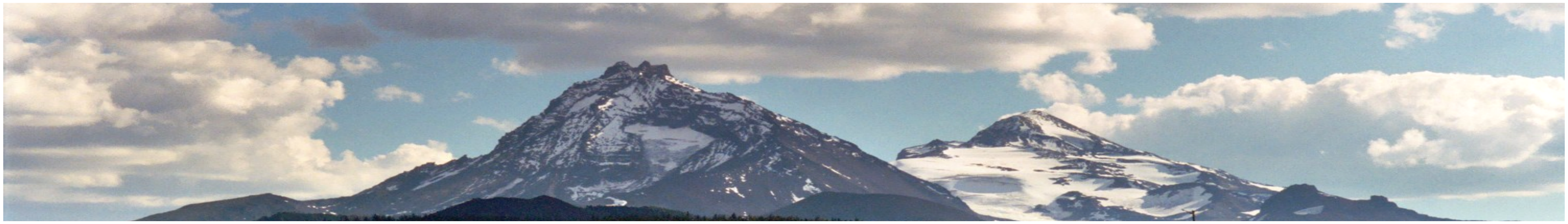


Lecture 3: Technologies

Allen D. Malony

Department of Computer and Information Science

UNIVERSITY OF OREGON

Performance Tools and Technologies

- It is never the case that performance tools are developed from scratch
- They depend on a range of technologies that can themselves be significant engineering efforts
  - Even simple conceptual things can be hard
- Most technologies deal with how to observe performance metrics or state

"If I have seen further it is by standing on the shoulders of giants."

- Sir Isaac Newton

Technologies

- Timers
- Counters
- Instrumentation
  - Source (PDT)
  - PMPI
  - Compiler instrumentation
  - Binary
    - Dyninst
    - PEBIL
    - MAQAO
  - Runtime Interfaces
- Program Address Resolution: Binutils
- Stack Walking
  - StackwalkerAPI
  - Libunwind
  - Backtrace
- Heterogeneous (accelerator) timers and counters

Time

- How is time measured in a computer system?
- How do we derive time from a clock?
- What clock/time technologies are available to a measurement system?
- How are clocks synchronized in a parallel computer in order to provide a "global time" common between nodes?
- Different technologies are available
  - Issues of resolution and accuracy

Timer: gettimeofday()

- UNIX function
- Returns wall-clock time in seconds and microseconds
- Actual resolution is hardware-dependent
- Base value is 00:00 UTC, January 1, 1970
- Some implementations also return the timezone

```
#include <sys/time.h>

struct timeval tv;
double walltime;  /* seconds */

gettimeofday(&tv, NULL);
walltime = tv.tv_sec + tv.tv_usec * 1.0e-6;
```

Timer: clock_gettime()

- POSIX function
- For *clock_id* CLOCK_REALTIME it returns wall-clock time in seconds and nanoseconds
- More clocks may be implemented but are not standardized
- Actual resolution is hardware-dependent

```
#include <time.h>

struct timespec tv;
double walltime;  /* seconds */

Clock_gettime(CLOCK_REALTIME, &tv);
walltime = tv.tv_sec + tv.tv_nsec * 1.0e-9;
```

Timer: getrusage()

- UNIX function
- Provides a variety of different information
  - Including user time, system time, memory usage, page faults, and other *resource use* information
  - Information provided system-dependent!

```
#include <sys/resource.h>

struct rusage ru;
double usrtime;  /* seconds */
int memused;

getrusage(RUSAGE_SELF, &ru);
usrtime = ru.ru_utime.tv_sec +
          ru.ru_utime.tv_usec * 1.0e-6;
memused = ru.ru_maxrss;
```

Timer: Others

- MPI provides portable MPI wall-clock timer

```
#include <mpi.h>
double walltime;   /* seconds */

walltime = MPI_Wtime();
```

  - Not required to be consistent/synchronized across ranks!
- OpenMP 2.0 also provides a library function

```
#include <omp.h>
double walltime;   /* seconds */

walltime = omp_get_wtime();
```

- Hybrid MPI/OpenMP programming?
  - Interactions between both standards (yet) undefined

Timer: Others

- Fortran 90 intrinsic subroutines
  - cpu_time()
  - system_clock()
- Hardware counter libraries typically provide "timers" because underlying them are cycle counters
  - Vendor APIs
    - PMAPI, HWPC, libhpm, libpfm, libperf, …
  - PAPI (Performance API)

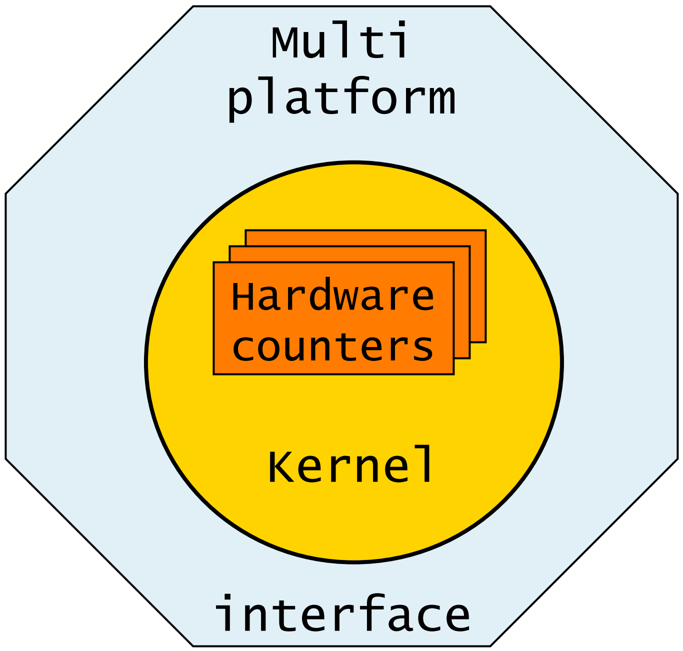
UNIVERSITY OF OREGON

What Are Performance Counters

- Extra processor logic inserted to count specific events
- Updated at every cycle (or when some event occurs)
- Strengths
  - Non-intrusive
  - Very accurate
  - Low overhead
- Weaknesses
  - Provides only hard counts
  - Specific for each processor
  - Access is not appropriate for the end user
    - nor is it well documented
  - Lack of standard on what is counted

Hardware Counter Issues

- Kernel level
  - Handling of overflows
  - Thread accumulation
  - Thread migration
  - State inheritance
  - Multiplexing
  - Overhead
  - Atomicity
- Multi-platform interfaces
  - Performance API (*PAPI*)
    - University of Tennessee, USA
  - Lightweight Performance Tools (*LIKWID*)
    - University of Erlangen, Germany



UNIVERSITY OF OREGON

Hardware Measurement

- Typical measured events account for:
  - Functional units status
    - float point operations
    - fixed point operations
    - load/stores
  - Access to memory hierarchy
  - Cache coherence protocol events
  - Cycles and instructions counts
  - Speculative execution information
    - instructions dispatched
    - branches mispredicted

Hardware Metrics

- **Typical hardware counter** | **Useful derived metrics**

Typical hardware counter	Useful derived metrics
Cycles / Instructions	IPC
Floating point instructions	FLOPS
Integer instructions	computation intensity
Load/stores	instructions per load/store
Cache misses	load/stores per cache miss
Cache misses	cache hit rate
Cache misses	loads per load miss
TLB misses	loads per TLB miss

- Derived metrics allow users to correlate the behavior of the application to hardware components
- Define threshold values acceptable for metrics and take actions regarding optimization when below/above thresholds

Accuracy Issues

- Granularity of the measured code
  - If not sufficiently large enough, overhead of the counter interfaces may dominate
  - Mainly applies to time
- Pay attention to what is not measured:
  - Out-of-order processors
  - Sometimes speculation is included
  - Lack of standard on what is counted
    - microbenchmarks can help determine accuracy of the hardware counters
- Impact of measurement on counters themselves
  - Typically less of an issue

Hardware Counters Access on Linux

- Linux had not defined an out-of-the-box interface to access the hardware counters!
  - Linux Performance Monitoring Counters Driver (PerfCtr) by Mikael Pettersson from Uppsala X86 + X86-64
    - needs kernel patching!
      http://user.it.uu.se/~mikpe/linux/perfctr/
  - Perfmon by Stephane Eranian from HP – IA64
    - it was being evaluated to be added to Linux
      http://www.hpl.hp.com/research/linux/perfmon/
- Linux 2.6.31
  - Performance Counter subsystem provides an abstraction of special performance counter hardware registers

Utilities to Count Hardware Events

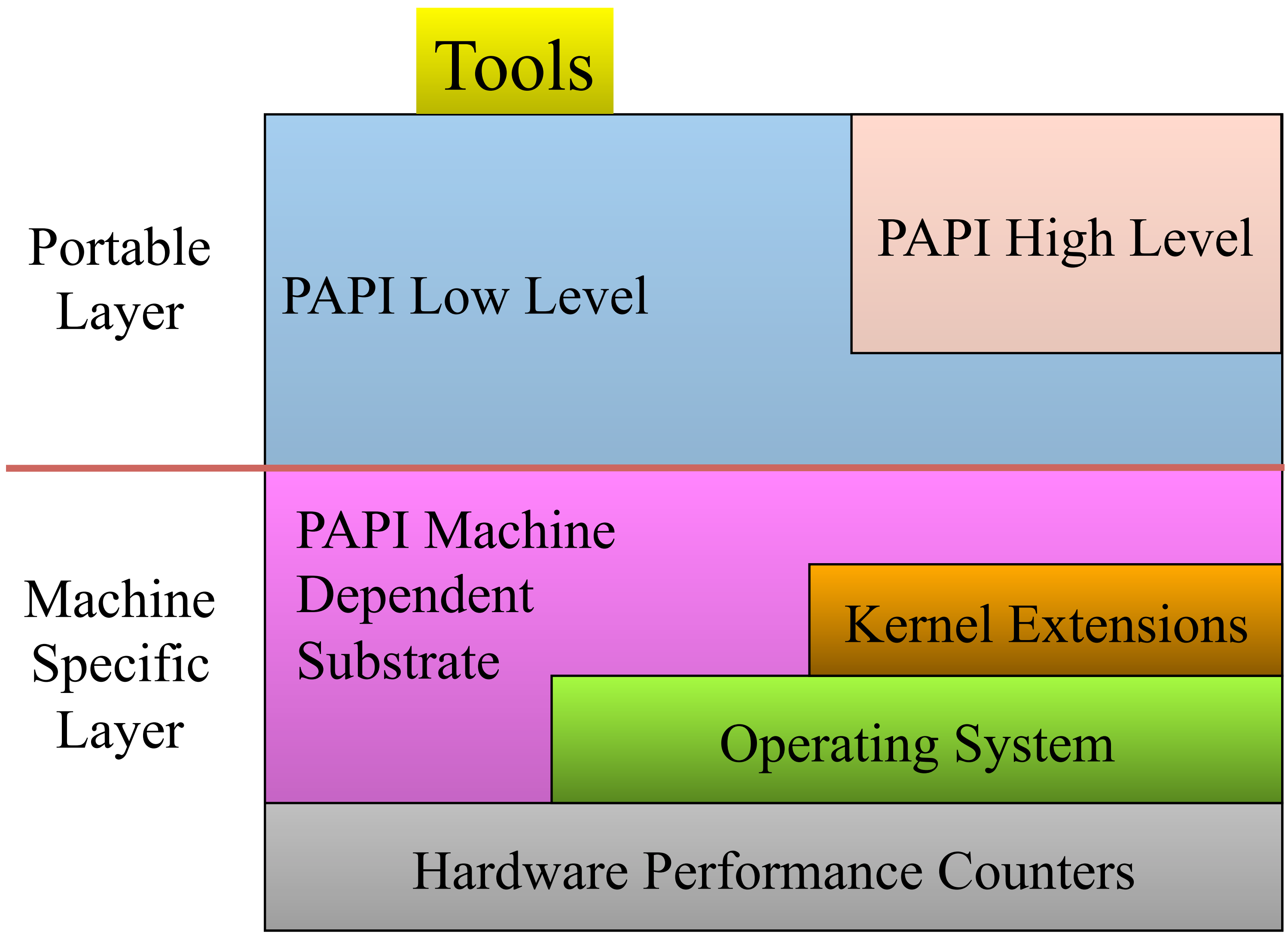
- There are utilities that start a program and at the end of the execution provide overall event counts
  - *hpmcount* (IBM)
  - *CrayPat* (Cray)
  - *pfmon* from HP (part of Perfmon for AI64)
  - *psrun* (NCSA)
  - *cputrack*, *har* (Sun)
  - *perfex*, *ssrun* (SGI)
  - *perf* (Linux 2.6.31)

PAPI – Performance API

- Middleware to provide a consistent and portable API for the performance counter hardware in microprocessors
- Countable events are defined in two ways:
  - Platform-neutral *preset* events
  - Platform-dependent *native* events
- Presets can be derived from multiple native events
- Two interfaces to the underlying counter hardware:
  - *High-level* interface simply provides the ability to start, stop and read the counters for a specified list of events
  - Low-level interface manages hardware events in user defined groups called *EventSets*
- Events can be multiplexed if counters are limited

http://icl.cs.utk.edu/papi/

PAPI Architecture



PAPI Predefined Events

- Common set of events deemed relevant and useful for application performance tuning (wish list)
  - papiStdEventDefs.h
  - Accesses to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit and pipeline status
  - Run PAPI papi_avail utility to determine which predefined events are available on a given platform
  - Semantics may differ on different platforms!
- PAPI also provides access to native events on all supported platforms through the low-level interface
  - Run PAPI papi_native_avail utility to determine which predefined events are available on a given platform

papi_avail Utility

- Provides information on what events are available on a particular hardware platform

```
% papi_avail -h
This is the PAPI avail program.
It provides availability and detail information
for PAPI preset and native events.  Usage:

    papi_avail [options] [event name]
    papi_avail TESTS_QUIET

Options:
  -a            display only available PAPI preset events
  -d            display PAPI preset event info in detailed format
  -e EVENTNAME  display full detail for named preset or native event
  -h            print this help message
  -t            display PAPI preset event info in tabular format (default)
```

High Level API

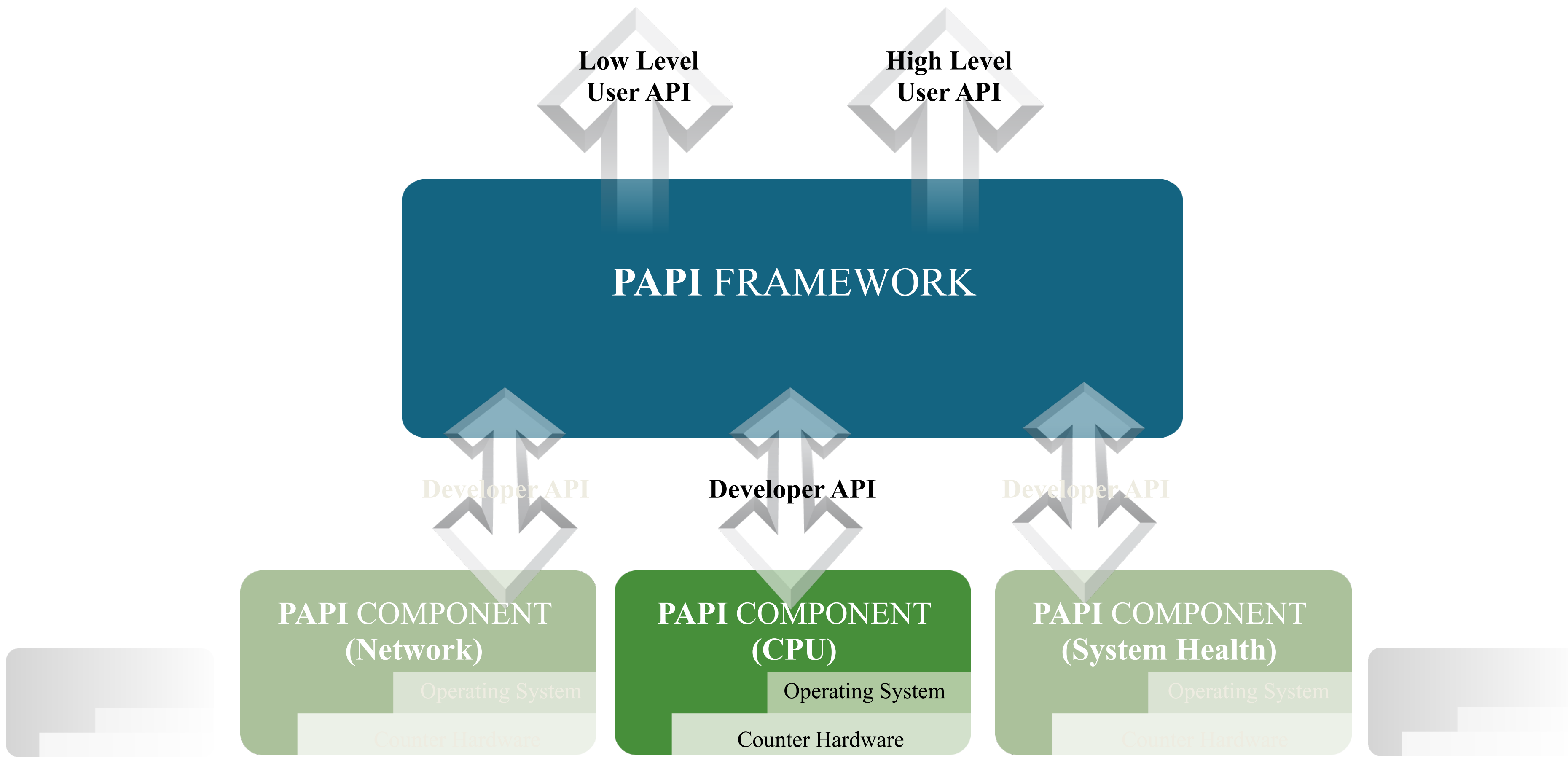
- Meant for application programmers wanting simple but accurate measurements
- Calls the lower level API
- Allows only PAPI preset events
- Eight functions:
  - *PAPI_num_counters*
  - *PAPI_start_counters, PAPI_stop_counters*
  - *PAPI_read_counters*
  - *PAPI_accum_counters*
  - *PAPI_flops*
  - *PAPI_flips, PAPI_ipc* (New in Version 3.x)
- Not thread-safe (Version 2.x)

Low Level API

- Increased efficiency and functionality over the high level PAPI interface
- 54 functions
- Access to native events
- Obtain information about the executable, the hardware, and memory
- Set options for multiplexing and overflow handling
- System V style sampling (profil())
- Thread safe

Component PAPI

- Developed for the purpose of extending counter sets while providing a common interface



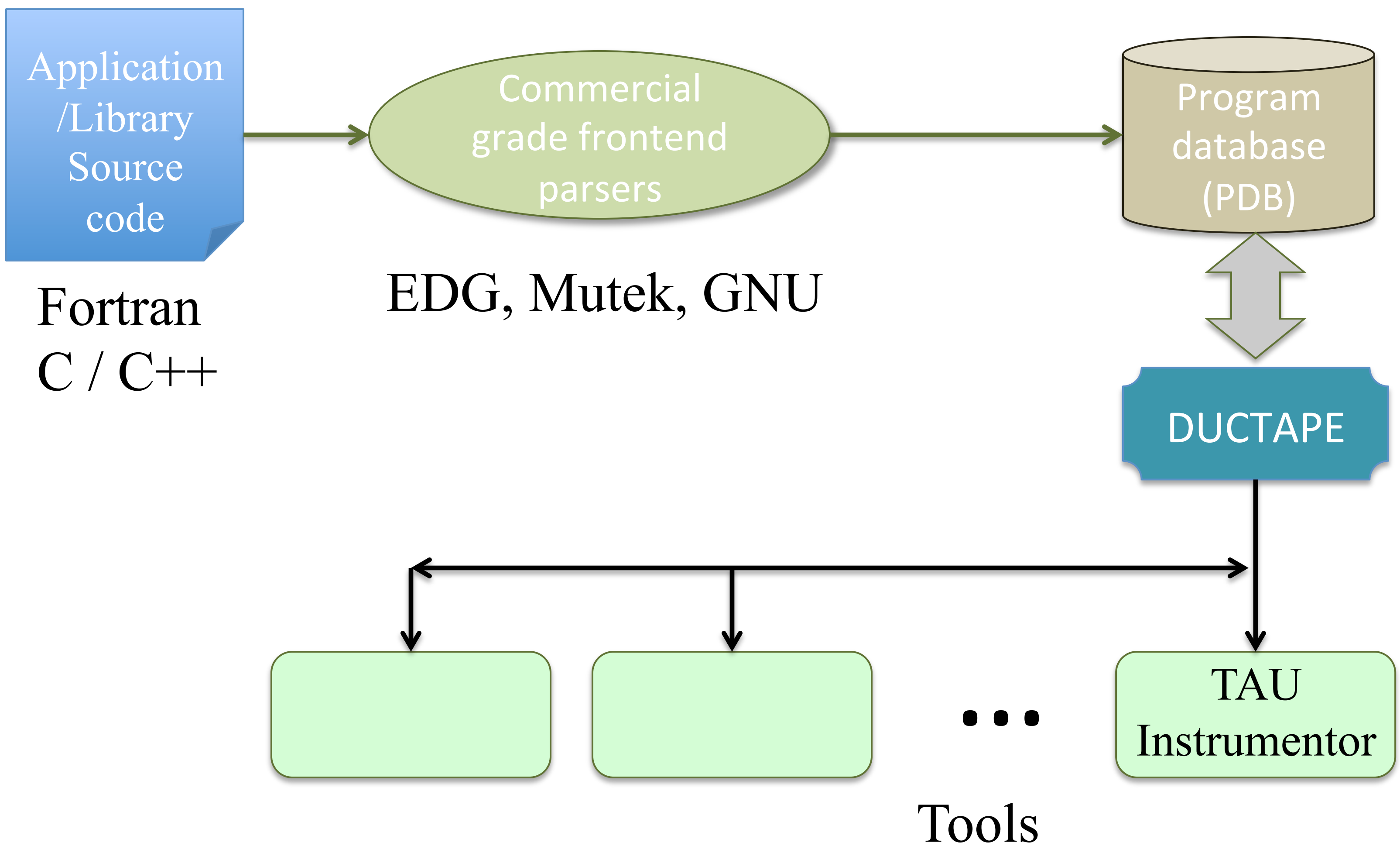
Source Instrumentation with Timers

- Measuring performance using timers requires instrumentation
  - Have to uniquely identify code region (name)
  - Have to add code for timer start and stop
  - Have to compute delta and accumulate statistics
- Hand-instrumenting becomes tedious very quickly, even for small software projects
- Also a requirement for enabling instrumentation only when wanted
  - Avoids unnecessary overheads when not needed

Program Database Toolkit (PDT)

- University of Oregon, Research Center Juelich (FZJ Germany), Edison Design Group, Inc. (USA), LLNL (USA)
- Automated instrumentation of C/C++, Fortran source code
- Source code parser(s) identify blocks such as function boundaries, loop boundaries, generates a .PDB file for each source file
- Instrumentor uses .PDB file to insert API calls into source code files at block enter/exit, outputs an instrumented code file
- Instrumented source passed to compiler for compilation to object file
- Linker links application with measurement library providing definitions for API calls
- Free download: http://tau.uoregon.edu

PDT Architecture



PMPI – MPI Standard Profiling Interface

- The MPI (Message Passing Interface) standard defines a mechanism for instrumenting all API calls in an MPI implementation
- Each MPI_* function call is actually a weakly defined interface that can be re-defined by performance tools
- Each MPI_* function call eventually calls a corresponding PMPI_* function call which provides the expected MPI functionality
- Performance tools can redefine MPI_* calls

PMPI Example

- Original MPI_Send() definition:

```
int __attribute__((weak))
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm) {
    PMPI_Send(buf, count, datatype, dest, tag, comm);
}
```

- *Possible* Performance tool definition:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm) {
    MYTOOL_Timer_Start("MPI_Send");
    PMPI_Send(buf, count, datatype, dest, tag, comm);
    MYTOOL_Timer_Stop("MPI_Send");
    MYTOOL_Message_Size("MPI_Send", count * sizeof(datatype));
}
```

Compiler Instrumentation

- Modern compilers provide the ability to instrument functions at compile time
- Can exclude files, functions
- GCC example:
  - `-finstrument-functions` parameter
  - Instruments function entry and exit(s)

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

Compiler Instrumentation – Tool Interface

- Measurement libraries have to implement those two functions:

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

- The function and call site pointers are instruction addresses
- How to resolve those addresses to source code locations?
  - Binutils: libbfd, libiberty (discussed later)

Binary Instrumentation

- Source Instrumentation not possible in all cases
  - Exotic / Domain Specific Languages (no parser support)
  - Pre-compiled system libraries
  - Utility libraries without source available
- Binary instrumentation modifies the existing executable and all libraries, adding user-specified function entry/exit API calls
- Can be done once, or as first step of execution

UNIVERSITY OF OREGON

Binary Instrumentation: Dyninst API

- University of Wisconsin, University of Maryland
- Provides binary instrumentation for runtime code patching:
  - Performance Measurement Tools
  - Correctness Debuggers (efficient data breakpoints)
  - Execution drive simulations
  - Computational Steering

http://www.dyninst.org

Binary Instrumentation: PEBIL

- San Diego Supercomputing Center / PMaC group
- Static binary instrumentation for x86_64 Linux
- PEBIL = **P**MaC's **E**fficient **B**inary **I**nstrumentation for **L**inux/x86
- Lightweight binary instrumentation tool that can be used to capture information about the behavior of a running executable

http://www.sdsc.edu/PMaC/projects/pebil.html

PEBIL Design

- Efficiency is priority #1
- Designed around a few use cases
  - Execution counting
  - Memory tracing
- Static binary rewriter
  - Write instrumented + runnable executable to disk
    - keep original behavior intact
    - gather information as a side-effect
  - Instrument once, run many times
  - No instrumentation cost at runtime
  - Code patching (not just-in-time compiled!)

How Binary Instrumentation Works

Original

Basic Block 1
Basic Block 2
Basic Block 3

```
0000c000 <foo>:
 c000:   48 89 7d f8   mov     %rdi,-0x8(%rbp)
 c004:   5e            pop     %rsi
 c005:   75 f8         jne     0xc004
 c007:   c9            leaveq
 c008:   c3            retq
```

Instrumented

```
0000d000 <foo>:
 d000:   e9 de ad be ef jmp    0x1000 # to instrumentation
 d005:   48 89 7d f8    mov    %rdi,-0x8(%rbp)
 d000:   e9 de ad be ef jmp    0x1010 # to instrumentation
 d00a:   5e             pop    %rsi
 d00b:   75 00 00 00 f8 jne    0xd009
 d000:   e9 de ad be ef jmp    0x1020 # to instrumentation
 d00a:   c9             leaveq
 d00b:   c3             retq
```

```
// do stuff
// jump back
```


UNIVERSITY OF OREGON

Use case: Memory Address Collection

- Collect the address of every load/store issued by the application
  - Put addresses in a buffer, process addresses in batch
    - Fewer function calls
    - Less cache pollution

```
for (i = 0; i < n; i++){
    if (cur + 2 > BUF_SIZE) clear_buf();
    buffer[cur + 0] = &(A[i]);
    buffer[cur + 1] = &(B[i]);
    A[i] = B[i];
}
```


UNIVERSITY OF OREGON

Binary Instrumentation: MAQAO

- Modular Assembly Quality Analyzer and Optimizer
- Tool for analyzing and optimizing binary code
- Intel64 and Xeon Phi architectures supported
- Binary release only (for now)

http://maqao.org

MAQAO: Introduction

- Easy install
  - Packaging : ONE (static) standalone binary
  - Easy to embed
- Audience
  - User/Tool developer: analysis and optimization tool
  - Performance tool developer: framework services
  - TAU: tau_rewrite (MIL)
  - ScoreP: on-going effort (MIL)

MAQAO: Architecture

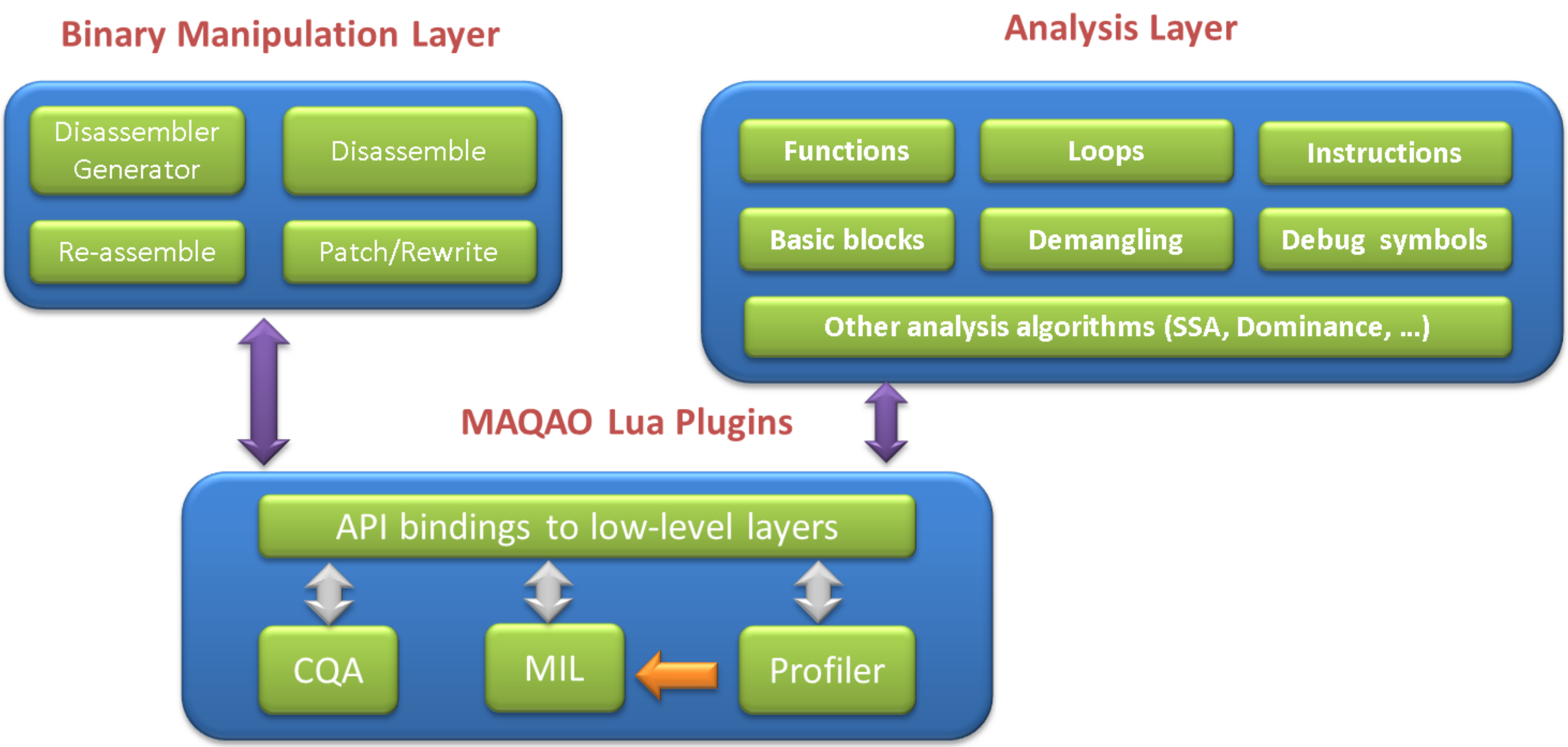


Image source: http://maqao.org

MAQAO: Measurement and Analysis Tool

- Scripting language
  - Lua language : simplicity and productivity
  - Fast prototyping
  - MAQAO Lua API : Access to services
- Built on top of the Framework
- Loop-centric approach
- Output: reports
  - System deals with low level details
  - User gets high level reports

Runtime Measurement Support

- Some runtime systems provide callback mechanisms for function entry / exit or state transitions
  - Java JVM
  - Python
  - Some OpenMP runtimes (Collector API, OMPT)
    - Sun/Oracle, OpenUH, Intel (in development)
- Measurement tools / libraries:
  - implement event handlers for callback functions
  - register with the runtime, are notified when events happen

Periodic Sampling – what is it?

- The application is interrupted after a specified period of time
- Interruption handler queries the program state
- The timer is reset and the process repeats until program termination
- Either at termination or during handler, a statistical profile is constructed
- Sampling theory states that the state (function) sampled the most frequently is the most time consuming state (function)

Periodic Sampling – how to do it?

- ANSI C / POSIX signaling and signal handling
- sigaction()
  - Specify a handler for when a signal is raised
  - Handler has to be signal-safe*
  - Handler gets program state context pointer, including current instruction pointer address and full program stack
- setitimer()
  - Portable (POSIX) interval timer
  - A signal is raised when the timer expires
  - Timers: real time (process-level only), user CPU time, or user CPU + system CPU time counters
  - Undefined behavior for threaded applications
- timer_create() / timer_settime()
  - POSIX function like setitimer(), but with a Linux-specific interval timer with threaded support for real time counter

*POSIX.1-2004 lists the functions that are signal-safe

Address Resolution: GNU Binutils

- Compiler instrumentation and signal handling deal with instruction pointer addresses
- Binutils provides utilities and libraries for looking up addresses and getting function name, source code file and line number
  - Source info available if code compiled with –g
- Iterates over executable and any shared object libraries (if applicable) to find address
- Command line version:
  - addr2line –f –e <executable> <address_1> … <address_n>

http://www.gnu.org/software/binutils/

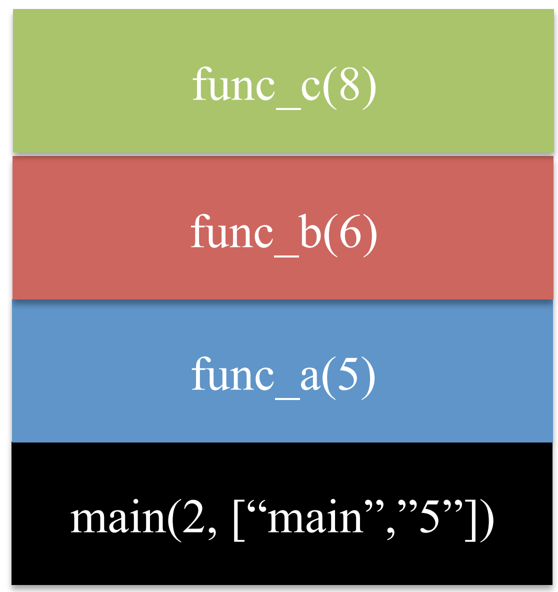
Stack Walking – how did I get here?

- Periodic sampling freezes the program state
- "Walking the stack" answers the question: how did I get here?

```c
int func_c(int c) {
  return (c+3);
}
int func_b(int b) {
  return func_c(b+2);
}
int func_a(int a) {
  return func_b(a+1);
}
int main(int argc, char* argv[]) {
  int in = atoi(argv[1]);
  printf("%d: %d\n", in, func_a(in));
}
```

Signal interrupt here

Current program stack:



UNIVERSITY OF OREGON

Stack Walking: libunwind

- Libunwind defines a portable and efficient C programming interface (API) to determine the call-chain of a program
- Provides the means to manipulate the preserved (callee-saved) state of each call-frame and to resume execution at any point in the call-chain (non-local goto)
- Supports both local (same-process) and remote (across-process) operation.
- Developed and supported by the Free Software Foundation (FSF)
- https://savannah.nongnu.org/projects/libunwind/

Stack Walking: StackWalkerAPI

- University of Wisconsin, University of Maryland
- An API that allows users to collect a call stack and access information about its stack frames
- Support for Linux (x86_64, AMD-64, Power, Power-64), BlueGene/L and BlueGene/P
- http://www.dyninst.org/stackwalker

Stack Walking: Linux Backtrace (libc)

- A *backtrace* is a list of the function calls that are currently active in a thread
- 2 steps: get array of *n* addresses, resolve them to symbols
- Warning! By default, address resolution is not signal-safe
- Signal-safe option writes to file descriptor (no malloc)

http://www.gnu.org/software/libc/manual/html_node/Backtraces.html

http://man7.org/linux/man-pages/man3/backtrace.3.html

```
#include <execinfo.h>

int backtrace(void **buffer, int size);
char **backtrace_symbols(void *const *buffer, int size);
void backtrace_symbols_fd(void *const *buffer, int size, int fd);
```

Backtrace example

```
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>

/* Obtain a backtrace and print it to stdout. */
void print_trace (void) {
    void *array[10];
    size_t size;
    char **strings;
    size_t i;

    size = backtrace (array, 10);
    // not signal safe! bactrace_symbols calls "malloc"
    strings = backtrace_symbols (array, size);
    printf ("Obtained %zd stack frames.\n", size);
    for (i = 0; i < size; i++)
        printf ("%s\n", strings[i]);
    free (strings);
}
```

NVIDIA CUDA Performance Tool Interface

- Use of accelerator and coprocessor hardware also requires access to timer and counting information
- NVIDIA is developing CUDA Performance Tool Interface (CUPTI) to enable the creation of profiling and tracing tools
  - CUPTI support was released with CUDA 4.0
  - Capabilities have steadily improved
  - Current version is released with CUDA 5.x and the just announced CUDA 6
- CUPTI is delivered as a dynamic library

NVIDIA CUPTI APIs

- *Callback API*
  - Interject tool callback code at the entry and exist to each CUDA runtime and driver API call
  - Registered tools are invoked for selected events
- *Counter API*
  - Query, configure, start, stop, read counters on CUDA devices
  - Device-level counter access
- *Activity API*
  - GPU kernel and memory copy timing information is stored in a buffer until a synchronization point is encounter and these timings are recorded by the CPU
  - Synchronization can be either be within a device, stream or occur during some synchronous memory copies and event synchronizations
- Can also get information on kernel registers and instructions

PAPI CUDA Component

- HW performance counter measurement technology for NVIDIA CUDA platform
- Access to HW counters inside the GPUs
- Based on CUPTI (CUDA Performance Tool Interface)
- PAPI CUDA component can provide detailed performance counter info regarding execution of GPU kernel
  - Initialization, device management and context management are enabled by CUDA driver API
  - Domain and event management are enabled by CUPTI
- Names of events are established by the following hierarchy:

  Component.Device.Domain.Event

Portion of CUDA Events on Tesla C870

Event Code	Symbol	Long Description
0x44000000	CUDA.GeForce_GTX_480.gpc0.local_load	# executed local load instructions per warp on a multiprocessor
0x44000001	CUDA.GeForce_GTX_480.gpc0.local_store	# executed local store instructions per warp on a multiprocessor
0x44000002	CUDA.GeForce_GTX_480.gpc0.gld_request	# executed global load instructions per warp on a multiprocessor
0x44000003	CUDA.GeForce_GTX_480.gpc0.gst_request	# executed global store instructions per warp on a multiprocessor
0x44000004	CUDA.GeForce_GTX_480.gpc0.shared_load	# executed shared load instructions per warp on a multiprocessor
0x44000005	CUDA.GeForce_GTX_480.gpc0.shared_store	# executed shared store instructions per warp on a multiprocessor
0x44000006	CUDA.GeForce_GTX_480.gpc0.branch	# branches taken by threads executing a kernel
0x44000007	CUDA.GeForce_GTX_480.gpc0.divergent_branch	# divergent branches within a warp
0x4400000b	CUDA.GeForce_GTX_480.gpc0.active_cycles	# cycles a multiprocessor has at least one active warp
0x4400000c	CUDA.GeForce_GTX_480.gpc0.sm_cta_launched	# thread blocks launched on a multiprocessor
0x4400000d	CUDA.GeForce_GTX_480.gpc0.l1_local_load_hit	# local load hits in L1 cache
0x4400000e	CUDA.GeForce_GTX_480.gpc0.l1_local_load_miss	# local load misses in L1 cache
0x44000011	CUDA.GeForce_GTX_480.gpc0.l1_global_load_hit	# global load hits in L1 cache
0x4400002e	CUDA.Tesla_C870.domain_a.tex_cache_hit	# texture cache misses
0x4400002f	CUDA.Tesla_C870.domain_a.tex_cache_miss	# texture cache hits
0x44000034	CUDA.Tesla_C870.domain_b.local_load	# local memory load transactions
0x44000037	CUDA.Tesla_C870.domain_b.branch	# branches taken by threads executing a kernel
0x44000038	CUDA.Tesla_C870.domain_b.divergent_branch	# divergent branches within a warp
0x44000039	CUDA.Tesla_C870.domain_b.instructions	# instructions executed

Short Course Outline

- Lecture 1: Introduction and Fundamentals
- Lecture 2: Methodology
- Lecture 3: Tools Technology
- Lecture 4: Tools Landscape – Part 1
- Lecture 5: Tools Landscape – Part 2
- Lecture 6: TAU Performance System
- Lecture 7: TAU Applications
- Lecture 8: Advances in TAU
- Lecture 9: Future Directions

Tuesday

Wednesday

Thursday