet

```plaintext
integer evset, status
integer*8 values[2]
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
call papif_read(evset, values, status)
call papif_stop(evset, values, status)
call papif_reset(evset, status)
call papif_cleanup_eventset(evset, status)
```

Freeing an EventSet

```plaintext
integer evset, status
integer*8 values[2]
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
call papif_read(evset, values, status)
call papif_stop(evset, values, status)
call papif_reset(evset, status)
call papif_cleanup_eventset(evset, status)
call papif_destroy_eventset(evset, status)
```

Using PAPI with Threads

- After PAPI_library_init need to register unique thread identifier function
- For Pthreads
  ```plaintext
  retval=PAPI_thread_init(pthread_self, 0);
  ```
- OpenMP
  ```plaintext
  retval=PAPI_thread_init(omp_get_thread_num, 0);
  ```
- Each thread responsible for creation, start, stop and read of its own counters

Using PAPI with Multiplexing

- Multiplexing allows simultaneous use of more counters than are supported by the hardware.
- PAPI_multiplex_init() should be called after PAPI_library_init() to initialize multiplexing
- PAPI_set_multiplex(int *EventSet);
  - Used after the eventset is created to turn on multiplexing for that eventset
- Then use PAPI normally

Issues with Multiplexing

- Some platforms support hardware multiplexing.
- On those that don’t, PAPI implements multiplexing in software.
- The more events you multiplex, the larger the sampling error in the result.

Multiplex Code Examples

From the PAPI source distribution:
```plaintext
tests/multiplex1.c
tests/multiplex1_pthreads.c
```
Native Events

- An event countable by the CPU can be counted even if there is no matching preset PAPI event.
- Same interface as when setting up a preset event, but a CPU-specific bit pattern is used instead of the PAPI event definition.

Native Event Examples

From the PAPI source distribution:
- tests/native.c
- ftests/native.F

Callbacks on Counter Overflow

- PAPI provides the ability to call user-defined handlers when a specified event exceeds a specified threshold.
- For systems that do not support counter overflow at the OS level, PAPI sets up a high resolution interval timer and installs a timer interrupt handler.

Counter Overflow and Statistical Profiling

PAPI_overflow

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

- Sets up an EventSet such that when it is PAPI_start’d, it begins to register overflows.
- The EventSet may contain multiple events, but only one may be an overflow trigger.

Overflow Code Examples

From the PAPI source distribution:
- tests/overflow.c
- tests/overflow_pthreads.c
Statistical Profiling

- PAPI provides support for SVR4-compatible execution profiling based on any counter event.
- PAPI_profil() creates a histogram of overflow counts for a specified region of the application code.

PAPI_profil

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

- `buf` – buffer of `bufsz` bytes in which the histogram counts are stored
- `offset` – start address of the region to be profiled
- `scale` – contraction factor that indicates how much smaller the histogram buffer is than the region to be profiled

Profiling Code Examples

From the PAPI source distribution:

- tests/profile.c
- tests/sprofile.c
- tests/profile_pthreads.c

Tools that use PAPI

Perfometer

- Application is instrumented with PAPI
  - call mark_perfometer(Color)
- Application is started. At the call to perfometer, signal handler and timer are set to collect and send the information to a Java applet containing the graphical view.
- Sections of code that are of interest can be designated with specific colors
  - Using a call to set_perfometer("color")
- Real-time display or trace file
Dynaprof

- Application is instrumented with PAPI
  - call performeter()’
  - call mark_performeter(Color)
- Application is started. At the call to performeter, signal handler and timer are set to collect and send the information to a Java applet containing the graphical view.
- Sections of code that are of interest can be designated with specific colors
  - Using a call to set_performeter('color')
- Real-time display or trace file
DynaProf Demo

Other Tools that use PAPI
- DEEP/PAPI (Pacific Sierra) http://www.pavi.com/deep_papi.htm
- TAU (Allen Mallory, U of Oregon) http://www.cs.uoregon.edu/research/parcomp/tau
- SvPablo (Dan Reed, U of Illinois) http://vixra.cs.uiuc.edu/software/SvPablo/SvPablo.html
- Vprof (Curtis Jansen, Sandia Livermore Lab) http://lanl.ca.sandia.gov/vprof/vprof.html
- Cluster Tools (Al Geist, ORNL)

For More Information
  - Software and documentation
  - Reference materials
  - Papers and presentations
  - Third-party tools
  - Mailing lists

Homework Assignment
An important step in porting PAPI to a new architecture is the creation of a preset map that maps PAPI events to native events on a specific platform. Creating such a map requires an understanding of the PAPI standard events and an understanding of the events that can be measured on the target system. Your assignment is to work, either alone or with your other team members (classmates), to create a preset map for the IBM POWER4, and deliver a copy of that effort to your supervisor (me) by the specified deadline.

If you work with others, please indicate who contributed to the definitions of which preset events.

Homework Resources
Because the POWER4 is a new architecture, the documentation is limited and often difficult to locate. So far, the only information you have found is at cs.utk.edu/~terpstru/power4, but other relevant information may be available on the PAPI web site or elsewhere on the web.