CS 594: SCIENTIFIC COMPUTING FOR ENGINEERS

PERFORMANCE ANALYSIS TOOLS

Vince Weaver
vweaver1@eecs.utk.edu

With slides from Heike Jagode, Karl Fuerlinger, Andreas Knuepfer, Shirley Moore, Sameer Shende, Felix Wolf and others.
Rough Outline

1. **Motivation**
   - What is Performance?
   - Why Performance Analysis?

1. **Concepts and Definitions**
   - The performance analysis cycle
   - Measurement: profiling vs. tracing
   - Analysis: manual vs. automated

1. **Performance Analysis Tools**
   - **PAPI**: Access to hardware performance counters
   - **Vampir Suite**: Instrumentation and Trace visualization
   - **KOJAK / Scalasca**: automatic performance analysis tool
   - **TAU**: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
WHAT IS PERFORMANCE?
WHAT IS PERFORMANCE?

- Simply, getting a job done?
- Producing expected results?
- Speed?
- Budget?
- Hardware Usage?
- Power consumption?
Why Performance Analysis? 1/2

- Domenico Ferrari (1986):
  “The study of performance evaluation as an independent subject has sometimes caused researchers in the area to lose contact with reality.”

- Why is performance analysis not an integrated and natural part of software development?
  - The primary duty of software developers is to create functionally correct programs!
  - Performance evaluation tends to be optional
    Some people compare it to the freestyle event in ice-skating
  - Raj Jain (1991)
    “Contrary to common belief, performance evaluation is an art. ... Like artist, each analyst has a unique style. Given the sample problem, two analysts may choose different performance metrics and evaluation methodologies.” ... but even they need tools!
Performance Analysis is important:

- Large investments in HPC systems
  - Procurement costs: ~$40 Mio
  - Operational costs: ~$5 Mio per year
  - Electricity costs: 1 MW / year ~$1 Mio

- Efficient usage is important because of expensive and limited resources
- Scalability is important to achieve next bigger simulation

- Performance analysis: **Get highest performance for a given cost**
- „Performance Analyst“: Anyone who is associated with computer systems,
  - i.e. system engineers, computer scientists, application developers and of course users
“More computing sins are committed in the name of efficiency (without
necessarily achieving it) than for any other single reason - including blind
stupidity.” - W.A. Wulf

“We should forget about small efficiencies, say about 97% of the time:
**premature optimization is the root of all evil.** Yet we should not pass up
our opportunities in that critical 3%. A good programmer will not be lulled into
complacency by such reasoning, he will be wise to look carefully at the
critical code; but only after that code has been identified”[6] - Donald Knuth

“The First Rule of Program Optimization: Don't do it. The Second Rule of
Program Optimization (for experts only!): Don't do it yet.” - Michael A.
Jackson
Amdahl's Law
Performance Optimization cycle:

Measure & Analyze:
- Have an optimization phase
- just like testing & debugging phase
- Do profiling and tracing
- Use tools!
- avoid do-it-yourself `printf` solutions
Profiling

- Aggregate performance metrics
- Number of times a routine was invoked
- Exclusive, inclusive time/counts spent executing
- Number of instrumented child routines invoked, etc.
- Structure of invocations (call-trees/call-graphs)
- Memory, message communication sizes

Tracing

- When and where events took place along a global timeline
- Time-stamped events (points of interest)
- Message communication events (sends/receives) are tracked
- Shows when and from/to where messages were sent
- Event Trace: collection of all events of a process/program sorted by time
MEASUREMENT: PROFILING

• Recording of summary information during execution
  ◦ inclusive, exclusive time, # calls, hardware counter statistics, ...
• Reflects performance behavior of program entities
  ◦ functions, loops, basic blocks
  ◦ user-defined “semantic” entities
• Very good for low-cost performance assessment
• Helps to expose performance bottlenecks and hotspots
• Implemented through either
  ◦ sampling: periodic OS interrupts or hardware counter traps
  ◦ measurement: direct insertion of measurement code
int main( )
{ /* takes 100 secs */
    f1(); /* takes 20 secs */
    /* other work */
    f2(); /* takes 50 secs */
    f1(); /* takes 20 secs */
    /* other work */
}

/* similar for other metrics, such as hardware performance counters, etc. */

- Inclusive time for main:
  100 secs

- Exclusive time for main:
  100
  - 20
  - 50
  - 20
  ----------
  = 10 secs

- Exclusive time sometimes called “self”
TRACING: ADVANTAGES & DISADVANTAGES

• Recording of information about significant points (events) during program execution
  o entering/exiting code region (function, loop, block, …)
  o thread/process interactions (e.g., send/receive message)

• Save information in event record
  o Timestamp
  o CPU identifier, thread identifier
  o Event type and event-specific information

• Event Trace: Collection of all events of a process/program sorted by time
  Can be used to reconstruct dynamic program behavior
  o Profiles can be calculated from traces

Tracing Disadvantages
  o traces can become very large
  o instrumentation and tracing is complicated
  o event buffering, clock synchronization, …
COMMON EVENT TYPES

- enter/leave of function/routine/region
  - time stamp, process/thread, function ID

- send/receive of P2P message (MPI)
  - time stamp, sender, receiver, length, tag, communicator

- collective communication (MPI)
  - time stamp, process, root, communicator, # bytes

- hardware performance counter values
  - time stamp, process, counter ID, value

- etc.
PARALLEL TRACE FILE

10010 P 1 ENTER 5
10090 P 1 ENTER 6
10110 P 1 ENTER 12
10110 P 1 SEND TO 3 LEN 1024 ...
10330 P 1 LEAVE 12
10400 P 1 LEAVE 6
10520 P 1 ENTER 9
10550 P 1 LEAVE 9
...

10020 P 2 ENTER 5
10095 P 2 ENTER 6
10120 P 2 ENTER 13
10300 P 2 RECV FROM 3 LEN 1024 ...
10350 P 2 LEAVE 13
10450 P 2 LEAVE 6
10620 P 2 ENTER 9
10650 P 2 LEAVE 9
...

DEF TIMERRES 1000000000
DEF PROCESS 1 `Master`
DEF PROCESS 2 `Slave`
...
DEF FUNCTION 5 `main`
DEF FUNCTION 6 `foo`
DEF FUNCTION 9 `bar`
DEF FUNCTION 12 `MPI_Send`
DEF FUNCTION 13 `MPI_Recv`
...

03/01/11
TRACING EXAMPLE:
INSTRUMENTATION, MONITOR, TRACE

CPU A:

```c
void master {
    trace(EXIT, 1);
    ...
    trace(SEND, B);
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}
```

CPU B:

```c
void slave {
    trace(EXIT, 2);
    ...
    recv(A, tag, buf);
    ...
    trace(ENTER, 2);
}
```

Event definition:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Event</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td>ENTER</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>slave</td>
<td>ENTER</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td>SEND</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>EXIT</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>EXIT</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MONITOR

CPU A:

03/01/11

CPU B:

03/01/11

MONITOR
TRACING: TIMELINE VISUALIZATION

<table>
<thead>
<tr>
<th></th>
<th>master</th>
<th></th>
<th>slave</th>
<th></th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td></td>
<td>slave</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>slave</td>
<td></td>
<td>master</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>master</th>
<th></th>
<th>slave</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>master</td>
<td></td>
<td>slave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>A ENTER</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>B ENTER</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>A SEND</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>A EXIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>B RECV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>B EXIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
PERFORMANCE DATA ANALYSIS

• Draw conclusions from measured performance data

• Manual analysis
  o Visualization
  o Interactive exploration
  o Statistical analysis
  o Modeling
  o Examples: TAU, Vampir Suite, Paraver, Intel Trace Collector & Analyzer

• Automated analysis
  o Try to cope with huge amounts of performance by automation
  o Examples: Paradyn, KOJAK, Scalasca
AUTOMATED PERFORMANCE ANALYSIS

• Reason for Automation
  0 Size of systems: several tens of thousand of processors
  0 ORNL's Jaguar Cray XT5: 224,256 compute cores (2010)
  0 LLNL Sequoia (IBM, based on future Blue Gene arch.): ~1.6 million compute cores (2011-2012)
  0 Trend to multi-core

• Large amounts of performance data
  0 Several gigabytes or even terabytes
  0 Overwhelms user

• Not all programmers are performance experts
  0 Scientists want to focus on their domain
  0 Need to keep up with new machines

• Automation can solve some of these issues
This is a situation that can be detected automatically by analyzing the trace file

-> late sender pattern
Most modern CPUs have specialized registers that measure various performance parameters.

HW performance counters provide application developers with valuable information about code sections that can be improved.

Hardware performance counters can provide insight into:

- Whole program timing
- Cache behaviors
- Branch behaviors
- Memory and resource contention and access patterns
- Pipeline stalls
- Floating point efficiency
- Instructions per cycle
- Subroutine resolution
- Process or thread attribution
How do we know if the Counters are Working?

• Four Common Failures:
  • Infrastructure not working (PAPI, Kernel)
  • Wrong counter (PAPI, Kernel, User)
  • Counter works but gives wrong values
  • Counter is giving “correct” values but documentation is wrong
Determinism of the Counters

In general you will not get the exact same results from run to run.

Various sources of non-determinism:

- Operating System Interaction
- Program Layout / Address Space Randomization
- Measurement Overhead
- Multi-Core Interaction
- Hardware Implementation Details
Unexpected Overheads on x86_64 retired instruction counts

- +1 extra for every Hardware Interrupt
- +1 extra for each page-fault
- +1 for first floating point instruction
- +1 extra for each floating point exception
- +1 on AMD on FP state save if exception bit set
- Instruction double counts on Pentium D with
  
  INSTRUCTIONS_RETIRED:NBOGUSNTAG

03/01/11
## Retired Instruction Counts

Difference from the expected 226,990,030 instructions

<table>
<thead>
<tr>
<th>Machine</th>
<th>Before Adjustment</th>
<th>Adjusted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core2</td>
<td>10,879±319</td>
<td>13±1</td>
</tr>
<tr>
<td>Atom</td>
<td>11,601±495</td>
<td>-41±12</td>
</tr>
<tr>
<td>Nehalem</td>
<td>11,409±3</td>
<td>8±2</td>
</tr>
<tr>
<td>Nehalem-EX</td>
<td>11,915±9</td>
<td>8±2</td>
</tr>
<tr>
<td>Pentium D (int retired)</td>
<td>2,610,571±8</td>
<td>561±3</td>
</tr>
<tr>
<td>Pentium D (inst completed)</td>
<td>10,794±28</td>
<td>-50±5</td>
</tr>
<tr>
<td>Phenom</td>
<td>310,601±11</td>
<td>12±0</td>
</tr>
<tr>
<td>Istanbul</td>
<td>311,830±78</td>
<td>11±1</td>
</tr>
</tbody>
</table>
Sometimes Results are Hard to Explain

Retired loads on Pentium 4 when using rep movsb show microcoded results
Understanding HW Counters Often Requires Micro-Architectural Knowledge

• Results might be unexpected.
• Counters typically not validated and bugs in counters are unlikely to hold up a chip release.
• Counters traditionally designed with chip designers, not end users, in mind.
L1 Cache Access Example

float array[1000], sum = 0.0;

PAPI_start_counters(events, 1);

for(int i=0; i<1000; i++) {
    sum += array[i];
}

PAPI_stop_counters(counts, 1);
L1 Cache Access Example

Normalized to 1000 loads

Normalized Accesses

- PPro
- P4
- Atom
- Core2
- Istanbul
- Nehalem EX
- Nehalem gcc 4.1
- Nehalem gcc 4.3

03/01/11
Must Look at Assembly Language

Expected Code (gcc 4.3)
* 4020d8:  f3 0f 58 00  addss (%rax),%xmm0
           48 83 c0 04  add   $0x4,%rax
           48 39 d0     cmp   %rdx,%rax
           75 f3        jne   4020d8

Unexpected Code (gcc 4.1)
* 401e18:  f3 0f 10 44  movss 0xc(%rsp),%xmm0
* 401e1e:  f4 0f 58 04  addss (%rdx,%rax,4),%xmm0
           48 83 c0 01  add   $0x1,%rax
           48 3d e8 03  cmp   $0x3e8,%rax
* 401e2d:  f3 0f 11 24  movvss %xmm0,0xc(%rsp)
           75 e3        jne   401e18
Cache Example
Allocate array as big as L1 D$
Walk through it one byte at a time
Count misses when walked fwd/back/random
Potential Causes of Divergences

- Hardware Prefetching.
- Speculative Execution
- PAPI Measurement Noise.
- Operating System Noise.
- Non-LRU Cache Replacement.
- Coherence Misses from other Processors.
- Bugs in the counters.
- Wrong counter being chosen.
Allocate array as big as L2 Cache
Walk through it one byte at a time
Count misses when walked fwd/back/random
WHAT’S PAPI?

• **Middleware** that provides a consistent and efficient programming interface for the performance counter hardware found in most major microprocessors.

• Started as a Parallel Tools Consortium project in 1998
  - Goal was to produce a specification for a **portable interface** to the hardware performance counters.

• Countable events are defined in two ways:
  - Platform-neutral **Preset Events** (e.g., PAPI_TOT_INS)
  - Platform-dependent **Native Events** (e.g., L3_CACHE_MISS)

• Preset Events can be **derived** from multiple Native Events (e.g. PAPI_L1_TCM might be the sum of L1 Data Misses and L1 Instruction Misses on a given platform)
PAPI HARDWARE 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` to see what preset events are available on a given platform

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_event_chooser` utility to select a compatible set of events
WHERE'S PAPI?

PAPI runs on most modern processors and Operating Systems of interest to HPC:

- IBM POWER{4, 5, 5+, 6} / AIX or Linux
- PowerPC{-32, -64, 970} / Linux
- Cell
- Blue Gene / {L, P}
- Intel Pentium II, III, 4, M, Core, etc. / Linux
- Intel Itanium{1, 2, Montecito, Montvale}
- AMD Athlon, Opteron / Linux
- Cray XT{3, 4, 5} Catamount, CNL
- Altix, Sparc, SiCortex…
- …and even Windows {XP, 2003 Server; PIII, Athlon, Opteron}!
- …but not Mac 😞
PAPI COUNTER INTERFACES

PAPI provides 3 interfaces to the underlying counter hardware:

1. A Low Level API manages hardware events (preset and native) in user defined groups called EventSets. Meant for experienced application programmers wanting fine-grained measurements.

2. A High Level API provides the ability to start, stop and read the counters for a specified list of events (preset only). Meant for programmers wanting simple event measurements.

1. Graphical and end-user tools provide facile data collection and visualization.
PAPI Block Diagram

PAPI_TOT_INS

PAPI

INSTRUCTION RETIRED

Event Name Translator

0x5300c0

Operating System

Hardware
PAPI HIGH LEVEL CALLS

- **PAPI_num_counters()**
  - get the number of hardware counters available on the system

- **PAPI_flips (float *rtime, float *ptime, long long *flpins, float *mflips)**
  - simplified call to get Mflips/s (floating point instruction rate), real and processor time

- **PAPI_flops (float *rtime, float *ptime, long long *flpops, float *mflops)**
  - simplified call to get Mflops/s (floating point operation rate), real and processor time

- **PAPI_ipc (float *rtime, float *ptime, long long *ins, float *ipc)**
  - gets instructions per cycle, real and processor time

- **PAPI_accum_counters (long long *values, int array_len)**
  - add current counts to array and reset counters

- **PAPI_read_counts (long long *values, int array_len)**
  - copy current counts to array and reset counters

- **PAPI_start_counts (int *events, int array_len)**
  - start counting hardware events

- **PAPI_stop_counts (long long *values, int array_len)**
  - stop counters and return current counts
EXAMPLE: LOW LEVEL API

```c
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={ PAPI_FP_OPS, PAPI_TOT_CYC };
int EventSet = PAPI_NULL;
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);

do_work(); /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop (EventSet, values);
```
krakenpf7: cs594> papi_cost -h
This is the PAPI cost program.
It computes min / max / mean / std. deviation for PAPI start/stop pairs and for PAPI reads. Usage:

    cost [options] [parameters]
    cost TESTS_QUIET

Options:

- **-b BINS**  set the number of bins for the graphical distribution of costs. Default: 100
- **-d**  show a graphical distribution of costs
- **-h**  print this help message
- **-s**  show number of iterations above the first 10 std deviations
- **-t THRESHOLD**  set the threshold for the number of iterations. Default: 100,000
Overhead in cycles on a Core2 machine

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Start/Stop</th>
<th>Read</th>
<th>Read ts</th>
<th>Accum</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.32 perf_events</td>
<td>17,749</td>
<td>8,982</td>
<td>9,019</td>
<td>11,519</td>
<td>2,550</td>
</tr>
<tr>
<td>2.6.35 perf_events</td>
<td>8,683</td>
<td>2,089</td>
<td>2,109</td>
<td>4,196</td>
<td>2,084</td>
</tr>
<tr>
<td>2.6.30 perfmon</td>
<td>9,644</td>
<td>1,200</td>
<td>1,202</td>
<td>3,623</td>
<td>2,406</td>
</tr>
<tr>
<td>2.6.32 perfctr</td>
<td>5,702</td>
<td>195</td>
<td>203</td>
<td>3,772</td>
<td>3,556</td>
</tr>
</tbody>
</table>
krakenpf7: cs594> papi_avail -h
Usage: papi_avail [options]
Options:

General command options:
- a, --avail   Display only available preset events
- d, --detail  Display detailed information about all preset events
- e EVENTNAME  Display detail information about specified preset or native event
- h, --help    Print this help message

This program provides information about PAPI preset and native events.
krakenpf7: cs594> aprun -n1 papi_avail
Available events and hardware information.

PAPI Version : 3.6.2.2
Vendor string and code : AuthenticAMD (2)
Model string and code : 6-Core AMD Opteron(tm) Processor 23 (D0) (16)
CPU Revision : 0.000000
CPU Megahertz : 2600.000000
CPU Clock Megahertz : 2600
CPU's in this Node : 12
Nodes in this System : 1
Total CPU's : 12
Number Hardware Counters : 4
Max Multiplex Counters : 512

The following correspond to fields in the PAPI_event_info_t structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Avail</th>
<th>Deriv</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>Yes</td>
<td>No</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>0x80000003</td>
<td>Yes</td>
<td>No</td>
<td>Level 2 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>
</tbody>
</table>

[...]

Of 103 possible events, 41 are available, of which 9 are derived.
Available events and hardware information.

PAPI Version : 3.6.2.2
Vendor string and code : AuthenticAMD (2)
Model string and code : 6-Core AMD Opteron(tm) Processor 23 (D0) (16)
CPU Revision : 0.000000
CPU Megahertz : 2600.000000
CPU Clock Megahertz : 2600
CPU's in this Node : 12
Nodes in this System : 1
Total CPU's : 12
Number Hardware Counters : 4
Max Multiplex Counters : 512

The following correspond to fields in the PAPI_event_info_t structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Deriv</th>
<th>Description (Note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>No</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x80000001</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>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>0x80000003</td>
<td>No</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L1_TCM</td>
<td>0x80000006</td>
<td>Yes</td>
<td>Level 1 cache misses</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAPI_FP_OPS</td>
<td>0x80000066</td>
<td>No</td>
<td>Floating point operations</td>
</tr>
</tbody>
</table>

Of 41 available events, 9 are derived.
krakenpf7: cs594> aprun -n1 papi_avail -e PAPI_L1_TCM

[...]
Event name: PAPI_L1_TCM
Event Code: 0x80000006
Number of Native Events: 2
Short Description: L1 cache misses
Long Description: Level 1 cache misses
Developer's Notes: 
Derived Type: DERIVED_ADD
Postfix Processing String: 
Native Code[0]: 0x40000029 INSTRUCTION_CACHE_MISSES
Number of Register Values: 4
Register[ 0]: 0x00000081 Event Code
Register[ 1]: 0x00000081 Event Code
Register[ 2]: 0x00000081 Event Code
Register[ 3]: 0x00000081 Event Code
Native Event Description: Instruction Cache Misses

Native Code[1]: 0x40000011 DATA_CACHE_MISSES
Number of Register Values: 4
Register[ 0]: 0x00000041 Event Code
Register[ 1]: 0x00000041 Event Code
Register[ 2]: 0x00000041 Event Code
Register[ 3]: 0x00000041 Event Code
Native Event Description: Data Cache Misses
```
krakenpf7: cs594> aprun -n1 papi_native_avail

Available native events and hardware information.

<table>
<thead>
<tr>
<th>Event Code</th>
<th>Symbol</th>
<th>Long Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40000003</td>
<td>RETIRED_SSE_OPERATIONS</td>
<td>Retired SSE Operations</td>
</tr>
<tr>
<td>40001003</td>
<td>:SINGLE_ADD_SUB_OPS</td>
<td>Single precision add/subtract ops</td>
</tr>
<tr>
<td>40002003</td>
<td>:SINGLE_MUL_OPS</td>
<td>Single precision multiply ops</td>
</tr>
<tr>
<td>40004003</td>
<td>:SINGLE_DIV_OPS</td>
<td>Single precision divide/square root ops</td>
</tr>
<tr>
<td>40008003</td>
<td>:DOUBLE_ADD_SUB_OPS</td>
<td>Double precision add/subtract ops</td>
</tr>
<tr>
<td>40010003</td>
<td>:DOUBLE_MUL_OPS</td>
<td>Double precision multiply ops</td>
</tr>
<tr>
<td>40020003</td>
<td>:DOUBLE_DIV_OPS</td>
<td>Double precision divide/square root ops</td>
</tr>
<tr>
<td>40040003</td>
<td>:OP_TYPE</td>
<td>Op type: 0=uops. 1=FLOPS</td>
</tr>
<tr>
<td>40080003</td>
<td>:ALL</td>
<td>All sub-events selected</td>
</tr>
</tbody>
</table>

[...]

0x40000010   DATA_CACHE_ACCESSSES | Data Cache Accesses

0x40000011   DATA_CACHE_MISSES | Data Cache Misses

[...]

Total events reported: 114
```
krakenpf7: cs594> aprun -n1 papi_event_chooser

Usage:
papi_event_chooser NATIVE|PRESET evt1 evt2 ...
krakenpf7: cs594> aprun -n1 papi_event_chooser PRESET PAPI_L1_TCM

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Deriv</th>
<th>Description (Note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>No</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x80000001</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>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>0x80000003</td>
<td>No</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_TCM</td>
<td>0x80000007</td>
<td>No</td>
<td>Level 2 cache misses</td>
</tr>
<tr>
<td>PAPI_L3_TCM</td>
<td>0x80000008</td>
<td>No</td>
<td>Level 3 cache misses</td>
</tr>
<tr>
<td>PAPI_FPU_IDL</td>
<td>0x80000012</td>
<td>No</td>
<td>Cycles floating point units are idle</td>
</tr>
<tr>
<td>PAPI_TLB_DM</td>
<td>0x80000014</td>
<td>No</td>
<td>Data translation lookaside buffer misses</td>
</tr>
<tr>
<td>PAPI_TLB_IM</td>
<td>0x80000015</td>
<td>No</td>
<td>Instruction translation lookaside buffer miss</td>
</tr>
<tr>
<td>PAPI_TLB_TL</td>
<td>0x80000016</td>
<td>Yes</td>
<td>Total translation lookaside buffer misses</td>
</tr>
</tbody>
</table>

[...]

PAPI_FP_OPS   0x80000066  No  Floating point operations

-------------------------------------------------------------------------
Total events reported: 39
krakenpf7: cs594> aprun -n1 papi_eventchooser

PRESET PAPI_L1_TCM PAPI_TLB_TL

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Deriv</th>
<th>Description (Note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x800000000</td>
<td>No</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x800000001</td>
<td>No</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_TLB_DM</td>
<td>0x800000014</td>
<td>No</td>
<td>Data translation lookaside buffer misses</td>
</tr>
<tr>
<td>PAPI_TLB_IM</td>
<td>0x800000015</td>
<td>No</td>
<td>Instruction translation lookaside buffer miss</td>
</tr>
</tbody>
</table>

Total events reported: 4
PAPI UTILITIES: PAPI_COMMAND_LINE

```bash
krakenpf7: cs594> aprun -n1 papi_command_line PAPI_FP_OPS
Successfully added: PAPI_FP_OPS

PAPI_FP_OPS : 40000000

Verification: None.
This utility lets you add events from the command line interface to see if they work.

krakenpf7: cs594> aprun -n1 papi_command_line PAPI_FP_OPS PAPI_L1_TCM
Successfully added: PAPI_FP_OPS
Successfully added: PAPI_L1_TCM

PAPI_FP_OPS : 40000000
PAPI_L1_TCM : 40
```
PERFORMANCE MEASUREMENT CATEGORIES

- **Efficiency**
  - Instructions per cycle (IPC)
  - Memory bandwidth

- **Caches**
  - Data cache misses and miss ratio
  - Instruction cache misses and miss ratio

- **L2 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
THE CODE

#define ROWS 1000    // Number of rows in each matrix
#define COLUMNS 1000  // Number of columns in each matrix

```c
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;
        }
    }
}
```

```c
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.
IPC – INSTRUCTIONS PER CYCLE

- Measure instruction level parallelism
- An indicator of code efficiency

```c
int events[] = {PAPI_TOT_CYC, PAPI_TOT_INS};

realtime[0] = PAPI_get_real_usec();
retval = PAPI_start_counters(events, 2);
classic_matmul();
retval = PAPI_stop_counters(cvalues, 2);
realtime[1] = PAPI_get_real_usec();
```

```c
int events[] = {PAPI_TOT_CYC, PAPI_TOT_INS};

retval = PAPI_library_init (PAPI_VER_CURRENT);
retval = PAPI_create_eventset(&EventSet);
retval = PAPI_add_events(EventSet, events, 2);
realtime[0] = PAPI_get_real_usec();
retval = PAPI_start(EventSet);
classic_matmul();
retval = PAPI_stop(EventSet, cvalues);
realtime[1] = PAPI_get_real_usec();
```
# 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></td>
<td>=================</td>
<td>==================</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High Level IPC Test (PAPI_{start,stop}_counters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real time</td>
</tr>
<tr>
<td>IPC</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low Level IPC Test (PAPI low level calls)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real time</td>
</tr>
<tr>
<td>IPC</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
</tr>
</tbody>
</table>

- Both PAPI methods are consistent
- Roughly 460% improvement in reordered code
DATA CACHE ACCESS

Cache miss: a failed attempt to read or write a piece of data in the cache
→ Results in main memory access with much longer latency
→ Important to keep data as close as possible to CPU

Data Cache Misses can be considered in 3 categories:
• **Compulsory misses:** Occurs on first reference to a data item
  - Prefetching can help

• **Capacity misses:** Occurs when the working set exceeds the cache capacity
  - **Spatial locality:** use all the data that is loaded into the cache
  - Smaller working set (blocking/tiling algorithms)

• **Conflict misses:** Occurs when a data item is referenced after the cache line containing the item was evicted earlier.
  - **Temporal locality:** reuse a word as long as possible
  - Data layout; memory access patterns

**Coherency Miss:** Occurs when other processor has invalidated
## 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>PAPI NATIVE EVENTS:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA_CACHE_ACCESSES</td>
<td>2,002,807,841</td>
<td>3,008,528,961</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS:L2_MODIFIED:L2_OWNED:L2_EXCLUSIVE:L2_SHARED</td>
<td>205,968,263</td>
<td>60,716,301</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS_FROM_SYSTEM:MODIFIED:OWNED:EXCLUSIVE:SHARED</td>
<td>61,970,925</td>
<td>1,950,282</td>
</tr>
<tr>
<td>PAPI PRESET EVENTS:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAPI_L1_DCA</td>
<td>2,002,808,034</td>
<td>3,008,528,895</td>
</tr>
<tr>
<td>PAPI_L1_DCM</td>
<td>268,010,587</td>
<td>62,680,818</td>
</tr>
</tbody>
</table>

- Data Cache Request Rate: 0.2224 req/inst
- Data Cache Miss Rate: 0.0298 miss/inst
- Data Cache Miss Ratio: 0.1338 miss/req

- Data Cache Request Rate: 0.3339 req/inst
- Data Cache Miss Rate: 0.0070 miss/inst
- Data Cache Miss Ratio: 0.0208 miss/req

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

03/01/11
EXAMPLE: SIX-CORE AMD OPTERON (ISTANBUL)

L1: 64 + 64 KB (DATA + INSTRUCTIONS)
L2: 512 KB

Three different Matrix Multiply implementations
(Blue: classic – Red: Transpose – Green: AMD Core Math Lib)

Note: logarithmic scale for L1_DCM
Motivation:

- Hardware counters aren’t just for CPUs anymore
  - Network counters; thermal & power measurement, GPU, …
- Often insightful to measure multiple counter domains at once

Goals:

- Support simultaneous access to on- and off-processor counters
- Isolate hardware dependent code in a separable component module
- Extend platform independent code to support multiple simultaneous components
- Add or modify API calls to support access to any of several components
- Modify build environment for easy selection and configuration of multiple available components
COMPONENT PAPI

PAPI FRAMEWORK

Low Level User API

High Level User API

Developer API

PAPI COMPONENT (NETWORK)
  Operating System
  Counter Hardware

PAPI COMPONENT (CPU)
  Operating System
  Counter Hardware

PAPI COMPONENT (THERMAL)
  Operating System
  Counter Hardware
SOME TOOLS THAT USE PAPI

• TAU (U Oregon) [http://www.cs.uoregon.edu/research/tau/](http://www.cs.uoregon.edu/research/tau/)
• PerfSuite (NCSA) [http://perfsuite.ncsa.uiuc.edu/](http://perfsuite.ncsa.uiuc.edu/)
• HPCToolkit (Rice Univ) [http://hipersoft.cs.rice.edu/hpctoolkit/](http://hipersoft.cs.rice.edu/hpctoolkit/)
• KOJAK and SCALASCA (FZ Juelich, UTK) [http://icl.cs.utk.edu/kojak/](http://icl.cs.utk.edu/kojak/)
• VampirTrace and Vampir (TU Dresden) [http://www.vamir.eu](http://www.vamir.eu)
• SvPablo (UNC Renaissance Computing Institute) [http://www.renci.unc.edu/Software/Pablo/pablo.htm](http://www.renci.unc.edu/Software/Pablo/pablo.htm)
• ompP (UTK) [http://www.ompp-tool.com](http://www.ompp-tool.com)
Hardware Counter Documentation

  http://www.intel.com/products/processor/manuals/

- AMD: BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors
Accessing Counters w/o PAPI

• **Perf** - works on Linux more recent that 2.6.31
  
  ```bash
  > perf stat -e instructions -- /bin/ls
  
  Performance counter stats for '/bin/ls':
  1706164 instructions  # 0.000 IPC
  0.003998449 seconds time elapsed
  
  > ./libpfm4/perf_examples/task -e
  INSTRUCTIONS_RETIRED -- /bin/ls
  1,715,943 INSTRUCTIONS_RETIRED (0.00% scaling,
  ena=2,090,539, run=2,090,539)
  ```

• **Libpfm4**

  > ./libpfm4/perf_examples/task -e

  INSTRUCTIONS_RETIRED -- /bin/ls
  1,715,943 INSTRUCTIONS_RETIRED (0.00% scaling,
  ena=2,090,539, run=2,090,539)
1. **Motivation**
   - What is Performance?
   - Why being annoyed with Performance Analysis?

1. **Concepts and Definitions**
   - The performance analysis cycle
   - Measurement: profiling vs. tracing
   - Analysis: manual vs. automated

1. **Performance Analysis Tools**
   - **PAPI**: Access to hardware performance counters
   - **Vampir Suite**: Instrumentation and Trace visualization
   - **KOJAK / Scalasca**: automatic performance analysis tool
   - **TAU**: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
VAMPIR: PERFORMANCE ANALYSIS SUITE

Consists of:

• VampirTrace part for instrumentation, monitoring and recording
• VampirServer part for visualization and analysis

VampirTrace

• Supports a variety of performance aspects: e.g. MPI comm events, subroutine calls from user code, HW perf counter, I/O events, etc.

VampirServer

• Implements client / server model with distributed server
• Allows very scalable interactive visualization for traces
Consists of:

- VampirTrace part for instrumentation, monitoring and recording
- VampirServer part for visualization and analysis

**VampirTrace**

- Supports a variety of performance aspects: e.g. MPI comm events, subroutine calls from user code, HW perf counter, I/O events, etc.

**VampirServer**

- Implements client / server model with distributed server
- Allows very scalable interactive visualization for traces
• **Instrumentation**: Process of modifying programs to detect and report events by calling *instrumentation* functions.

• *instrumentation functions* provided by trace library

• call == notification about run-time event

• there are various ways of instrumentation:

  o **Source Code instrumentation** (manual or automatic)

  o **Wrapper function instrumentation** (e.g. MPI profiling interface)

  o **Compiler instrumentation** (e.g. gcc `-finstrument-functions -c foo.c`)
    supported by many compilers (GCC, Intel, IBM, PGI, NEC, Hitachi, Sun Fortran, …)

  o **Dynamic instrumentation** (modify binary executable in memory)
    very platform/machine dependent, expensive (see www.dyninst.org)
• provide instrumentation functions

• receive events of various types

• collect event properties
  - time stamp
  - location (thread, process, cluster node, MPI rank)
  - event specific properties
  - perhaps hardware performance counter values

• record to memory buffer, flush eventually

• try to be fast, minimize overhead
• TAU Trace Format (University of Oregon)
• Epilog (ZAM, FZ Jülich, Germany)
• STF (Pallas, now Intel)
• Open Trace Format (OTF)
  ◦ ZIH, TU Dresden, Germany in cooperation with Oregon & Jülich
  ◦ single/multiple files per trace with
  ◦ fast sequential and random access
  ◦ including API for writing/reading
  ◦ supports auxiliary information
  ◦ see http://www.tu-dresden.de/zih/otf/
INSTRUMENTATION WITH VAMPIRTRACE

Edit – Compile – Run Cycle

Source Code → Compiler → Binary → Run → Results

Edit – Compile – Run Cycle with VampirTrace

Source Code → Compiler → VT Wrapper → Binary → Run → Results

Traces
INSTRUMENTATION TYPES

- Automatic instrumentation using compiler wrappers
- Manual instrumentation
- Binary instrumentation
Easiest way of using VampirTrace
No source code modifications
In the build system of your application: Swap calls to the regular compiler with calls to the VampirTrace compiler wrappers
  - For compiling and linking
  - e.g. in the makefile change: CC=mpicc CC=vtcc

- Rebuild the application
- Run the application to produce trace data
Captured events:

- All user function entries and exits
  - If supported by the compiler (Intel, GNU, PGI, NEC, IBM)
- MPI calls and messages
  - If the application is MPI parallel
- OMP regions
  - If the application is OpenMP parallel

```bash
icc  hello.c  -o hello
vtcc hello.c  -o hello
icpc hello_parallel.cpp  -lmpi  -o hello_parallel
vtcxx hello_parallel.cpp  -lmpi  -o hello_parallel
```
• Allows detailed source code instrumentation, e.g. regions of functions such as loops
• Can – and should – be combined with automatic instrumentation
• Be sure to instrument all function exits!

• Add the following into our source code to instrument a region, e.g. C: (available for C++ and FORTRAN as well)

```c
#include "vt_user.h"
...
VT_USER_START("Region_1");
...
VT_USER_END("Region_1");
...
```

• Compile with “-DVTRACE”
  • otherwise, VampirTrace macros will expand to empty blocks, producing zero overhead

```bash
vtcc -vt:inst manual prog.c -DVTRACE -o prog
```
RUNTIME MEASUREMENT SETTINGS

- Environment Variables
- PAPI hardware performance counters
- Memory allocation counters
- Application I/O calls
- Filtering
- Grouping
**ENVIRONMENT VARIABLES**

- By default, trace data is written to the `pwd`
- About everything of this can be customized with environment variables
- Environment variables must be set prior to running the application, not prior to building the application

VT_PFORM_GDIR Directory where final trace file is stored
VT_PFORM_LDIR Directory for intermediate trace files
VT_FILE_PREFIX Trace file name
VT_BUFFER_SIZE Internal trace buffer size
VT_MAX_FLUSHES Max number of buffer flushes
VT_MEMTRACE Enable memory allocation tracing
VT_IOTRACE Enable I/O tracing
VT_MPITRACE Enable MPI tracing
VT_FILTER_SPEC Name of filter file
VT_GROUPS_SPEC Name of function groups file
VT_COMPRESSION Compress trace files
VT_METRICS List of PAPI counters
PAPI COUNTERS IN TRACES

• PAPI counters can be included in traces
  • If VampirTrace was build with PAPI support
  • If PAPI is available on the platform

• VT_METRICS can be used to specify a colon-separated list of PAPI counters

```bash
export VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM
```
Filtering is one of the ways to reduce trace file size.

Environment variable VT_FILTERSPEC:

```sh
export VT_FILTERSPEC=/home/user/filter.spec
```

Filter file contains a list of filters for functions that are applied during the execution of the application:

```sh
my*;test -- 1000
calculate -- -1          # -1: unlimited
*      -- 1000000
```

The vtfilter tool:

- can create a filter file
- can reduce the size of trace files
• Groups can be defined by the user to group related functions
  • Groups can be assigned different colors in Vampir and VampirServer, highlighting application behavior
• Environment variable VT_GROUPS_SPEC
  
  ```
  export VT_GROUPS_SPEC=/home/<user>/groups.spec
  ```

• Groups file contains a list of groups with associated functions
  
  ```
  CALC=calculate
  MISC=my*;test
  UNKNOWN=*  
  ```
Consists of:

- VampirTrace part for instrumentation, monitoring and recording
- VampirServer part for visualization and analysis

**VampirTrace**

- Supports a variety of performance aspects: e.g. MPI comm events, subroutine calls from user code, HW perf counter, I/O events, etc.

**VampirServer**

- Implements client / server model with distributed server
- Allows very scalable interactive visualization for traces
• Vampir Architecture
• Various Displays
  ◦ Global Timeline, Process Timeline + Counter Display
  ◦ Summary Chart (aka. Profile)
  ◦ Summary Timeline
  ◦ Message Statistics
  ◦ Collective Communication Statistics
  ◦ Counter Timeline
  ◦ I/O Event Display
  ◦ Call Tree
Merged Traces Analysis

Classic Analysis:
Worker 1
Worker 2
Worker m

Parallel Program
Monitor System
Process

File System
Trace 1
Trace 2
Trace 3
Trace N

Parallel I/O
Message Passing

Visualization Client

Timeline with 16 Traces visible

768 Processes Thumbnail View

Segment Indicator

VAMPIR ARCHITECTURE 1/2

03/01/11
parallel/distributed server
  - runs in (part of) production environment
  - no need to transfer huge traces
  - parallel I/O
lightweight client on local workstation
  - receive visual content only
  - already adapted to display resolution
  - moderate network load
scales to traces > 40 GB
TIMELINE DISPLAY (ZOOMED)

1. Shows all selected processes
2. Shows state changes (activity color)
3. Shows messages, collective and MPI–IO operations
Message statistics for each process/node pair:

- Byte and message count
- min/max/avg message length, bandwidth
COLLECTIVE OPERATIONS

Start of op

Data being sent

Stop of op

Data being received

Connection lines
Trace Visualization

- Vampir provides a number of display types
- each allows many different options

Advice

- make a hypothesis about performance problems
- consider application's internal workings if known
- select the appropriate display
- use statistic displays in conjunction with timelines
FINDING PERFORMANCE BOTTLENECKS

Where?
- Computation
- Communication
- Memory, I/O, ...
- Tracing itself

What?
- unbalanced computation (e.g. single late comer)
- strictly serial parts of program (e.g. idle processes/threads)
- very frequent tiny function calls
- sparse loops
EXAMPLE: IDLE OPENMP THREADS
• communication as such (domination over computation)

• late sender, late receiver

• point-to-point messages instead of collective communication

• unmatched messages

• overcharge of MPI’s buffers

• bursts of large messages (bandwidth)

• frequent short messages (latency)

• unnecessary synchronization (barrier)
**EXAMPLE: DOMINATING COMMUNICATION**

- **Summary Timeline**: # of processes, actively involved in activity at certain point in time (vertical histogram)
• memory bound computation
  • inefficient L1/L2/L3 cache usage
  • TLB misses
  • detectable via HW performance counters

• I/O bound computation
  • slow input/output
  • sequential I/O on single process
  • I/O load imbalance

• exception handling
EXAMPLE: LOW FLOATING POINT RATE DUE TO HEAVY FP EXCEPTIONS

PAPI counter:
• Measurement overhead
  o especially grave for tiny function calls
  o solve with selective instrumentation; filtering
• Long, asynchronous trace buffer flushes
• Too many concurrent counters
• Heisenbugs
  o software bug that disappears or alters its characteristics when an attempt is made to study it
  o considered exceptionally difficult to understand and repair
EXAMPLE: TRACE BUFFER FLUSH

- Default: VT_BUFFER_SIZE = 32 MB ; VT_MAX_FLUSHES = 1
- Events that are to be recorded after the limit has been reached are no longer written into the trace file
- Remove limit to get complete trace of an application: VT_MAX_FLUSHES = 0
  Then VampirTrace always writes the buffer to disk when it is full
- Increasing size of VT buffer will decrease memory available to application
  → parts of app. swapped to disk → leads to significant change in behavior of app.
OUTLINE

1. Motivation
   • What is Performance?
   • Why being annoyed with Performance Analysis?

1. Concepts and Definitions
   • The performance analysis cycle
   • Measurement: profiling vs. tracing
   • Analysis: manual vs. automated

1. Performance Analysis Tools
   • PAPI: Access to hardware performance counters
   • Vampir Suite: Instrumentation and Trace visualization
   • KOJAK / Scalasca: automatic performance analysis tool
   • TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
BASIC IDEA

Huge amount of Measurement data

• For non-standard / tricky cases (10%)
• For expert users

• For standard cases (90% ?!)
• For “normal” users
• Starting point for experts

Automatic searching of event traces of parallel applications or execution patterns that indicate inefficient behavior!
**MPI-1 PATTERN: WAIT AT BARRIER**

- Time spent in front of MPI synchronizing operation such as barriers
- This pattern covers the time spent waiting in front of an MPI barrier, which is the time inside the barrier call until the last process has reached the barrier. A large amount of waiting time spent in front of barriers can be an indication of load imbalance.
**MPI-1 PATTERN: LATE SENDER / RECEIVER**

- **Late Sender:** Time lost waiting in a blocking receive operation posted earlier than the corresponding send operation.

- **Late Receiver:** Time lost waiting in a blocking send operation until the corresponding receive operation is called.
KOJAK: SPPM RUN ON (8X16X14) 1792 PES

- New topology display
- Shows distribution of pattern over HW topology
- Easily scales to even larger systems
1. Motivation
   • What is Performance?
   • Why being annoyed with Performance Analysis?

1. Concepts and Definitions
   • The performance analysis cycle
   • Measurement: profiling vs. tracing
   • Analysis: manual vs. automated

1. Performance Analysis Tools
   • PAPI: Access to hardware performance counters
   • Vampir Suite: Instrumentation and Trace visualization
   • KOJAK / Scalasca: automatic performance analysis tool
   • TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
TAU PARALLEL PERFORMANCE SYSTEM

- Multi-level performance instrumentation
  - Multi-language automatic source instrumentation

- Flexible and configurable performance measurement

- Widely-portied parallel performance profiling system
  - Computer system architectures and operating systems
  - Different programming languages and compilers

- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid

- Integration in complex software, systems, applications
USING TAU: A BRIEF INTRODUCTION

• To instrument source code use TAU stub Makefiles:
  \% setenv TAU_MAKEFILE /usr/pkgs/tau/xt3/lib/Makefile.tau-mpi-pdt-pgi

And use tau_f90.sh, tau_cxx.sh or tau_cc.sh as Fortran, C++ or C compilers:
\% mpif90 foo.f90
changes to
\% tau_f90.sh foo.f90

• Execute application and then run:
  \% pprof  (flat profile for text based profile display)
  \% paraprof  (for GUI)
TAU INSTRUMENTATION APPROACH

• Support for standard program events
  • Routines
  • Classes and templates
  • Statement-level blocks
• Support for user-defined events
  • Begin/End events ("user-defined timers")
  • Atomic events (e.g., size of memory allocated/freed)
  • Selection of event statistics
• Support definition of “semantic” entities for mapping
• Support for event groups
• Instrumentation optimization (eliminate instrumentation in lightweight routines)
Flexible instrumentation mechanisms at multiple levels:

- **Source code**
  - manual (TAU API, TAU Component API)
  - automatic
    - C, C++, F77/90/95 (Program Database Toolkit (*PDT*))
    - OpenMP (directive rewriting (*Opari*), *POMP* spec)
- **Object code**
  - pre-instrumented libraries (e.g., MPI using *PMPI*)
  - statically-linked and dynamically-linked
- **Executable code**
  - dynamic instrumentation (pre-execution) (*DynInstAPI*)
  - virtual machine instrumentation (e.g., Java using *JVMPI*)
  - Python interpreter based instrumentation at runtime
TAU MEASUREMENT APPROACH

• Portable and scalable parallel profiling solution
  • Multiple profiling types and options
  • Event selection and control (enabling/disabling, throttling)
  • Online profile access and sampling
  • Online performance profile overhead compensation
• Portable and scalable parallel tracing solution
  • Trace translation to Open Trace Format (OTF)
  • Trace streams and hierarchical trace merging
• Robust timing and hardware performance support
• Multiple counters (hardware, user-defined, system)
• Performance measurement for CCA component software
PARAPROF MAIN WINDOW

Analyze workload characteristics

click left mouse button

click right mouse button
3D PERFORMANCE DATA EXPLORATION WITH TAU

Paraprof viewer (from the TAU toolset)
• Each point is a “thread” of execution

• A total of four metrics shown in relation

• ParaVis 3D profile visualization library
  • JOGL

32k processors
• Four significant events automatically selected (from 16K processors)
• Clusters and correlations are visible
PARAPROF – CALLGRAP

• A simple one ....

![Call Graph for n,c,t, 0,0,0 – tmp/private/](Call Graph for n,c,t, 0,0,0 – tmp/private/)

- main() (calls f1, f5)
  - f5() (sleeps 5 sec)
  - f1() (sleeps 1 sec, calls f2, f4)
    - f4() (sleeps 4 sec, calls f2)
      - f2() (sleeps 2 sec, calls f3)
        - f3() (sleeps 3 sec)
PERFEXPLORER - CORRELATION ANALYSIS (FLASH)

- Describes strength and direction of a linear relationship between two variables (events) in the data
-0.995 indicates strong, negative relationship

As CALC_CUT_BLOCK_CONTRIBUTIONS() increases in execution time, MPI_Barrier() decreases
DOCUMENTATION, MANUALS, USER GUIDES

• **PAPI:**
  PAPI documentation page available from the PAPI website:
  http://icl.cs.utk.edu/papi/

• **VAMPIR:**
  VampirTrace download and documentation:
  http://www.tu-dresden.de/zih/vampirtrace
  Vampir Suite
  http://www.vampir-ng.org/

• **Scalasca:**
  http://www.scalasca.org

• **TAU:**
  TAU Users Guide and papers available from the TAU website:
  http://www.cs.uoregon.edu/research/tau/
SUMMARY: STEPS OF PERFORMANCE EVALUATION

• Collect basic routine-level timing profile to determine where most time is being spent

• Collect routine-level hardware counter data to determine types of performance problems

• Collect callpath profiles to determine sequence of events causing performance problems

• Conduct finer-grained tracing to pinpoint performance bottlenecks
  o E.g. Loop-level profiling with hardware counters
  o E.g. Tracing of communication operations
CONCLUSION

• performance analysis very important in HPC

• use performance analysis tools for profiling and tracing

• do not spend effort in *Do-It-Yourself* solutions, e.g. like printf-debugging

• use tracing tools with some precautions
  - overhead
  - data volume