Sampling-based Performance Analysis with HPCToolkit

Measurement and Analysis of Unmodified, Optimized Applications

John Mellor-Crummey

Department of Computer Science
Rice University

15 November 2021

Outline

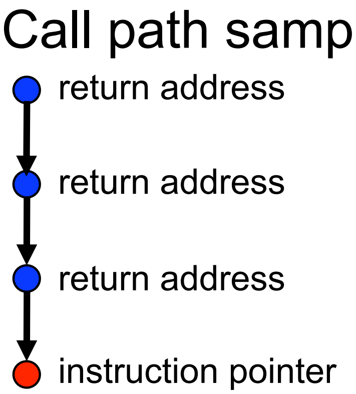
- **Sampling-based call path profiling**
- **Using HPCToolkit on a single node**
- **Demo data collection and profile analysis**
- **Demo trace analysis**
- **Pointers to additional information about measurement**

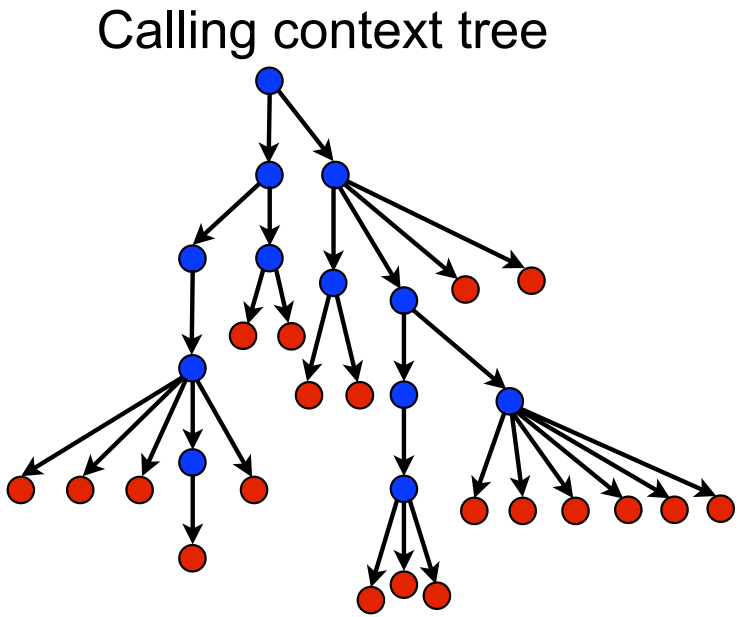
Sampling-based Performance Measurement

- **Periodically interrupt each thread in an application**
  - interrupts are triggered by a “sample source” as a metric reaches some pre-determined threshold
  - example sample sources
    - timer
      - a thousandth of a second has passed
    - hardware counters
      - five million instructions have completed
      - a million cache misses have occurred
- **Why sampling?**
  - controllable overhead
  - avoids blind spots and minimizes systematic error

Attribute Metrics to Call Paths

- **When a thread's timer or HW counter reaches some predetermined threshold T**
  - **interrupt a thread**
  - **unwind its call stack**





  - **charge T to the thread's current calling context**

Overhead proportional to sampling frequency, not call frequency

HPCToolkit Quickstart

```
% hpcrun myapp
```

 # profile CPUTIME of application and deposit results in hpctoolkit-myapp-measurements

```
% hpcstruct hpctoolkit-myapp-measurements
```

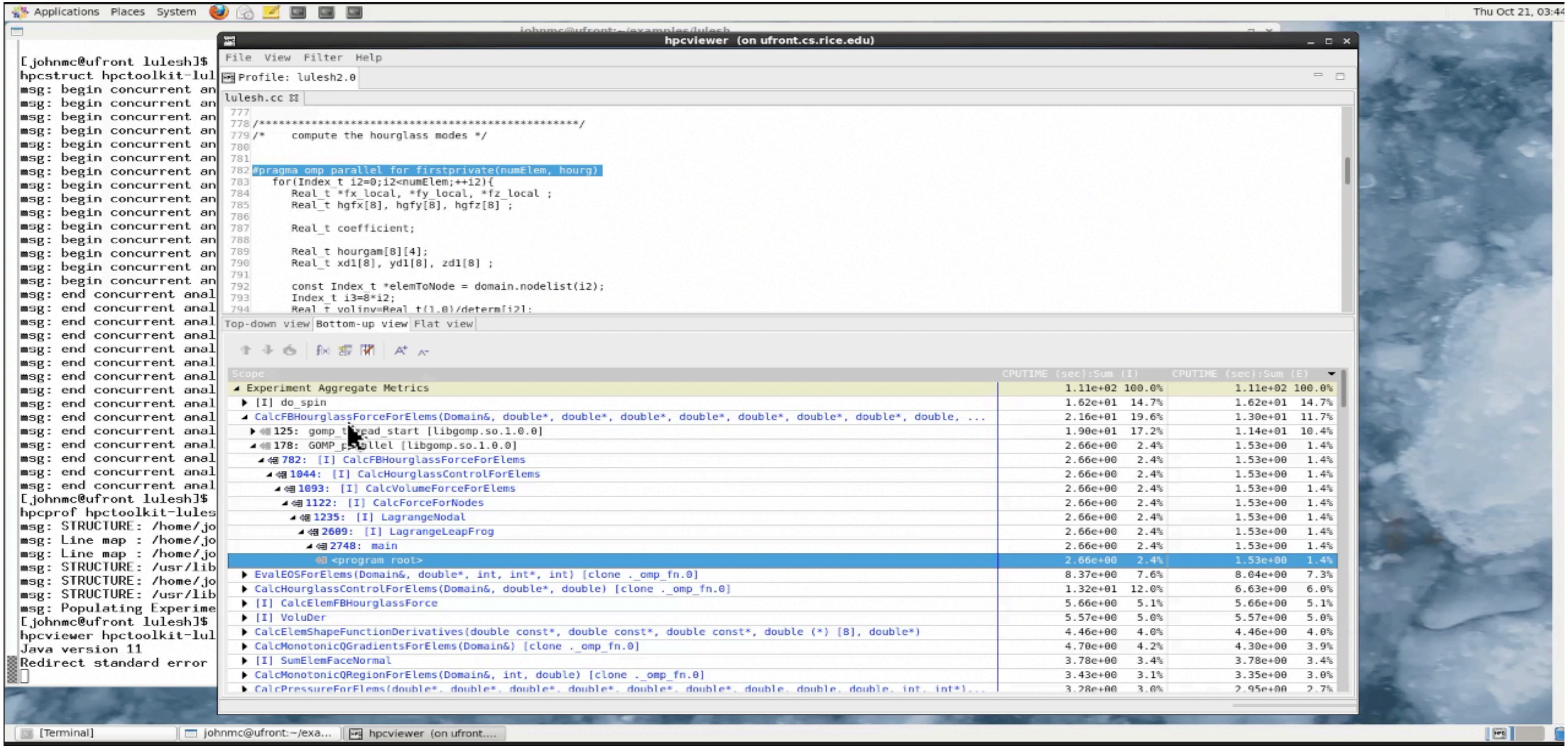
 # analyze application binary and all dynamically loaded libraries involved in the execution

```
% hpcprof hpctoolkit-myapp-measurements
```

 # analyze measurement data and correlate it to source using program structure from hpcstruct

```
% hpcviewer hpctoolkit-myapp-database
```

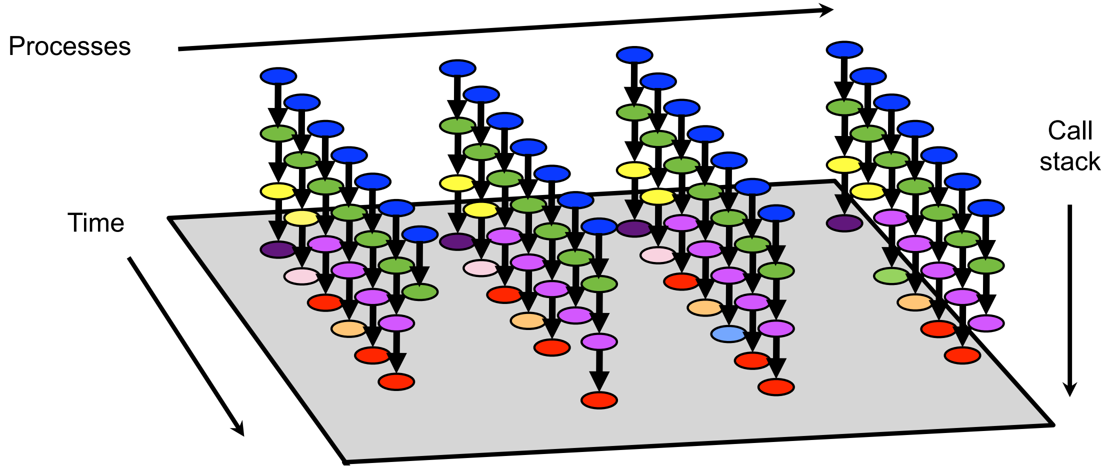
Video: Using HPCToolkit to Measure an OpenMP Program



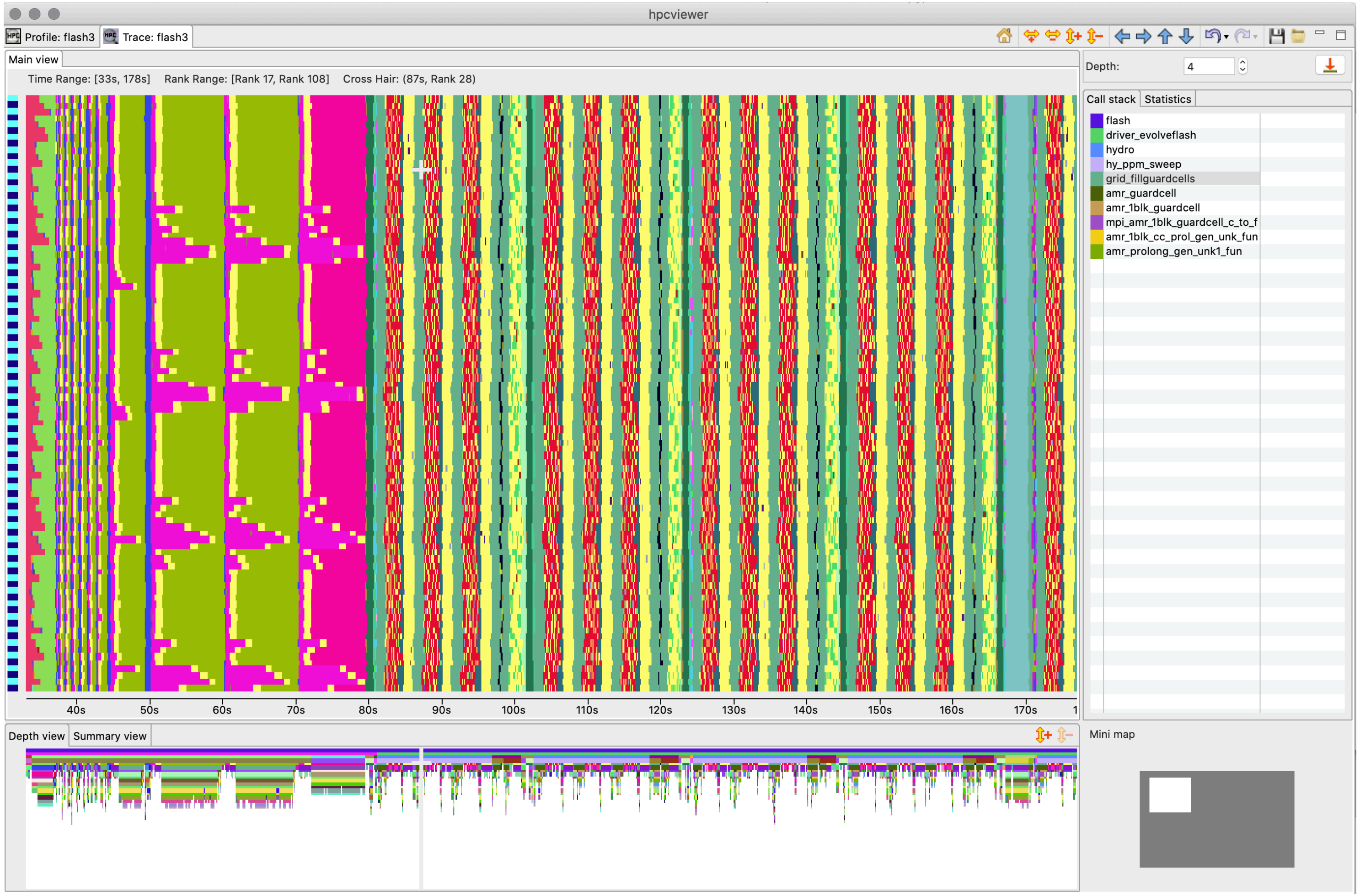
EXASCALE COMPUTING PROJECT

Understanding Temporal Behavior

- **Profiling compresses out the temporal dimension**
  - Temporal patterns, e.g. serial sections and dynamic load imbalance are invisible in profiles
- **What can we do? Trace call path samples**
  - N times per second, take a call path sample of each thread
  - Organize the samples for each thread along a time line
  - View how the execution evolves left to right
  - What do we view? assign each procedure a color; view a depth slice of an execution



Video: Using HPCToolkit to Analyze the Trace of an MPI Program



EXASCALE COMPUTING PROJECT

See Slide Deck for Additional Details about Measurement

- Measuring applications when using a job launcher
- Specifying sample sources
  - timers
  - hardware counters
- Controlling measurement frequency
  - automatic
  - frequency-based sampling
  - period-based sampling

Measuring Performance with hpcrun

- Profile a dynamic binary (sequential or multithreaded)
  - hpcrun *[measurement options]* myapp ….

- Use hpcrun with example job launcher commands
  - jsrun -n 32 -g 1 -a 1 hpcrun *[measurement options]* myapp
  - srun -n 1 -G 1 hpcrun *[measurement options]* myapp
  - aprun -n 16 -N 8 -d 8 hpcrun *[measurement options]* myapp

- Specifying CPU events to measure
  - hpcrun *-e <event1>[@<howoften1>] -e <event2>[@<howoften2>]* myapp ….

Note: To profile statically-linked applications, you must link your application with HPCToolkit's measurement subsystem using hpclink. See the HPCToolkit manual for details.

hpcrun - Tracing

- Specify tracing simply by adding “-t” as an argument to hpcrun
- Requirements
  - must be measuring execution with a time-based metric
    - Linux timer
    - “cycles” measured with `perf_event`

Sample Sources: Linux Timers

- **CPUTIME** (DEFAULT if no sample source is specified)
  - -e CPUTIME@<period>: interrupt each thread every <period> microseconds it executes
  - does not include time blocked in the kernel
    - disadvantage: misses time a thread is blocked
    - advantage: a blocked thread is never woken to take a sample

Best for analysis of profile data

- **REALTIME**
  - -e REALTIME@<period>: interrupt each thread every <period> microseconds
  - includes time blocked in the kernel
    - advantage: shows where a thread spends its time, even when blocked
    - disadvantages
      - activates a blocked thread to take a sample
      - a blocked thread appears active even when blocked

May produce more intuitive traces

Note: Only use one Linux timer to measure an execution

Sample Sources: Hardware Counters

- Each core in a modern processor has a performance monitoring unit with counters for HW events
  - each thread has a small number of HW counters
- Linux kernel: perf_event subsystem for performance monitoring
  - access and manipulate
    - hardware counters: cycles, instructions, …
    - software counters: context switches, page faults, …
  - available in Linux kernels 2.6.31+

A useful explanation about events available through perf
https://sites.google.com/site/lbathen/research/perf

EXASCALE COMPUTING PROJECT

Sample Sources: perf_event Hardware Event Counters

- PERF_COUNT_HW_CPU_CYCLES
- PERF_COUNT_HW_INSTRUCTIONS
- PERF_COUNT_HW_CACHE_REFERENCES
- PERF_COUNT_HW_CACHE_MISSES
- PERF_COUNT_HW_BRANCH_INSTRUCTIONS
- PERF_COUNT_HW_BRANCH_MISSES
- PERF_COUNT_HW_BUS_CYCLES
- PERF_COUNT_HW_STALLED_CYCLES_FRONTEND
- PERF_COUNT_HW_STALLED_CYCLES_BACKEND
- PERF_COUNT_HW_REF_CPU_CYCLES

Sample Sources: `perf_event` Hardware Cache Events

- **Hardware cache**
  - PERF_COUNT_HW_CACHE_L1D
  - PERF_COUNT_HW_CACHE_L1I
  - PERF_COUNT_HW_CACHE_LL
  - PERF_COUNT_HW_CACHE_DTLB
  - PERF_COUNT_HW_CACHE_ITLB
  - PERF_COUNT_HW_CACHE_BPU
- **Operations**
  - PERF_COUNT_HW_CACHE_OP_READ
  - PERF_COUNT_HW_CACHE_OP_WRITE
  - PERF_COUNT_HW_CACHE_OP_PREFETCH
- **Results**
  - PERF_COUNT_HW_CACHE_RESULT_ACCESS
  - PERF_COUNT_HW_CACHE_RESULT_MISS

Sample Sources: `perf_event` Software Events

- PERF_COUNT_SW_CPU_CLOCK
- PERF_COUNT_SW_TASK_CLOCK
- PERF_COUNT_SW_PAGE_FAULTS
- PERF_COUNT_SW_CONTEXT_SWITCHES
- PERF_COUNT_SW_CPU_MIGRATIONS
- PERF_COUNT_SW_PAGE_FAULTS_MIN
- PERF_COUNT_SW_PAGE_FAULTS_MAJ
- PERF_COUNT_SW_ALIGNMENT_FAULTS
- PERF_COUNT_SW_EMULATION_FAULTS

useful when monitoring data-intensive codes

Sample Sources: Measuring Other HW Events with `perf_event`

- See the full list of available events with
  - hpcrun -L
- Perf events are grouped by categories indicated by a prefix
  - `ix86arch::<event>` // Intel architecture
  - `perf::<event>` // perf_event builtin
  - `bdw_ep::<event>` // Broadwell EP specific
  - …
- For convenience
  - you may omit the category prefix, e.g. "`perf::`"
  - you may specify `perf_event` counter names using lower case

EXASCALE COMPUTING PROJECT

Controlling perf_event Sampling Frequency

- **Automatic** Recommended
  - HPCToolkit samples `perf_event` counters min(300x/second, maximum Linux allows)
    - may be higher than necessary for long executions
      - reducing the frequency will reduce measurement overhead
- **Specify frequency**
  - use the @f<freq> suffix for an event to specify frequency
    - `hpcrun -e cycles@f100 -e instructions@f200 …`
  - specify a different default frequency using the -c option
    - example: sample both cycles and instructions 200x per second
      - `hpcrun -c f200 -e cycles -e instructions …`
- **Specify period**
  - use the @<period> suffix for an event to specify a period
    - `hpcrun -e cycles@1000000 -e instructions@5000000 …`

ECP EXASCALE COMPUTING PROJECT

Sample Sources: Multiplexing Events

- A single execution can measure more HW events than the number of counters available per thread
- If you specify more events than counters available
  - `perf_event` will automatically multiplex them
- How multiplexing works with Linux `perf_event` subsystem
  - at any time, the number of events being collected will not exceed the number of HW counters per thread
  - kernel will partition events into sets that can be monitored simultaneously using counter resources
    - monitors one set of events for a while then switches to another
  - kernel uses schedules event sets round-robin
  - multiplexing is convenient but there is some loss of accuracy
    - advice: multiplexing is fine for casual execution analysis

EXASCALE COMPUTING PROJECT