Ménage à Trois:
Hybrid Profiling, Performance Visualization, and Kernel Measurement

Allen D. Malony, Chee Wai Lee, Wyatt Spear, Will Voorhees
Sameer Shende, Scott Biersdorff, Suzanne Millstein

Dept. Computer and Information Science
Performance Research Laboratory
University of Oregon
Hybrid Profiling – Motivation

- Different approaches for observing parallel performance

- **Sampling-based measurement**
  - Event-based / instruction-based sampling (EBS / IBS)
  - Examples: PerfSuite, HPCToolkit, ...

- **Probe-based measurement** (PBM)
  - Instrumentation of program code
  - Example: TAU, Scalasca, ...

- Combine the two to exploits advantages of probe-based instrumentation with advantages of sampling

- TAUebs
  - TAU for probe-based instrumentation and measurement
  - Event-based sampling measurement (with callstack unwinding)
**Integrated Probe + EBS Measurement Design (1)**

**Sampling Timeline**

**Sampling-Based Profile**
- 0x2110
- 0x3820
- 0x4894

**PC Histogram**

<table>
<thead>
<tr>
<th>PC addr range</th>
<th>1</th>
</tr>
</thead>
</table>

**Probe-Based Profile**

Active event stack = TAUkey

**Integration?**

**Instrumentation Timeline**
Hmm, seems like we have seen this before …

- Previously, TAUebs:
  - Captured a trace of EBS samples
  - Post-processed the trace to recover symbol information
  - Merged sample traces with generated profiles offline
  - Paper in ICPP 2010 and discussed briefly at CScADS
Integrated Probe + EBS Measurement Design (2)

Sampling Timeline

Sampling-Based Profile

PC Histogram

PC addr range

Probe-Based Profile

Active event stack = TAUkey

Integration

Sampling-Based Trace

TAUkey 0

TAUkey 1

TAUkey 2

Samples since last flush

Instrumentation Timeline

DOE CScADS
So what’s new?

- Previously, TAUebs:
  - Captured a trace of EBS samples
  - Post-processed the trace to recover symbol information
  - Merged sample traces with generated profiles offline
  - Paper in ICPP 2010 and discussed briefly at CScADS

- Now, TAUebs:
  - Captures EBS sample histograms at runtime (profiling)
  - Sample histograms are associated with TAU event context
  - TAU profile output now incorporates sampled histograms
  - It is still possible to generate EBS traces
TAUebs Hybrid Profiling

- Instance of new sample contextualized by TAUkey and integrated into TAU profile structures at runtime

♫ New Sample:
Function a(), address: 0x21098

loop
foo()
...
main

TAUkey:
“main-> … -> foo() -> loop”

PC Histogram:
0x21098 == 15
...
0x45362 == 23
Hybrid Profiling Implementation

- Uses existing timer-interrupt framework to trigger samples
- With each sample:
  - Query active TAU event context to determine TAUkey
  - Create/update PC address histogram for the active TAU event context represented by its key
- Addresses are resolved to meaningful symbol information via BFD at the end of the run
- TAU event context can be controlled by the event path depth

Caveats

- Current implementation does not unwind the callstack
  - flat sample profile for each TAU event context
- Works only with single-threaded processes presently
Examples: Simple Benchmark – Flat Profile

- Pure sampling – only main() is instrumented
Examples: Simple Benchmark – Hybrid Profile

- Mix of sampling and probed-based instrumentation

![Image showing a diagram with labeled sections: C: main, C: main → a, C: main → a → main_foo, C: main → a → main_foo → main_bar]
Examples: Simple Benchmark – Loops

- TAU events not restricted to routines (e.g., blocks, loops, …)

Samples show performance relative to line number
Examples: NAMD – Runtime System Contexts

- NAMD is implemented with Charm++ programming model
- Charm++ exposes its programming and runtime constructs
  - Callback system for TAU for probed-based measurement
- Hybrid profiling reveals action within Charm++ “idle” state

Uninstrumented runtime routines

Hopper, Cray XE6
144 processors
**Examples: FLASH4 – Event Paths + Samples**

Follow probed event path to end and pick up samples to highlight significant code points.

**Hopper, Cray XE6**

144 processors
Example: FLASH4 – Reverse Event/Call Path

- Aggregate events / samples performance data

TAU events within which this memcpy sample occurs
Issues – Signal Safety and Dropped Samples

- Signal safety
  - Strictly, signal handlers cannot attempt to acquire lock
  - TAUebs drops a sample if inside TAU operations for safety
  - EBS sample handler must avoid memory allocation
    - currently, use own C++ allocator and minimize calls to malloc

- Signal safety handling will drop samples
  - Keep track of # dropped samples in metadata

- Rules can be loosened to drop sample in a more intelligent way (only if unsafe)
TAUebs Future Work

- Add callstack unwinding when sampling
  - Incorporate robust module for this
  - Control of unwinding depth with event/routine matching
  - Selective unwinding determined by event context
  - Mix of flat and structured sample blocks per context
- Rational interpretation of code structures for samples relative to non-function TAU contexts (e.g., loops)
- Improvements in data management
- Handling symbol resolution for sampled addresses at epoch transitions due to dynamic library loading and unloading
- More accurate sample timing
  - Measuring time elapsed between a sample and last probed event
- Sampling with other metrics and state (e.g., GPU counters)
Performance Visualization – Motivation

- Large performance data presents interpretation challenges
- Visualization aids in data exploration and pattern analysis
  - 3D visualization can help in identifying relations between events/metrics
- Existing tools provide “canned” views
  - TAU provides a few
    - 2D: bargraph, histogram
    - 3D: full profile, correlation
- Developing new visualizations is a challenge
  - Strategy 1: Create new view for each problem
  - Strategy 2: Use external visualization environment
- Provide high-level support to use within existing framework
User defines visualization based on performance data model
- Specifies layout based on events, metrics, and metadata
- UI provides control of data binding and visualization
Using Process Topology Metadata

- Inspired by the CUBE topology display for BG/P
- Each point represents a thread of execution (MPI process)
  - Positioned according to the Cartesian (x,y,z,t) coordinates
- Color is determined by selected event/metric value
- Topology information can be recorded in TAU metadata
- ParaProf reads metadata to determine topology and create layout
- Sweep3D 16K run on BG/L
  - Color is exclusive time in the “sweep” function
**Topology Control UI**

- *Layout* tab allows customization of the position and visibility of data points
- Performance event/metric data used to define color and position is selected in the *Event* tab
- Additional rendering options, such as color scale and point size are available
- 4k-core S3D run on BG/P
Alternate Topologies

- Certain views may hide deeper inter-process behavior
- Spatially dependent performance issues may be revealed by manipulating topology
- Sweep3D profile with alternative Cartesian mapping exposes distribution of computational effort
- Topology has direct effect on communication
- Visualization mapped to hardware topologies can suggest better node/rank mapping
- `MPI_AllReduce()` values for Sweep3D highlights waiting distribution from rank 0 (lower left) to the most distant rank (upper right)
Viewing Internal Structure

- Dense topologies can hide internal structure
- Restrict visibility by color value to expose performance patterns
- ParaProf visualization UI now allows for range filtering
  - Mid-level values can be excluded
  - Remaining points are:
    - high outliers (hotspots)
    - low outliers (underutilized nodes)
Slicing to Reduce Dimensionality

- Restrict visibility to slices along the spatial axes
- Multiple axis controls allow selection of planes, lines, or an individual point
- ParaProf visualization UI provides filtering behavior
  - Averaging the color value for all points in the selected area
Visual Layout Specification

- Want to allow creation of explicit layouts
- Define a specification “language” that allows mathematical expressions to describe features of performance display
  - Equations define X, Y, Z coordinates and color per process
  - Event and metrics are seen as variables
    - $\text{eventX}.\text{val}$ : value for Xth specified event and metric
    - $\text{eventX.}\{\text{min, max, mean}\}$ : global aggregate values
    - $\text{atomicY}$ : Yth atomic event value
  - Intermediate variables can be used in the calculation
  - Defined global variables (e.g., max rank) are provided
- Specifications are loaded and processed by ParaProf
  - Use the MESP expression parser
Sphere Layout Specification

- Spatially mediated performance behavior may not be represented directly in topology metadata
  - Applications allocate resources with respect to a data-driven model
- The position of each point can be defined by custom equations in terms of event/metric, aggregate, atomic event and metadata
- Sweep3D profile mapped to a sphere

```
BEGIN_VIZ=Sphere
rootRanks=sqrt(maxRank)
theta=2*pi()/rootRanks*mod(rank,rootRanks)
phi=pi()/rootRanks*(ceil(rank/rootRanks))
x=cos(theta)*sin(phi)*100
y=sin(theta)*sin(phi)*100
z=cos(phi)*100
END_VIZ
```
ParaProf Events Panel

- Events / metrics get bound in ParaProf UI

- Example:
  - event0 is the FLOP count for function foo
  - event1 is the time value for function foo
  - To set the X coordinate for each process point to the FLOPS for event foo:
    \[ x = \text{event0.val} / \text{event1.val} \]
  - To set the Y coordinate for each process point to the global average FLOPS for event foo:
    \[ y = \text{event0.mean} / \text{event1.mean} \]
Adding Dimensionality

- Topologies can involve more than three dimensions (e.g., intranode)
- Mirror actual machine layout to capture communication structure and cores
- Custom layouts allow specification of multiple points from a single process/rank
- 4K-core S3D run on BG/P
- Default topology only covers X, Y, Z coordinates
- A custom topology divides each $n$th core into its own block

BEGIN_VIZ=4K_8x8x16Block
xdim=8
ydim=8
zdim=16

$x=\text{mod}(\text{rank},\text{xdim})+16*\text{floor}(\text{rank}/1024)$
$y=\text{mod}(\text{floor}(\text{rank}/\text{xdim}),\text{ydim})$
$z=\text{mod}(\text{floor}(\text{rank}/\text{xdim}/\text{ydim}),\text{zdim})$

END_VIZ
Non-Spatial Relationships

- Positioning of points needs not be with respect to physical or data topology.
- Correlation of metrics within the same events or events between processes can indicate relevant performance effects.
- Partitioning or clustering of different processes based on selected performance criteria.
- 3D scatterplot for 10240 core run of GCRM/ZGrd application.
- Correlates four selected events, one for each spatial axis plus color.

BEGIN_VIZ=ScatterTest
restrictDim=1
x=event0.val
y=event1.val
z=event2.val
END_VIZ
**Visualization Next Steps**

- Collect topology data from additional platforms (e.g. Cray)
- Expand UI for more general access to performance data model
- Allow independent manipulation of unconnected segments
- Improve presentation of data values, ranks, and metrics
- Better functionality for automatic higher-dimensional layouts
- Add representation of communication channels
TROIS
Kernel Measurement (KTAU) – Motivation

- Observe kernel performance and integrated with application performance measurements
- Earlier development of KTAU (Kernel TAU)
  - Profiling and tracing measurement of kernel events
  - Done via source instrumentation of Linux kernel
- Need more viable solution going forward
  - No patches or source modification
  - Easy to use and install
- Objective
  - Re-implement original KTAU features
  - Leverage work in the kernel instrumentation community
Approach

- Utilizes kernel infrastructure for tracing
  - *tracepoints* and *kprobes*
- Simple user application with loadable kernel module
  - Similar to Unix “time”
- Efficient memory mapping between user and kernel space
- Minimal instructions required to record performance data
- Support for both profiling and tracing
Major Components of KTAU

Ktau core

Kernel Space

User Space

IOCTL_COMMANDS
+cmd_size_profile
+cmd_read_profile
+cmd_merge_profile
+cmd_add_shcont
+cmd_del_shcont
+cmd_add_shcnt
+cmd_del_shcnt
+cmd_inject_noise

kernel

Ktau_source_inst

runKtau

Ktau

Ktau_prof
+rcu_hash_node: struct hlist_node
+struct hlist_node: struct list_head
+create_task_profile(): ktau_prof *
+free_task_profile(): void
+free_task_profile_sched()
+dump_hash_self(): int
+dump_hash_other(): int
+get_profile_size()
+get_profile_size_many()
+dump_profile_many()
+dump_profile_self()

Ktau_inst
+ktau_start_prof()
+ktau_stop_prof()
+ktau_start_trace()
+ktau_stop_trace()
+get_ktau_index()
+incr_ktau_index()

ktau_prof
+ktau_pidhash
+struct hlist_head *ktau_pid_hash
+struct list_head ktau_prof_list;
+ktau_pidhash_Init()
+ktauPidhash_Free()

ktau_pidhash

syscall
+syscall_enter()
+syscall_exit()
KTAU and Other Projects

- What about Oprofile, LTTNG and SystemTap?
- These provide similar data from the kernel
- Way in which data is used and displayed is different perhaps
- KTAU focus is on application developers
- Comparing to LTTNG
  - Both use tracepoints, kprobes, RCU locking, kernel timestamp
  - KTAU
    - only requires kernel headers to build, root to install module
    - new development with targeted instrumentation
    - works with TAU to produce profiles and traces
  - LTTNG
    - mature with lots of instrumentation points
    - requires kernel patches
    - very basic user space instrumentation
    - no profile support
**KTAU Status and Future**

- Just finished initial prototype
- Undergoing more robust testing and evaluation
- Re-engineering of profile/trace merging tools

- Investigate interactions with LTTNG
  - Some movement towards including LTTNG in kernel
- Develop more efficient mechanism for accessing KTAU data
  - Use shared memory regions between kernel and user