



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

🐉 Applications Places System 🎈			15 million and a strength	Thu Oct 21
	iohume@uiront=devamilar/luideh			
	端 npcviewer (on urront.cs.rice.eau)		×	and the second se
[johnmc@ufront lulesh]\$	File View Filter Help			100 Section and and
hpcstruct hpctoolkit-lul	Profile: lulesh2.0			and the second second
msg: begin concurrent an	Julash co W			AND CONTRACTOR OF SHIELD
sg: begin concurrent an	Contract as a second seco			
sg: begin concurrent an	7778 /**********************************			
sg: begin concurrent an	779/* compute the hourglass modes */			Concernence of the second second
sg: begin concurrent an	780			And the second
sg: begin concurrent an	781 782 Morgama amp parallel for firstprivate(numElem_hourn)		1	and the second second second
sg: begin concurrent an	783 for(Index t i2=0;i2 <numelem;++12){< th=""><th></th><th></th><th></th></numelem;++12){<>			
sg: begin concurrent an	784 Real_t *fx_local, *fy_local, *fz_local ;			the first state of the second
sg: begin concurrent an	785 Real_t hgfx[8], hgfy[8], hgfz[8];			
sg: begin concurrent an	780 782 Real t coefficient:			A DESCRIPTION OF THE OWNER OF THE
sg: begin concurrent an				and the second s
sg: begin concurrent an	789 Real_t hourgam[8][4];			And the second second
sg: begin concurrent an	790 Real_t xd1[8], yd1[8];			Constant of the local division of the local
sg: begin concurrent an	791 702 const Index t *elemToNode = domain nodelist(i2):			and the second second
sg: end concurrent anal	793 Index t i3=8*i2:			
sg: end concurrent anal	794 Real t volinv=Real t(1.0)/determii21:			and the second second
sg: end concurrent anal	Top-down view Bottom-up view Flat view			Martin Carlos
sg: end concurrent anal				and the second second
sg: end concurrent anal	1 + 6 10 25 17 At At			and the second
sg: end concurrent anal	Score -	CONTINE Leach-Sum (T) CONTIN	E (coc) Sum (E)	
sg: end concurrent anal	A Experiment Angregate Metrics	1 110482 100 0%	1 110+92 188 85	and the second se
sg: end concurrent anal	Lapriment aggregate metrics	1.620+01 14.7%	1 620+01 14 75	and the second se
sg: end concurrent anal	 CalFEBHourglassEnccEDFElems(Domain& double* double* double* double* double* double* double* double* 	2 166481 19 6%	1 390+91 11 75	the set of
sg: end concurrent anal	 Carteniori gran tive a start librorn so 1.0.01 	1.986+81 17.2%	1.14e+01 10.4%	and the second second
sg: end concurrent anal	A 4178 - GOMP rotallel (liborn so 1 6 a)	2 660+80 2 4%	1.530+00 1.4%	
sg: end concurrent anal	47782: [1] [3] CEBHourd Jass Force For Flows	2.666+86 2.4%	1 530+00 1.45	
sg: end concurrent anal	A minimum of the second	2.660+80 2.4%	1.536+00 1.4%	
sg: end concurrent anal	491093: [1] CalcVolumeEorceForElems	2.660+80 2.4%	1.53e+00 1.4%	ALC: NOT
johnmc@ufront lulesh]\$	4 (#1122) [I] CalcEnceEnrNodes	2.66e+80 2.4%	1.530+00 1.4%	and the second se
pcprof hpctoolkit-lules	4 df 1235: [1] LagrangeNoda]	2.66e+80 2.4%	1.53e+00 1.4%	and the second states
sg: STRUCTURE: /home/jo	49 2689: [1] LagrangeleapErog	2.66e+80 2.4%	1.536+00 1.4%	
sg: Line map : /home/jo	4 41 2748: main	2.66e+00 2.4%	1.53e+00 1.4%	and the second s
sg: Line map : /home/jo	∰ <pre>sprearam root></pre>	2.66c+00 2.4%	1.53e+00 1.4%	Statements Street and
sg: SIRUCIURE: /usr/lib	EvalEOSForElems(Domain&, double*, int, int*, int) [clone . omp fn.0]	8.37e+00 7.6%	8.04e+00 7.3%	the second second
sg: STRUCTURE: /home/jo	CalcHourglassControlForElems(Domain&, double*, double) [clone . omp fn.0]	1.32e+01 12.0%	6.63e+00 6.0%	and the second se
sg: SIKULIUKE: /usr/lib	[I] CalcElemFBHourglassForce	5.66e+00 5.1%	5.66e+00 5.1%	and the second se
ichnwc@ufront lulach]#	▶ [I] VoluDer	5.57e+00 5.0%	5.57e+00 5.0%	THE OWNER OF THE OWNER OWNER OF THE OWNER OWNE OWNER OWNE OWNER OWNE OWNER OWNER OWNE OWNER OWNE
cviewer boctoolkit-lul	CalcElemShapeFunctionDerivatives(double const*, double const*, double const*, double (*) [8], double*)	4.46c+00 4.0%	4.46e+00 4.0%	a state of the sta
ava version 11	CalcMonotonicQGradientsForElems(Domain6) [clone . omp fn.0]	4.70e+00 4.2%	4.30e+00 3.9%	and the second se
edirect standard error	[I] SumElemFaceNormal	3.78e+00 3.4%	3.78e+00 3.4%	and the second se
	CalcHonotonicQRegionForElems(Domain&, int, double) [cloneomp_fn.0]	3.43e+00 3.1%	3.35e+00 3.0%	and the second se
	CalcPressureForFlems(double*, double*, double*, double*, double*, double*, double, double, int, int*)	3.28e+80 3.6%	2.95e+08 2.7%	
		A REAL PROPERTY AND A REAL		a company of the
[Terminal] 📄 jo	hnmc@ufront:~/exa] 🔄 hpcviewer (on ufront			



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





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
- 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



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

Sample Sources: perf_event Hardware Event Counters

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





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

Sample Sources: perf_event Software Events

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



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 ...

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

