PAPI and Hardware Performance Analysis Tools

Center for Parallel Computers, Royal Institute of Technology
Performance Workshop, May 26th, 2003

Philip Mucci, Research Consultant
Innovative Computing Laboratory/UTK
Performance Evaluation Research Center/LBNL
mucci@cs.utk.edu
The PAPI Interface

PAPI provides two standardized APIs to access the underlying performance counter hardware

- A low level interface designed for tool developers and expert users.
- The high level interface is for application engineers.
Overview

- PAPI
  - Quick Overview
  - 3.0 Feature Outline
- Performance Analysis Tools
- Trends
PAPI Goal

• To understand why the application performs as it does.
  – Optimize the application's performance.
  – Evaluate the algorithms efficiency.
  – Generate an application signature.
  – Develop a performance model.

• Data is NOT PORTABLE, but the interface is...
Overview of Hardware Counters

- Small number of registers dedicated for performance monitoring functions.
  1. AMD Athlon, 4 counters
  2. Pentium <= III, 2 counters
  3. Pentium IV, 18 counters
  4. IA64, 4 counters
  5. Alpha 21x64, 2 counters
     - Power 3, 8 counters
     - Power 4, 8 counters
     - UltraSparc II, 2 counters
     - MIPS R14K, 2 counters
Itanium 1 Block Diagram
Itanium 2 Block Diagram
Itanium 2 Block Diagram
PDC Itanium 2 System Architecture
Importance of Optimization

Example: Speed up from Static Compiler Optimization on Itanium-1 in 2002 (SpecInt)
PAPI Implementation

Portable Layer

- PAPI Low Level
- PAPI Machine Dependent Substrate
- Kernel Extension
- Operating System
- Hardware Performance Counters

Machine Specific Layer

- PAPI High Level

Tools
Preset Events

- Proposed standard set of event names deemed most relevant for application performance tuning
- No standardization of the actual definition
- Mapped to native events on a given platform
• PAPI supports approximately 100 preset events.
  – Preset events are mappings from symbolic names to machine specific definitions for a particular hardware event.
    • Example: PAPI_TOT_CYC
  – PAPI also supports presets that may be derived from the underlying hardware metrics
    • Example: PAPI_L1_DCM
Test case 8: Available events and hardware information.

<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>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>No</td>
<td>No</td>
<td>Level 3 data cache misses</td>
</tr>
<tr>
<td>PAPI_L3_ICM</td>
<td>0x80000005</td>
<td>No</td>
<td>No</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_SNP</td>
<td>0x80000009</td>
<td>No</td>
<td>No</td>
<td>Requests for a snoop</td>
</tr>
<tr>
<td>PAPI_CA_SHR</td>
<td>0x8000000a</td>
<td>No</td>
<td>No</td>
<td>Requests for shared cache line</td>
</tr>
<tr>
<td>PAPI_CA_CLN</td>
<td>0x8000000b</td>
<td>No</td>
<td>No</td>
<td>Requests for clean cache line</td>
</tr>
<tr>
<td>PAPI_CA_INV</td>
<td>0x8000000c</td>
<td>No</td>
<td>No</td>
<td>Requests for cache line inv.</td>
</tr>
</tbody>
</table>
Native Events

• PAPI supports native events:
  – An event countable by the CPU can be counted even if there is no matching preset PAPI event.
  – The developer uses the same API as when setting up a preset event, but a CPU-specific bit pattern is used instead of the PAPI event definition
High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- As easy to use as IRIX calls
- Requires no setup code
- Restrictions:
  - Allows only PAPI presets
  - Not thread safe
  - Only aggregate counters
Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- Approximately 60 functions
  
  (http://icl.cs.utk.edu/projects/papi/files/html_man/papi.html#4)
- Thread-safe (SMP, OpenMP, Pthreads)
- Supports both presets and native events
Low-level Functionality

• API Calls for:
  – Counter multiplexing
  – Callbacks on user defined overflow value
  – SVR4 compatible profiling
  – Processor information
  – Address space information
  – Static and dynamic memory information
  – Accurate and low latency timing functions
  – Hardware event inquiry functions
  – Eventset management functions
  – Simple locking operations
PAPI and Multiplexing

• Multiplexing allows simultaneous use of more counters than are supported by the hardware.
  – This is accomplished through timesharing the counter hardware and extrapolating the results.

• Users can enable multiplexing with one API call and then use PAPI normally.
Interrupts 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 hardware level, PAPI emulates this in software at the user level.
  – Code must run a reasonable length of time.
Hardware Profiling

• On overflow of hardware counter, dispatch a signal/interrupt.
• Get the address at which the code was interrupted.
• Store counts of interrupts for each address.
• Vendor/GNU prof and gprof (-pg and –p compiler options) use interval timers.
Results of Statistical Profiling

- The result: A probabilistic distribution of where the code spent its time and why.
For More Information

  - Software and documentation
  - Reference materials
  - Papers and presentations
  - Third-party tools
  - Mailing lists
PAPI 2.3.4 Release

• Additional Platforms
  – IBM PPC604, 604e, Power 3
  – Intel x86
  – Sun UltraSparc I/II/III
  – SGI MIPS R10K/R12K/R14K
  – Compaq Alpha 21164/21264 with DADD/DCPI
  – Itanium
  – Itanium 2
  – Power 4
  – AIX 5, Power 3, 4

• Enhancements
  – Static/dynamic memory info
  – Multiplexing improvements
  – Lots of bug fixes
PAPI 3.0

• Using lessons learned from years earlier
  – Substrate code: 90% used only 10% of the time

• I Want to formalize the API during this visit!

• Redesign for:
  – Robustness
  – Feature Set
  – Elegance
  – Portability
Some PAPI 3.0 Features

- Multiway multiplexing
  - Use all available counter registers instead of one per time slice.
- Superb performance
  - Example: On Pentium 4, a PAPI_read() costs 230 cycles, while register access alone costs 100 cycles.
- Full native and programmable event support
  - Thresholding
  - Instruction matching
  - Per event counting domains
• Third-party interface
  – Allows PAPI to control counters in other processes
• Internal timer/signal/thread abstractions
  – Support signal forwarding
• Additional internal layered API to support robust extensions
PAPI 3.0 Features 3

- Advanced profiling interface
  - Support profiling on multiple counters
  - Support hardware or operating system assisted profiling

- New sampling interface
  - P4, IA64 provide Event Address Registers of BTB misses, Cache misses, TLB misses, etc...

- Revised memory API
  - Process footprint
PAPI 3.0 Features 4

- System-wide and process wide counting implementation
- High level API made thread safe
- New language bindings
  - Java
  - Lisp
  - Matlab
PAPI 3.0 Release Targets

- First release expected Summer, 2003
- Additional platforms
  - Cray X-1
  - AMD Opteron/K8
  - Nec SX-6
  - Blue Gene (BG/L)
PAPI Tools
Paradyn from U. Wisconsin

- From Barton Miller’s Group
- DynInst based dynamic discovery of bottlenecks
- Different visualization plugins
- Supports all forms of parallelism
- New version will do discovery based on hardware metrics
  - Memory stall time
  - Cache misses
Tuning and Analysis Utilities

- From Allen Maloney’s Group at U. Oregon
- Source or binary based
- Different visualization plugins
- Supports all forms of parallelism
- Integration with Vampir
Vampir (NAS Parallel Benchmark – LU)
Vampir v3.x: HPM Counter

- **Counter Timeline Display**

- **Process Timeline Display**
SvPablo from UIUC

- Source based instrumentation of loops and function calls
- Supports serial and MPI jobs
- Freely available
- Rough F90 parser
Vprof from Sandia National Laboratory

- Based on statistical sampling of the hardware counters
- Must instrument the source
- Ported to other architectures for generalized use
- Parallel codes with some modification
- Not actively supported

http://aros.ca.sandia.gov/~cljanss/perf/vprof
HPCToolkit from Rice University

• Tools for:
  – Collecting raw statistical profiles
  – Conversion of profiles into platform independent XML
  – Synthesizing browsable representations that correlate metrics with source code

• http://www.hipersoft.rice.edu/hpctoolkit
• Collection: papirun/hvprof, equivalent to SGI's "ssrun"
• Loop/CFG recovery from binary: bloop
• Data formatting: papiprof
• Data display and exploration: hpcview
• Call stack profiles: csprof
• Data is aggregated into an XML database
• HPCView is a Java applet that generates dynamic HTML
HPCView Screenshot

May 27th, 2003
Trapper in PAPI Tools Tree

- Tool that allows the user to write functions that get executed at:
  - Process Creation/Deletion
  - Thread Creation/Deletion
- Actually, any function can be “preempted”.
- The object code of the application isn’t modified.
- Works by “preloading” special shared libraries and overloading function calls in cooperation with the run-time linker.
PerfSuite from NCSA

- Libraries and tools for machine information, memory information, aggregate counts, derived metrics and statistical profiles
- Targeted for x86 and IA64 systems
- http://perfsuite.ncsa.uiuc.edu
PerfSuite Tools

- **psinv**: Gather information on a processor and the PAPI events it supports
- **psrun**: Collection of aggregate/derived counts or statistical profiles of unmodified binaries
- **psprocess**: Formatting and output of psrun data into text or HTML
Psprocess Example Output

PerfSuite Hardware Performance Report

<table>
<thead>
<tr>
<th>Metric Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctly executed instructions per cycle</td>
<td>1.777</td>
</tr>
<tr>
<td>Incorrectly executed instructions per cycle</td>
<td>0.116</td>
</tr>
<tr>
<td>Floating-point percentage of all executed instructions</td>
<td>9.210%</td>
</tr>
<tr>
<td>Incorrectly executed floating-point instructions</td>
<td>0.143</td>
</tr>
<tr>
<td>Correctly executed floating-point instructions</td>
<td>1.134</td>
</tr>
<tr>
<td>Data cache miss per instruction</td>
<td>0.012</td>
</tr>
<tr>
<td>Correctly executed instruction that hit data cache</td>
<td>0.116</td>
</tr>
<tr>
<td>LLC miss rate per instruction</td>
<td>0.006</td>
</tr>
<tr>
<td>Data cache miss rate</td>
<td>0.006</td>
</tr>
<tr>
<td>Branch mispredict per instruction</td>
<td>0.122</td>
</tr>
<tr>
<td>Branch mispredict rate</td>
<td>0.122</td>
</tr>
<tr>
<td>Floating-point instructions that correctly predicted branches</td>
<td>0.116</td>
</tr>
<tr>
<td>Floating-point instructions that correctly predicted branches</td>
<td>0.116</td>
</tr>
<tr>
<td>L1 data cache hit rate</td>
<td>0.477</td>
</tr>
<tr>
<td>L2 data cache hit rate</td>
<td>0.71</td>
</tr>
<tr>
<td>Floating-point instructions that correctly predicted branches</td>
<td>0.477</td>
</tr>
<tr>
<td>Floating-point instructions that correctly predicted branches</td>
<td>0.71</td>
</tr>
<tr>
<td>Bundles Unrolled (L1 cache)</td>
<td>824</td>
</tr>
<tr>
<td>Bundles Staged (L1 cache)</td>
<td>1,188</td>
</tr>
<tr>
<td>Bundles in Use (L1 cache)</td>
<td>1,464</td>
</tr>
<tr>
<td>Percentage of cache misses per instruction</td>
<td>36.13%</td>
</tr>
<tr>
<td>Percentage of cache hits per instruction</td>
<td>63.87%</td>
</tr>
<tr>
<td>MOPS (CPU cycles)</td>
<td>1,188</td>
</tr>
<tr>
<td>MOPS (CPU cycles)</td>
<td>1,188</td>
</tr>
<tr>
<td>MOPS (CPU cycles)</td>
<td>1,188</td>
</tr>
<tr>
<td>Processed instructions</td>
<td>55.48%</td>
</tr>
</tbody>
</table>
Psrun Statistical Profile Example Output

Function Summary

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1839543</td>
<td>35.01%</td>
<td>35.01%</td>
<td>inl3130</td>
</tr>
<tr>
<td>541829</td>
<td>10.31%</td>
<td>45.32%</td>
<td>ns5_core</td>
</tr>
<tr>
<td>389741</td>
<td>7.42%</td>
<td>52.74%</td>
<td>inl0100</td>
</tr>
<tr>
<td>355349</td>
<td>6.76%</td>
<td>59.51%</td>
<td>spread_q_bsplines</td>
</tr>
<tr>
<td>213172</td>
<td>4.06%</td>
<td>63.56%</td>
<td>gather_f_bsplines</td>
</tr>
<tr>
<td>200546</td>
<td>3.82%</td>
<td>67.38%</td>
<td>do_longrange</td>
</tr>
<tr>
<td>182691</td>
<td>3.48%</td>
<td>70.86%</td>
<td>make_bsplines</td>
</tr>
<tr>
<td>149924</td>
<td>2.85%</td>
<td>73.71%</td>
<td>ewald_LRcorrection</td>
</tr>
<tr>
<td>112883</td>
<td>2.15%</td>
<td>75.86%</td>
<td>inl3100</td>
</tr>
<tr>
<td>105317</td>
<td>2.00%</td>
<td>77.86%</td>
<td>solve_pme</td>
</tr>
<tr>
<td>92257</td>
<td>1.76%</td>
<td>79.62%</td>
<td>flincs</td>
</tr>
</tbody>
</table>
PerfSuite Libraries

- **libperfsuite**: Provides simple wrappers for machine information, process memory usage and high-precision timing
- **libpshwpc**: Provides simple wrappers that are used to collect hardware performance data
program mxm
include 'fperfsuite.h'
c Initialize libpshwpc
    call PSF_hwpc_init(ierr)
c Start performance counting using libpshwpc
    call PSF_hwpc_start(ierr)
c Stop hardware performance counting and write the
results to a file named 'perf.XXXXX' (XXXXX will be
c replaced by the process ID of the program)
    call PSF_hwpc_stop('perf', ierr)
c Shutdown use of libpshwpc and the underlying libraries
    call PSF_hwpc_shutdown(ierr)

• Environment variables and XML input file dictate what
gets measured
HPMToolkit from IBM ACTC

- Command line utility to gather aggregate counts.
  - PAPI version has been tested on IA32 & IA64
  - User can manually instrument code for more specific information
  - Reports derived metrics like SGI’s perfex
- Libhpm for manual instrumentation
- Hpmviz is a GUI to view resulting data

http://www.ncsa.uiuc.edu/UserInfo/Resources/Software/Tools/HPMToolkit
#include "libhpm.h"
hpmInit( tasked, "my program" );
hpmStart( 1, "outer call" );
do_work();
hpmStart( 2, "computing meaning of life" );
do_more_work();
hpmStop( 2 );
hpmStop( 1 );
hpmTerminate( taskID );
ToolGear Overview

- Dynamic instrumentation and analysis suite from LLNL
- Based on DPCL from IBM
  - Tested only on AIX
- Qt Front end can theoretically accept data from any source
- GUI displays instrumentable points
- Instrumented points update display with data in real time
- http://www.llnl.gov/CASC/tool_gear
```c
for(i = 0; i < 1; i++) {
    /* Tiled */
    init_array[X];
    printf("Doing %d flops of tiled test\n", FLOPS);
    do_tiled_cache_test(); (FLOPS);

    /* Untiled */
    init_array[X];
    printf("Doing %d flops of untiled test\n", FLOPS);
    do_unsliced_cache_test(); (FLOPS);

    /* Indexed */
    init_array[X];
    printf("Doing %d flops of indexed address test\n", FLOPS);
    do_indirect_address_test(); (FLOPS);
    printf("Done with series %s", i);
    fprintf(stderr, "Done with series %s", i);
}
```

8 data pts: Max 0.93747 (Rank 0; Thread 1) Min 0.937464 (7/1) Mean 0.937468 StdDev 1.90709e-06 Sum 7.49974
DynaProf

• A portable tool to dynamically instrument serial and parallel programs for the purpose of performance analysis.
• Simple and intuitive command line interface like GDB.
• Java/Swing GUI.
• Instrumentation is done through the run-time insertion of function calls to specially developed performance probes.
Why the “Dyna” in DynaProf?

• Instrumentation:
  – Functions are contained in shared libraries.
  – Calls to those functions are generated at run-time.
  – Those calls are dynamically inserted into the program’s address space.

• Built on DynInst and DPCL

• Can choose the mode of instrumentation, currently:
  – Function Entry/Exit
  – Call site Entry/Exit
  – One-shot
DPCL vs. DynInst

- Parallel framework based on DynInst
- Asynch./Sync. operation
- Functions for getting data back to tool
- Integrated with POE

- Available on all HPC platforms (and Windows)
- Breakpoints
- Arbitrary ins. points
- Full Loop, CFG and Basic Block decoding
A Brief History of Dynamic Instrumentation

- Popularized by James Larus with EEL: An Executable Editor Library at U. Wisc.
  - http://www.cs.wisc.edu/~larus/eel.html
- Technology matured by Dr. Bart Miller and (now Dr.) Jeff Hollingsworth at U. Wisc.
  - DynInst Project at U. Maryland
    - http://www.dyninst.org/
  - IBM’s DPCL: A Distributed DynInst
    - http://oss.software.ibm.com/dpcl/
DynaProf Goals

• Make collection of run-time performance data easy by:
  – Avoiding instrumentation and recompilation
  – Avoiding perturbation of compiler optimizations
  – Providing complete language independence
  – Allowing multiple insert/remove instrumentation cycles

No source code required!
DynaProf Goals 2

- Using the same tool with different probes
- Providing useful and meaningful probe data
- Providing different kinds of probes
- Allowing custom probe development

Make collection of run-time performance data easy by:

No source code required!
Dynaprof Probes

- **perfmeterprobe**
  - Visualize hardware event rates in “real-time”

- **papiprobe**
  - Measure any combination of PAPI presets and native events

- **wallclockprobe**
  - Highly accurate elapsed wallclock time in microseconds.

- **The latter 2 probes report:**
  - Inclusive
  - Exclusive
  - 1 Level Call Tree
Sample DynaProf Session

$. / dynaprof
(dynaprof) load tests/swim
(dynaprof) list
DEFAULT_MODULE
swim.F
libm.so.6
libc.so.6
(dynaprof) list swim.F
MAIN__
  initial_
  calc1_
  calc2_
  calc3z_
  calc3_
(dynaprof) list swim.F MAIN__
  Entry
    Call s_wsle
    Call do_lio
    Call e_wsle
    Call s_wsle
    Call do_lio
    Call e_wsle
(dynaprof) use probes/papiprobe
Module papiprobe.so was loaded.
Module libpapi.so was loaded.
Module libperfctr.so was loaded.
(dynaprof) instr module swim.F calc*
swim.F, inserted 4 instrumentation points
(dynaprof) run
papiprobe: output goes to
/home/mucci/dynaprof/tests/swim.1671
DynaProf Probe Design

• Probes export a few functions with loosely standardized interfaces.
• Easy to roll your own.
  – If you can code a timer, you can write a probe.
• DynaProf detects thread model.
• Probes dictate how the data is recorded and visualized.
Threads and Dynaprof Probes

• For threaded code, use the same probe!
• Dynaprof detects threads and loads a special version of the probe library.
• Each probe specifies what to do when a new thread is discovered.
• Each thread gets the same instrumentation.
PAPI Probe

- Can count any PAPI preset or Native event accessible through PAPI
- Can count multiple events
- Supports PAPI multiplexing
- Supports multithreading
  - AIX: SMP, OpenMP, Pthreads
  - Linux: SMP, OpenMP, Pthreads
Wallclock Probe

- Counts microseconds using RTC
- Supports multithreading
  - AIX: SMP, OpenMP, Pthreads
  - Linux: SMP, OpenMP, Pthreads
Reporting Probe Data

• The wallclock and PAPI probes produce very similar data.

• Both use a parsing script written in Perl.
  – wallclockrpt <file>
  – papiproberpt <file>

• Produce 3 profiles
  – Inclusive: \( T_{function} = T_{self} + T_{children} \)
  – Exclusive: \( T_{function} = T_{self} \)
  – 1-Level Call Tree: \( T_{child} = \text{Inclusive } T_{function} \)
(dynaprof) use probes/papiprobe PAPI_TOT_CYC, PAPI_TOT_INS
Module papiprobe.so was loaded.
Module libpapi.so was loaded.
Module libperfctr.so was loaded.
(dynaprof) instr function swim.F calc*
Swim.F, inserted 3 instrumentation points
(dynaprof) instr
calc1_
calc2_
calc3_
calc3z_
Swim Benchmark: Cycles & Instructions
Swim Benchmark: Instructions per Cycle

calc2     0.59
calc1     0.53
calc3     0.46
DynaProf GUI

- Displays module tree for instrumentation
- Simple selection of probes and instrumentation points
- Single-click execution of common DynaProf commands
- Coupling of probes and visualizers (e.g. Perfometer)
- Does not work well!
DynaProf GUI Screenshot
Dynaprof Status

• It's a bit rough

• Supported Platforms
  – Using DynInst 3.0
    • Linux 2.x
    • AIX 4.3/5?
    • Solaris 2.8
    • IRIX 6.x
  – Using DPCL (formal MPI support)
    • AIX 4.3
    • AIX 5

• Includes:
  – Java/Swing GUI
  – User’s Guide
  – Probe libraries

• Available as a development snapshot from:

http://www.cs.utk.edu/~mucci/dynaprof
Dynaprof Future

- Port to DynInst API 4.0 (Released RSN)
- IA64 Support
- New instrumentation point support:
  - Object
  - Instance
  - Loop
  - Basic Block
  - Arbitrary
- Breakpoints
- Support for programs that dynamically load modules during run-time. (Mozilla)
- Integration with TAU
Performance Tool Trends

- Most of the infrastructure now exists.
- Many sites are “rolling their own”.
- Can one size fit all?
- 2 types of tools evolving:
  - Simple: papiprof
  - Comprehensive: TAU
Performance Database

- Database of all relevant information regarding the performance of a code.
  - Source code structure
  - Transformations performed during optimization
  - Static and dynamic memory allocation information
  - Derived data types, etc...

- Examples:
  - TAU PDT: Program Database Toolkit
  - HPC Tools: XML Database
  - ToolGear

- This data can be quite large! Remember MPI traces?
Some problems to be solved

• How do we get the data out of the threads/processors/nodes/application and back to the user? Maybe...
  – DPCL for all DynInst: LANL and me
• How do we correlate performance data from optimized code to the source?
• We want to understand all aspects of a program’s performance. What about behaviour over time?
Phase Profiling

- Statistical profiling is often static
  - Gprof, Quantify, Speedshop, Workshop, Tprof, etc...
- Applications vs. kernels have distinct phases.
  - Initialization
  - Data input
    - Compute
    - Communicate
    - Repeat
  - Data output
  - Finalization
• Workloads on the hardware are most often periodic.

• More open questions:
  – How do we process, visualize and understand this data in a scalable fashion?
  – Can we use this data to optimize an application in the temporal domain?
  – Can we parameterize this data against \((t)\) for performance models?