PAPI - Performance API

ANDRÉ PEREIRA
ampereira@di.uminho.pt
Motivation

- Application and functions execution time is easy to measure
  - time
  - gprof
  - valgrind (callgrind)
  - ...

- It is enough to identify bottlenecks, but...
  - Why is it slow?
  - How does the code behaves?
Motivation

- Efficient algorithms should take into account
  - Cache behaviour
  - Memory and resource contention
  - Floating point efficiency
  - Branch behaviour
HW Performance Counters

- Hardware designers added specialised registers to measure various aspects of a microprocessor.

- Generally, they provide an insight into:
  - Timings
  - Cache and branch behaviour
  - Memory access patterns
  - Pipeline behaviour
  - FP performance
  - IPC
  - ...

What is PAPI?

- Interface to interact with performance counters
  - With minimal overhead
  - Portable across several platforms
- Provides utility tools, C, and Fortran API
  - Platform and counters information
PAPI Organisation

3rd Party Tools

Low Level API

High Level API

PAPI Portable Layer

PAPI Hardware Specific Layer

Kernel Extension

Operating System

Perf Counter Hardware
Supported Platforms

- Mainstream platforms (Linux)
  - x86, x86_64 Intel and AMD
  - ARM, MIPS
  - Intel Itanium II
  - IBM PowerPC
Utilities

- papi_avail
Utilities

- papi_avail
- papi_native_avail
Utilities

- papi_avail
- papi_native_avail
- papi_event_chooser
PAPI Performance Counters

- Preset events
  - Events implemented on all platforms
    - PAPI_TOT_INS

- Native events
  - Platform dependent events
    - L3_CACHE_MISS

- Derived events
  - Preset events that are derived from multiple native events
    - PAPI_L1_TCM may be L1 data misses + L1 instruction misses
PAPI High-level Interface

- Calls the low-level API
- Easier to use
- Enough for coarse grain measurements
  - You will not optimise code based on the amount of L2 TLB flushes per thread…
- For preset events only!
The Basics

- PAPI_start_counters
- PAPI_stop_counters
The Basics

#include "papi.h"
#define NUM EVENTS 2
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 counters and store results in values */
retval = PAPI_stop_counters(values,NUM EVENTS);
PAPI Low-level Interface

- Increased efficiency and functionality
- More information about the environment
- Concepts to check
  - EventSet
  - Multiplexing
The Basics

- **Initialised**
  - library_init
  - create_eventset

- **EventSet Setup**
  - add_event

- **Counter Setup**
  - add_event

- **Measurement Storage**
  - stop

- **Measuring**
  - start
  - start

**PREFIX: PAPI_**

André Pereira, UMinho, 2015/2016
The Basics

#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC};
int 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 Flops and total cycles to the eventset */
retval = PAPI_add_events(EventSet,Events,NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);
/* What we want to monitor*/
do_work();
/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
PAPI CUDA Component

- PAPI is also available for CUDA GPUs
- Uses the CUPTI
  - Which counters can be directly accessed
  - Define a file with the counters and an environment variable
- Gives useful information about the GPU usage
  - IPC
  - Memory load/stores/throughput
  - Branch divergences
  - SM(X) occupancy
  - …
What to Measure?

- The whole application?
- PAPI usefulness is limited when used alone
  - Combine it with other profilers
  - Bottleneck identification + characterisation
A Practical Example

\begin{verbatim}
for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++)
        for (int k = 0; k < SIZE; k++)
            c[i][j] += a[i][k] * b[k][j];
\end{verbatim}
A Practical Example

```c
int sum;

for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++) {
        sum = 0;
        for (int k = 0; k < SIZE; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
```
Execution Time

@ 2x Intel Xeon E5-2695v2, 12C with 24t each, 2.4GHz
FLOP's

@ 2x Intel Xeon E5-2695v2, 12C with 24t each, 2.4GHz
Cache Miss Rate

L2 Miss Rate

L3 Miss Rate

@ 2x Intel Xeon E5-2695v2, 12C with 24t each, 2.4GHz
Arithmetic Intensity

@ 2x Intel Xeon E5-2695v2, 12C with 24t each, 2.4GHz
Useful Counters

- **Instruction mix**
  - PAPI_FP_INS
  - PAPI_SR/LD_INS
  - PAPI_BR_INS
  - PAPI_SP/DP_VEC

![Pie chart showing instruction mix percentages]

- 32% Float point
- 19% Stores
- 19% Branches
- 18% Loads
- 12% Integer
Useful Counters

- Instruction mix
  - PAPI_FP_INS
  - PAPI_SR/LD_INS
  - PAPI_BR_INS
  - PAPI_SP/DP_VEC

- FLOPS and operational intensity
  - PAPI_FP_OPS
  - PAPI_SP/DP_OPS
  - PAPI_TOT_INS

- Cache behaviour and bytes transferred
  - PAPI_L1/2/3_TCM
  - PAPI_L1_TCA
Useful Hints

- Be careful choosing a measurement heuristic

- Automatise the measurement process
  - With scripting/C++ coding
  - Using 3rd party tools that resort to PAPI
    - PerfSuite
    - HPCToolkit
    - TAU

- Available for Java and on virtual machines
Compiling and Running the Code

- Use the same GCC/G++ version as
  - The PAPI compilation on your home (preferably)
  - The PAPI available at the cluster

- Code compilation
  ```
g++ -L$PAPI_DIR/lib -l$PAPI_DIR/include c.cpp -lpapi
  ```

- Code execution
  ```
  export LD_LIBRARY_PATH=$PAPI_DIR/lib: $LD_LIBRARY_PATH
  ```
  (dynamic library dependencies are resolved at runtime; you can have it on your .bashrc)

  - Run the code!
Hands-on

- Assess the available counters
- Perform the FLOPs and miss rate measurements
  - https://bitbucket.org/ampereira/papi/downloads
References


