How’re you doing?
Basic Performance Measurement with PAPI and Opteron

Dan Terpstra
ICL Lunch
07.11.08
Table of Contents

♦ PAPI
♦ Performance Measurement Categories*
♦ The Code*
♦ Performance Data for Opteron
♦ Future Directions

What’s PAPI?

- Middleware to provide a consistent programming interface for the performance counter hardware found in most major micro-processors.

- Countable events are defined in two ways:
  - platform-neutral Preset Events
  - Platform-dependent Native Events

- Presets can be derived from multiple Native Events

- All events are referenced by name and collected in EventSets for sampling

- Events can be multiplexed if counters are limited

- Statistical sampling implemented by:
  - Hardware overflow if supported by the platform
  - Software overflow with timer driven sampling
Where’s PAPI

- PAPI runs on most modern processors and Operating Systems of interest to HPC:
  - IBM POWER series / AIX
  - POWER4,5, PowerPC / Linux
  - Blue Gene/L/P
  - Intel Pentium series, Core2 / Linux
  - Intel Itanium 1, 2, Montecito
  - AMD Athlon, Opteron, Barcelona / Linux
  - Cray X1, X2, XT3/4 / Catamount / CNL
  - Altix, Sparc, SiCortex …
PAPI Counter Interfaces

PAPI provides 3 interfaces to the underlying counter hardware:

1. A Low Level API manages hardware events in user defined groups called EventSets, and provides access to advanced features.

2. A High Level API provides the ability to start, stop and read the counters for a specified list of events.

3. Graphical and end-user tools provide facile data collection and visualization.
PAPI High Level Calls

1. `PAPI_num_counters` - get the number of hardware counters available on the system
2. `PAPI_flips` - simplified call to get Mflips/s (floating point instruction rate), real and processor time
3. `PAPI_flops` - simplified call to get Mflops/s (floating point operation rate), real and processor time
4. `PAPI_ipc` - gets instructions per cycle, real and processor time
5. `PAPI_accum_counters` - add current counts to array and reset counters
6. `PAPI_read_counters` - copy current counts to array and reset counters
7. `PAPI_start_counters` - start counting hardware events
8. `PAPI_stop_counters` - stop counters and return current counts
PAPI Preset Events

♦ Preset Events
  ➢ Standard set of over 100 events for application performance tuning
  ➢ No standardization of the exact definition
  ➢ Mapped to either single or linear combinations of native events on each platform
  ➢ Use papi_avail utility to see what preset events are available on a given platform

Level 3 Cache

PAPI_L3_DCH: Level 1 data cache hits
PAPI_L3_DCA: Level 1 data cache accesses
PAPI_L3_DCR: Level 1 data cache reads
PAPI_L3_DCW: Level 1 data cache writes
PAPI_L3_DCM: Level 1 data cache misses

PAPI_L3_ICH: Level 1 instruction cache hits
PAPI_L3_ICA: Level 1 instruction cache accesses
PAPI_L3_ICR: Level 1 instruction cache reads
PAPI_L3_ICW: Level 1 instruction cache writes
PAPI_L3_ICM: Level 1 instruction cache misses

PAPI_L3_TCH: Level 1 total cache hits
PAPI_L3_TCA: Level 1 total cache accesses
PAPI_L3_TCR: Level 1 total cache reads
PAPI_L3_TCW: Level 1 total cache writes
PAPI_L3_TCM: Level 1 cache misses

PAPI_L3_LDM: Level 1 load misses
PAPI_L3_STM: Level 1 store misses

Cache Sharing

PAPI_CA_SNP: Requests for a snoop
PAPI_CA_SHR: Requests for exclusive access to shared cache line
PAPI_CA_CLN: Requests for exclusive access to clean cache line
PAPI_CA_INV: Requests for cache line invalidation
PAPI_CA_ITV: Requests for cache line intervention
PAPI Native Events

- Native Events
  - Any event countable by the CPU
  - Same interface as for preset events
  - Use `papi_native_avail` utility to see all available native events

- Use `papi_eventchooser` utility to select a compatible set of events

PRESET,
PAPI_L1_DCH,
DERIVED_SUB,
DATA_CACHE_ACCESSES,
DATA_CACHE_MISSES
Performance Measurement Categories

♦ Efficiency
  ➢ Instructions per cycle (IPC)
  ➢ Memory bandwidth

♦ Caches
  ➢ Data cache misses and miss ratio
  ➢ Instruction cache misses and miss ratio

♦ Lower level cache misses and miss ratio

♦ Translation lookaside buffers (TLB)
  ➢ Data TLB misses and miss ratio
  ➢ Instruction TLB misses and miss ratio

♦ Control transfers
  ➢ Branch mispredictions
  ➢ Near return mispredictions

♦ Special cases
  ➢ Unaligned data access
  ➢ Floating point operations
  ➢ Floating point exceptions
#include <stdio.h>

#include <stdlib.h>

#define ROWS 1000

#define COLUMNS 1000

int main() {
    int i, j, k;
    float sum;
    float matrix_a[ROWS][COLUMNS];
    float matrix_b[ROWS][COLUMNS];
    float matrix_c[ROWS][COLUMNS];

    // Matrix multiplication
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLUMNS; j++) {
            sum = 0.0;
            for (k = 0; k < COLUMNS; k++) {
                sum += matrix_a[i][k] * matrix_b[k][j];
            }
            matrix_c[i][j] = sum;
        }
    }

    return 0;
}

void classic_matmul() {
    // Multiply the two matrices
    int i, j, k;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLUMNS; j++) {
            float sum = 0.0;
            for (k = 0; k < COLUMNS; k++) {
                sum +=
                matrix_a[i][k] * matrix_b[k][j];
            }
            matrix_c[i][j] = sum;
        }
    }
}

void interchanged_matmul() {
    // Multiply the two matrices
    int i, j, k;
    for (i = 0; i < ROWS; i++) {
        for (k = 0; k < COLUMNS; k++) {
            for (j = 0; j < COLUMNS; j++) {
                matrix_c[i][j] +=
                matrix_a[i][k] * matrix_b[k][j];
            }
        }
    }
    // Note that the nesting of the innermost
    // loops has been changed. The index variables j
    // and k change the most frequently and the access
    // pattern through the operand matrices is sequential
    // using a small stride (one.) This change improves
    // access to memory data through the data cache.
    // Data translation lookaside buffer (DTLB) behavior
    // is also improved.
Performance Data
# IPC – instructions per cycle

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAPI_IPC Test (PAPI_ipc)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real time</td>
<td>13.6093 sec</td>
<td>2.9796 sec</td>
</tr>
<tr>
<td>Processor time</td>
<td>13.5359 sec</td>
<td>2.9556 sec</td>
</tr>
<tr>
<td>IPC</td>
<td>0.3697</td>
<td>1.6936</td>
</tr>
<tr>
<td>Instructions</td>
<td>9007035063</td>
<td>9009011383</td>
</tr>
<tr>
<td><strong>High Level IPC Test (PAPI_{start,stop}_counters)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real time</td>
<td>13.6106 sec</td>
<td>2.9762 sec</td>
</tr>
<tr>
<td>IPC</td>
<td>0.3697</td>
<td>1.6939</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>24362605525</td>
<td>5318626915</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>9007034503</td>
<td>9009011245</td>
</tr>
<tr>
<td><strong>Low Level IPC Test (PAPI low level calls)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real time</td>
<td>13.6113 sec</td>
<td>2.9772 sec</td>
</tr>
<tr>
<td>IPC</td>
<td>0.3697</td>
<td>1.6933</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>24362750167</td>
<td>5320395138</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>9007034381</td>
<td>9009011130</td>
</tr>
</tbody>
</table>

- Measures instruction level parallelism
- All three PAPI methods consistent
- Roughly 460% improvement in reordered code
## Memory Bandwidth

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SYSTEM_READ_RESPONSES</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:EXCLUSIVE:MODIFIED:SHARED</td>
<td>62081741</td>
<td>61831162</td>
</tr>
<tr>
<td>QUADWORDS_WRITTEN_TO_SYSTEM:ALL</td>
<td>1857263</td>
<td>1036769</td>
</tr>
<tr>
<td>DRAM_ACCESS_PAGE:HIT:MISS:CONFLICT</td>
<td>45785165</td>
<td>33645477</td>
</tr>
<tr>
<td>Read Bandwidth</td>
<td>293.3849 MB/sec</td>
<td>1338.5084 MB/sec</td>
</tr>
<tr>
<td>Write Bandwidth</td>
<td>8.7770 MB/sec</td>
<td>22.4438 MB/sec</td>
</tr>
<tr>
<td>DRAM Bandwidth</td>
<td>216.3708 MB/sec</td>
<td>728.3504 MB/sec</td>
</tr>
</tbody>
</table>

- System Clock: 1.8 GHz
- Memory BlockSize assumed to be 64 bytes
- Similar number of reads from memory in both codes
- Dramatically improved Read Bandwidth
- Significantly improved Write Bandwidth
Data Cache Accesses can be considered in 3 categories:

- **Compulsory**: Occur on first reference to a data item.
  - Prefetching

- **Capacity**: Occur when the working set exceeds the cache capacity.
  - Smaller working set (blocking/tiling algorithms)

- **Conflict**: Occur when a data item is referenced after the cache line containing the item was evicted earlier.
  - Data layout; memory access patterns
L1 Data Cache Access

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_CACHE_ACCESSES</td>
<td>2002807841</td>
<td>3008528961</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS:L2_MODIFIED:L2_OWNED:L2_EXCLUSIVE:L2_SHARED</td>
<td>205968263</td>
<td>60716301</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS_FROM_SYSTEM:MODIFIED:OWNED:EXCLUSIVE:SHARED</td>
<td>61970925</td>
<td>1950282</td>
</tr>
<tr>
<td>PAPI_L1_DCA</td>
<td>2002808034</td>
<td>3008528895</td>
</tr>
<tr>
<td>PAPI_L1_DCM</td>
<td>268010587</td>
<td>62680818</td>
</tr>
</tbody>
</table>

- Data Cache Request Rate: 0.2224 req/inst (Classic), 0.3339 req/inst (Reordered)
- Data Cache Miss Rate: 0.0298 miss/inst (Classic), 0.0070 miss/inst (Reordered)
- Data Cache Miss Ratio: 0.1338 miss/req (Classic), 0.0208 miss/req (Reordered)

- Two techniques
  - First uses native events
  - Second uses PAPI presets only
- ~50% more requests from improved code
- 1/4 as many misses per instruction
- 1/6 as many misses per request
# L1 Instruction Cache Access

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_ICR</td>
<td>3014322225</td>
<td>3014205662</td>
</tr>
<tr>
<td>INSTRUCTION_CACHE_REFILLS_FROM_L2</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>INSTRUCTION_CACHE_REFILLS_FROM_SYSTEM</td>
<td>73</td>
<td>36</td>
</tr>
<tr>
<td>PAPI_L1_ICR</td>
<td>3014322033</td>
<td>3014205070</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>60</td>
<td>44</td>
</tr>
</tbody>
</table>

- Instr Cache Request Rate: 0.3347 req/inst, 0.3346 req/inst
- Instr Cache Miss Rate: 0.0000 miss/inst, 0.0000 miss/inst
- Instr Cache Miss Ratio: 0.0000 miss/req, 0.0000 miss/req

- **Two techniques, like Data Cache case**
  - First uses native events
  - Second uses PAPI presets only
- **Small subroutines fit completely in cache**
- **Virtually no misses; pretty boring**
# L2 Cache Access

## Measurement

<table>
<thead>
<tr>
<th></th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------------</td>
<td>-----------------</td>
<td>-------------------</td>
</tr>
</tbody>
</table>

### Direct L2 Cache Test

<table>
<thead>
<tr>
<th>REQUESTS_TO_L2:INSTRUCTIONS:DATA:TLB_WALK</th>
<th>1057556622</th>
<th>70996294</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2_CACHE_MISS:INSTRUCTIONS:DATA:TLB_WALK</td>
<td>62120093</td>
<td>4167947</td>
</tr>
<tr>
<td>L2_FILL_WRITEBACK:ALL</td>
<td>268201418</td>
<td>122740586</td>
</tr>
</tbody>
</table>

- **L2 Cache Request Rate**: 0.1472 req/inst, 0.0215 req/inst
- **L2 Cache Miss Rate**: 0.0069 miss/inst, 0.0005 miss/inst
- **L2 Cache Miss Ratio**: 0.0469 miss/req, 0.0215 miss/req

### Indirect L2 Cache Test

| INSTRUCTION_CACHE_REFILLS_FROM_L2       | 4          | 0        |
| INSTRUCTION_CACHE_REFILLS_FROM_SYSTEM   | 30         | 9        |
| L2_CACHE_MISS:TLB_WALK                  | 260        | 5438     |
| REQUESTS_TO_L2:TLB_WALK                 | 78763271   | 803242   |
| DATA_CACHE_REFILLS:L2_SHARED:L2_EXCLUSIVE:L2_OWNED:L2_MODIFIED | 205977083 | 60715886 |
| DATA_CACHE_REFILLS_FROM_SYSTEM:SHARED:EXCLUSIVE:OWNED:MODIFIED | 61973057 | 1950318  |

- **L2 Cache Request Rate**: 0.1172 req/inst, 0.0070 req/inst
- **L2 Cache Miss Rate**: 0.0069 miss/inst, 0.0002 miss/inst
- **L2 Cache Miss Ratio**: 0.0587 miss/req, 0.0308 miss/req
## L2 Cache Access

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 Cache Request Rate</td>
<td>0.1172 req/inst</td>
<td>0.0070 req/inst</td>
</tr>
<tr>
<td>L2 Cache Miss Rate</td>
<td>0.0069 miss/inst</td>
<td>0.0002 miss/inst</td>
</tr>
<tr>
<td>L2 Cache Miss Ratio</td>
<td>0.0587 miss/req</td>
<td>0.0308 miss/req</td>
</tr>
</tbody>
</table>

- **L2 cache is unified on Opteron**
- **Two techniques:**
  - First is coarser grained
  - Second provides more detail but requires 7 events (two passes)
  - No major differences for this code
- **L2 requests and misses down dramatically in reordered code**
  - Recall, memory accesses are up by 50%
- **Almost all (98+%) L2 access are for data in reordered code**
## DTLB Access

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCA</td>
<td>2002809207</td>
<td>3008530341</td>
</tr>
<tr>
<td>L1_DTLB_MISS_AND_L2_DTLB_HIT:ALL</td>
<td>296943120</td>
<td>350824</td>
</tr>
<tr>
<td>L1_DTLB_AND_L2_DTLB_MISS:ALL</td>
<td>783208861</td>
<td>785470</td>
</tr>
</tbody>
</table>

- **L1 DTLB Request Rate**: 0.2224 req/inst vs. 0.3339 req/inst
- **L1 DTLB Miss Rate**: 0.1199 miss/inst vs. 0.0001 miss/inst
- **L1 DTLB Miss Ratio**: 0.5393 miss/req vs. 0.0004 miss/req

- **L2 DTLB Request Rate**: 0.1199 req/inst vs. 0.0001 req/inst
- **L2 DTLB Miss Rate**: 0.0870 miss/inst vs. 0.0001 miss/inst
- **L2 DTLB Miss Ratio**: 0.7251 miss/req vs. 0.6913 miss/req

- Goto and van de Geijn claim TLB misses can limit fast matrix multiply ("On Reducing TLB Misses in Matrix Multiplication")
- L1 Data Cache Access == DTLB Access
- More L1 requests in improved code
- Dramatically fewer misses
**ITLB Access**

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_ICR</td>
<td>3014320811</td>
<td>3014204576</td>
</tr>
<tr>
<td>L1_ITLB_MISS_AND_L2_ITLB_HIT</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>L1_ITLB_MISS_AND_L2_ITLB_MISS:ALL</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>L1 ITLB Request Rate</td>
<td>0.3347 req/inst</td>
<td>0.3346 req/inst</td>
</tr>
<tr>
<td>L1 ITLB Miss Rate</td>
<td>0.0000 miss/inst</td>
<td>0.0000 miss/inst</td>
</tr>
<tr>
<td>L1 ITLB Miss Ratio</td>
<td>0.0000 miss/req</td>
<td>0.0000 miss/req</td>
</tr>
<tr>
<td>L2 ITLB Request Rate</td>
<td>0.0000 req/inst</td>
<td>0.0000 req/inst</td>
</tr>
<tr>
<td>L2 ITLB Miss Rate</td>
<td>0.0000 miss/inst</td>
<td>0.0000 miss/inst</td>
</tr>
<tr>
<td>L2 ITLB Miss Ratio</td>
<td>0.6923 miss/req</td>
<td>0.8571 miss/req</td>
</tr>
</tbody>
</table>

- See DTLB...
- L1 Instruction Cache Reads == ITLB Access
- Boring...but useful in identifying code layout problems
# Branching

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_BR_INS</td>
<td>1001028240</td>
<td>1001006987</td>
</tr>
<tr>
<td>PAPI_BR_MSP</td>
<td>1028256</td>
<td>1006984</td>
</tr>
<tr>
<td>PAPI_BR_TKN</td>
<td>1000027233</td>
<td>1000005980</td>
</tr>
</tbody>
</table>

- Branch Rate: 0.1111 br/inst 0.1111 br/inst
- Branch Miss Rate: 0.0001 miss/inst 0.0001 miss/inst
- Branch Miss Ratio: 0.0010 miss/br 0.0010 miss/br
- Branch Taken Rate: 0.1110 tkn/inst 0.1110 tkn/inst
- Branch Taken Ratio: 0.9990 tkn/br 0.9990 tkn/br
- Instr / Branch: 8.9978 inst/br 8.9999 inst/br

- Uses all PAPI Presets!
- Branch behavior nearly identical in both codes
- Roughly 1 branch every 9 instructions
- 1 miss per 1000 branches (remember ROWS?)
- Branching and branch misses can be reduced with loop unrolling, loop fusion and function in-lining.
Future Directions

♦ Good programmers are lazy
♦ Native event names can be long:
  ➢ DATA_CACHE_REFILLS_FROM_SYSTEM:MODIFIED :OWNED:EXCLUSIVE:SHARED
♦ Derived metrics can be complicated:
  ➢ L2 DTLB miss ratio = DTLB_L1M_L2M / (DTLB_L1M_L2H + DTLB_L1M_L2M)
♦ PAPI 3.6 reads PRESET events from a text file:
  ➢ PRESET, PAPI_L1_TCH, DERIVED_POSTFIX, N0|N1|+|N2|--N3|--, DATA_CACHE_ACCESSES, INSTRUCTION_CACHE_FETCHES, DATA_CACHE_MISSES, INSTRUCTION_CACHE_MISSES
♦ The Solution is...
User Defined Events

♦ Extend the High Level Interface
  - Read a user file at init time:
    - USER, L2_DTLB_miss_ratio, DERIVED_POSTFIX, |N0|N1|N2|+|/|, DTLB_L1M_L2M, DTLB_L1M_L2H, DTLB_L1M_L2M

♦ Users can then:
  - Create nicknames for favorite native events
  - Experiment with alternative event definitions
  - Extend the PAPI presets with new definitions
  - Create meta-event metrics derived from available Preset or Native events
  - Nested definitions?
Goodbye…

...and thanks for all the pizza...

-apologies to Douglas Adams
What’s a Native Event?

8 mask bits: 256 events

16 mask bits: 64 events

PMD: AMD Athlon, Opteron

PMC: Intel Pentium II, III, M, Core; AMD Athlon, Opteron

PMC: Pentium 4

Reserved

Tag Enable
OS
USR

PMC: Pentium 4