Computationally Efficient Multiplexing of Events on Hardware Counters

Robert V. Lim
University of California, Irvine
roblim1@ics.uci.edu

David Carrillo-Cisneros
University of California, Irvine
dcarril@ics.uci.edu

Wail Y. Alkowaileet
University of California, Irvine
walkowai@ics.uci.edu

Isaac D. Scherson
University of California, Irvine
isaac@ics.uci.edu

Abstract

This paper proposes a novel approach for scheduling n performance monitoring events onto m hardware performance counters, where $n > m$. Whereas existing scheduling approaches overlook monitored task information, the proposed algorithm utilizes the monitored task's behavior and schedules the combination of the most costly events. The proposed algorithm was implemented in Linux Perf Event subsystem in kernel space (build 3.11.3), which provides finer granularity and less system perturbation in event monitoring when compared to existing user space approaches. Benchmark experiments in PARSEC and SPLASH.2x suites compared the existing round-robin scheme with the proposed rate-of-change approach. Results demonstrate that the rate-of-change approach reduces the mean-squared error on average by 22%, confirming that the proposed methodology not only improves the accuracy of performance measurements read, but also makes scheduling multiple event measurements feasible with a limited number of hardware performance counters.

1 Introduction

Modern performance tools (PAPI, Perf Event, Intel vTune) incorporate hardware performance counters in systems monitoring by sampling low-level hardware events, where each performance monitoring counter (PMC) is programmed to count the number of occurrences of a particular event, and its counts are periodically read from these registers. The monitored results collectively can provide insights into how the task behaves on a particular architecture. Projecting performance metrics such as instructions-per-cycle (IPC), branch mispredictions, and cache utilization rates not only helps analysts identify hotspots, but can lead to code optimization opportunities and performance tuning enhancements. Hardware manufacturers provide hundreds of performance events that can be programmed onto the PMCs for monitoring. For instance, Intel provides close to 200 events for the current i7 architecture [6], while AMD provides close to 100 events [12]. Other architectures that provide event monitoring capabilities include NVIDIA's `nvprof` and Qualcomm's Adereno profilers [11, 14]. While manufacturers have provided an exhaustive list of event types to monitor, the issue is that microprocessors usually provide two to six performance counters for a given architecture, which restricts the number of events that can be monitored simultaneously.

Calculating performance metrics involves n low-level hardware events, and modern microprocessors provide m physical counters (two to six), making scheduling multiple performance events impractical when $n > m$. A single counter can monitor only one event at a time, which means that two or more events assigned to the same register cannot be counted simultaneously (conflicting events) [9].

Monitoring more events than available counters can be achieved with time interpolation techniques, such as multiplexing and trace alignment. Multiplexing consists of scheduling events for a fraction of the execution and extrapolating the full behavior of each metric from its samples. Trace alignment, on the other hand, involves collecting separate traces for each event run and combining the independent runs into a single trace.

Current approximation techniques for reconstructing event traces yield estimation errors, which provides inaccurate measurements for performance analysts [10]. The estimation error increases with multiplexing be-

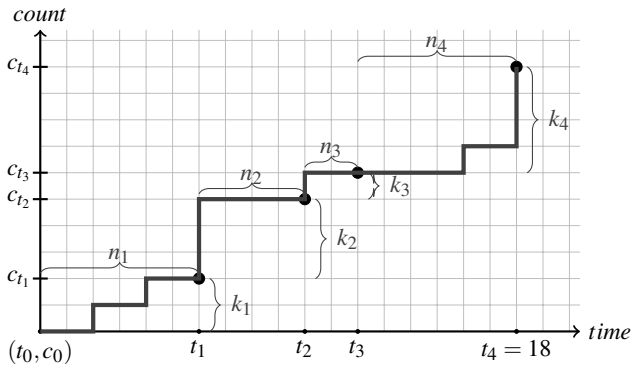

Figure 1: An example of a sequence of sampled values. The sampling times $t_1 \ldots t_4$ and counts $c_{t_0} = 0, \ldots c_{t_4}$ are known.

cause each event timeshares the PMC with the other events, which results in loss of information when the event is not being monitored at a sampled interval. Trace alignment may not be feasible in certain situations, where taking multiple runs of the same application for performance monitoring might take days or weeks to complete. In addition, the authors have shown that between-runs variability affects the correlation between the sampled counts for monitored events, due to hardware interrupts, cache contention, and system calls [16]. Current implementations schedule monitoring events in a round-robin fashion, ignoring any information about the program task. Opportunities for better event scheduling exist if information about the behavior of the task is taken into account, an area we address in this paper.

This paper is organized as follows. Section 2 discusses previous work. Our multiplexing methodology is presented in Section 3. Section 4 evaluates the experimental results. Lastly, Section 5 concludes with future work.

2 Previous Work

To the best of our knowledge, there has not been any prior work similar to our methodology for multiplexing n performance events onto m hardware counters. The next subsections discuss several performance monitoring tools and its respective multiplexing strategy.

2.1 Performance Monitoring Tools

Performance monitoring tools provide access to hardware performance counters either through user space or kernel space.

Performance Application Programming Interface (PAPI) is an architecture independent framework that provides access to generalized high-level hardware events for modern processors, and low-level native events for a specific processor [2]. PAPI incorporates MPX and a high resolution interval timer to perform counter multiplexing [9]. The TAU Performance System, which integrates PAPI, is a probed-based instrumentation framework that profiles applications, libraries, and system codes, where execution of probes become part of the normal control flow of the program [15]. PAPI's ease of use, and feature-rich capabilities make the framework a top choice in systems running UNIX/Linux, ranging from traditional microprocessors to high-performance heterogeneous architectures.

Perfmon2, a generic kernel-level performance monitoring interface, provides access to the hardware performance monitoring unit (PMU) and supports a variety of architectures, including Cray X2, Intel, and IBM PowerPC [4]. Working at the kernel level provides fine granularity and less system perturbation when accessing hardware performance counters, compared to user space access [17]. Scheduling multiple events in Perfmon2 is handled via round-robin, where the order of event declaration determines its initial position in the queue. Linux Perf Event subsystem is a kernel level monitoring platform that also provides multi-architectural support (x86, PowerPC, ARM, etc.) [13]. Perf has been mainlined in the Linux kernel, making Perf monitoring tool available in all Linux distributions. Our proposed methodology was implemented in Perf Event.

2.2 Perf Event in Linux

Perf Event samples monitoring events asynchronously, where users set a *period* (at every i^{th} interval) or a *frequency* (the number of occurrence of events). Users declare an event to monitor by creating a file descriptor, which provides access to the performance monitoring unit (PMU). The PMU state is loaded onto the counter register with a `perf_install_in_context` call. Similar to Perfmon2, the current criteria for multiplexing events is round-robin.

A monitoring Perf Event can be affected under the following three scenarios: `hrtimer`, scheduler tick, and interrupt context.

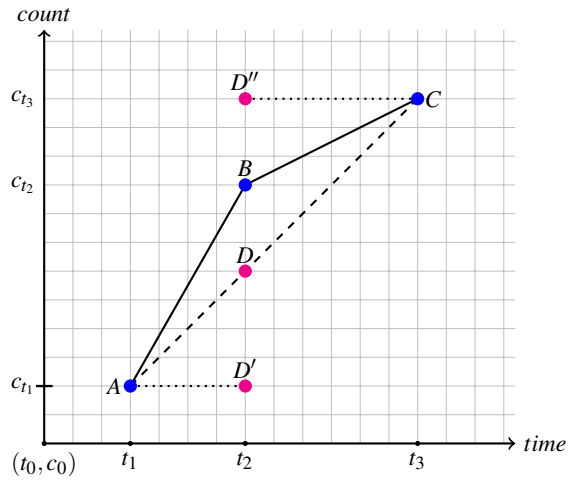

Figure 2: Triangle representing rate-of-change calculation for the recent three observations A, B, and C.

2.2.1 `hrtimer`

`hrtimer` [5] is a high resolution timer that gets triggered when the PMU is overcommitted [7]. `hrtimer` invokes `rotate_ctx`, which performs the actual multiplexing of events on the hardware performance counters and is where our rate-of-change algorithm is implemented.

2.2.2 Scheduler tick

Performance monitoring events and its count values are removed and reprogrammed on the PMU registers during each operating system scheduler tick, usually set at *HZ* times per second.

2.2.3 Interrupt context

A non-maskable interrupt (NMI) triggers a PMU interrupt handler during a hardware overflow, usually when a period declared by the user has been reached. `perf_rotate_context` completes the interrupt context by multiplexing the events on the PMC.

Our methodology uses `hrtimer` to perform time-division multiplexing. That is, at each `hrtimer` triggered, Perf Event timeshares the events with the performance counters. During the intervals that the events are not monitored, the events are linearly interpolated to estimate the counts [9].

2.3 Linear interpolation

To define linear interpolation for asynchronous event sampling, we will first define a sample, and then use a pair of samples to construct a linear interpolation.

A sample $s_i = (t_i,\ c_{t_i})$ is the i-th sample of a PMC counting the occurrences of an arbitrary event. The sample s_i occurs at time t_i and has a value c_{t_i}. We define:

$$k_i = c_{t_i} - c_{t_{i-1}} \tag{1}$$

$$n_i = t_i - t_{i-1} \tag{2}$$

as the increments between samples s_{i-1} and s_i for an event's count and time, respectively.

The slope of the linear interpolation between the two samples is defined as follows:

$$m_i = \frac{k_i}{n_i} \tag{3}$$

Since all performance counters store non-negative integers, then $0 \leq k_i, 0 \leq n_i, 0 \leq m_i$, for all i. An event sample represents a point in an integer lattice. Figure 1 displays sampled values and the variables defined above.

3 Multiplexing Methodology

Time interpolation techniques for performance monitoring events have shown large accuracy errors when reconstructing event traces [10]. Although increasing the number of observations correlates with more accurate event traces, taking too many samples may adversely affect the quality of the monitored behavior, since each sample involves perturbing the system. Linear interpolation techniques has demonstrated its effectiveness in reconstructing unobserved event traces [8, 9].

Our rate-of-change algorithm increases the amount of monitoring time for events that do not behave linearly. Our intuition tells us that higher errors will occur for non-linear behaved events when reconstructing event traces with linear interpolation techniques. The current round-robin scheme does not detect varying behavior since it schedules each event indiscriminately, missing the opportunity to reduce scheduling time for events where linear interpolation may have been sufficient for reconstruction.

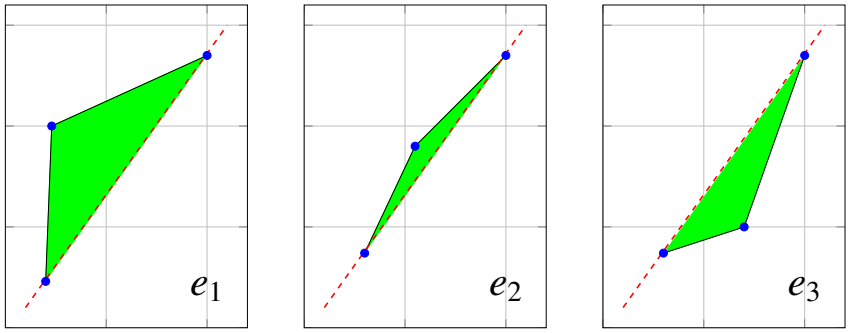

Figure 3: Triangle cost function for events e_1, e_2, and e_3

3.1 Rate-of-Change Definition

To motivate our proposed scheduling algorithm, we define rate-of-change as follows. Let $A_{x,y}$, $B_{x,y}$, and $C_{x,y}$ be the last three observations for a given event in a current program run, where x, y represent the observed time and count, respectively (Sec. 2.3). A triangle can be constructed in an integer lattice using the three observations, where the triangle's area represents the loss of information if $B_{x,y}$ were skipped (Fig. 2). Based on the principle of locality in software applications [3], we hypothesize that the loss of information due to an unknown $B_{x,y}$ is similar to the potential loss of information due to skipped future observations. In Figure 2, the dashed line represents an extracted trace from a linear interpolator, provided that $A_{x,y}$ and $C_{x,y}$ are known, and serves as our scheduling criteria.

3.2 Triangle Cost Function

The triangle cost function is calculated as follows. For any three observations $A_{x,y}$, $B_{x,y}$, and $C_{x,y}$, we have

$$D_x = B_x \qquad D_y = \delta_y + A_y$$
$$D'_x = B_x \qquad D'_y = A_y \qquad \delta_y = \frac{C_y - A_y}{C_x - A_x} \cdot (B_x - A_x)$$
$$D''_x = B_x \qquad D''_y = C_y$$

Definition 1 *The rate-of-change in observations $A_{x,y}$, $B_{x,y}$, and $C_{x,y}$ is given by the sum of the areas $\triangle_{ABD}$ and $\triangle_{BCD}$.*

$\triangle_{ABD}$ is calculated as follows:

$$\triangle_{ABD} = \frac{B_x - A_x}{2} \cdot (B_y - A_y - \delta_y) \tag{4}$$

The scheduling cost, C_{ABD}, is determined as follows:

$$C_{ABD} = \left| \frac{B_y - A_y - \delta_y}{2} \right| \cdot \delta_t \tag{5}$$

Calculations for $\triangle_{BCD}$ and C_{BCD} are similar to (4) and (5), respectively:

$$\triangle_{BCD} = \frac{C_x - B_x}{2} \cdot (B_y - A_y - \delta_y) \tag{6}$$
$$C_{BCD} = \left| \frac{B_y - A_y - \delta_y}{2} \right| \cdot \delta_t \tag{7}$$

3.3 Rate-of-Change as Scheduling Criteria

The rate-of-change algorithm calculates a cost function based on the recent event observations to determine whether the monitoring event should be scheduled next. A smaller triangle area implies less variability, since the area will be equivalent to a linear slope rate, and is easier to extrapolate. Conversely, a greater triangle area reflects sudden changes in the observations, or higher variability, which means that those events should be prioritized over others.

Figure 3 illustrates how the rate-of-change algorithm calculates the scheduling cost function for each event e_1, e_2, and e_3 with respect to the linear interpolator (red dashed line). The triangle area shaded in green represents the cost of scheduling a particular e_i. Event e_2 exhibits nearest linear behavior, which will be placed in the rear of the scheduling queue. Note that the constructed triangle for e_3 reflects with the linear interpolator, which is addressed with the absolute value sign in Equation 5. The objective of our rate-of-change algorithm is to "punish" non-linearly behaved events by scheduling those events ahead of the queue. After running our algorithm, the scheduling queue will be arranged as follows: $\{e_1, e_3, e_2\}$.

Hardware		Cache			
Event	Period	Event	Period	Event	Period
instructions	4,000,000	L1-dCache-hits	1,000,000	dTLB-Cache-hits	1,100,000
cache-references	9,500	L1-dCache-misses	7,000	dTLB-Cache-misses	750
cache-misses	150	L1-dCache-prefetch	7,000	iTLB-Cache-hits	4,200,000
branch-instructions	850,000	L1-iCache-hits	1,800,000	iTLB-Cache-misses	500
branch-misses	10,000	L1-iCache-misses	50,000	bpu-Cache-hits	900,000
stalled-cycles (frontend)	15,000	LL-Cache-hits	3,100	bpu-Cache-misses	600,000
stalled-cycles (backend)	1,500,000	LL-Cache-misses	75	node-Cache-hits	70
ref-cpu-cycles	2,000,000	LL-Cache-prefetch	500	node-Cache-misses	70
				node-Cache-prefetch	50

Table 1: Generalized Perf Events and its period settings.

Event starvation prevented

Our rate-of-change scheduling criteria prevents event starvation because the cost of scheduling increases as time since the last scheduled event (δ_t) increases; hence, preventing any events from not being scheduled (Eq. 5).

Computationally efficient

Our rate-of-change algorithm is computationally efficient because for every `hrtimer` triggered, only two integer multiplications and two integer divisions are taking place. Below is a snippet of `C` code for our computation:

```
delta_y = ((Cy-Ay)/(Cx-Ax)) * (Bx-Ax);
cost = ((By-Ay-delta_y) / 2) * d_time;
```

In cases where $C_x - A_x = 0$, δ_y is set to 0 which implies $C_x = A_x$, or that no changes have occurred since observation A_x.

4 Experiments and Results

4.1 Experiments

In order to verify our proposed rate-of-change methodology, we ran a subset of PARSEC benchmarks [1] and SPLASH.2X benchmarks [18], listed in Table 2, while sampling the events from the PMC periodically. SPLASH.2X is the SPLASH-2 benchmark suite with bigger inputs drawn from actual production workloads.

Our goal was to make use of `hrtimer` in our multiplexing methodology, which was released in kernel version `3.11.rc3` [7].

The generalized Perf Events for this experiment, listed in Table 1, were periodically sampled for each benchmark run. Each of the periods was determined by trial-and-error using the following formula.

$$samples/msec \quad = \quad \frac{\mu_{nr.samples}}{\mu_{t.elapsed}} \tag{8}$$

Ideal samples per milliseconds is 1, where the number of samples taken for an event is proportional to the time spent monitoring the event.

To account for between-runs variability, each of the events listed in Table 1 executed ten times for each benchmark package with `simlarge` as input, which ran on average 15 seconds each. Each execution consisted of programming one single event in the PMC to disable multiplexing, which serves as the baseline comparison. In addition, the events were multiplexed with the performance counters in the same run under the current round-robin scheme and with our rate-of-change methodology. Each of the ten runs was sorted in ascending order, based on counts for each scheduling strategy, and the fourth lowest value was selected as our experimental result for the single event baseline comparison, and for the two multiplexing strategies (round-robin, rate-of-change).

Our experiments ran on an Intel i7 Nehalem processor with four hardware performance counters. Table 3 lists the machine configuration for this experiment. Note that CPU frequency scaling, which facilitates the CPU in power consumption management, was disabled to make

Platform	*Package*	*Application Domain*	*Description*
PARSEC	blackscholes	Financial Analysis	Calculates portfolio price using Black-Scholes PDE.
	bodytrack	Computer Vision	Tracks a 3D pose of a markerless human body.
	canneal	Engineering	Minimizes routing costs for synthetic chip design.
	vips	Media Processing	Applies a series of transforms to images.
SPLASH.2x	cholesky	HPC	Factors sparse matrix into product of lower triangular matrix.
	fmm	HPC	Simulates interaction in 2D using Fast Multipole method
	ocean_cp	HPC	Large scale movements based on eddy and boundary currents.
	water_spatial	HPC	Evaluates forces and potentials in a system of molecules.

Table 2: PARSEC and SPLASH.2x Benchmarks.

Type	Events
Architecture	Intel Core i7, M640, 2.80 GHz (Nehalem)
Performance counters	IA32_PMC0, IA32_PMC1, IA32_PMC2, IA32_PMC3
Operating system	Linux kernel 3.11.3-rc3
CPU frequency scaling	Disabled
CPU speed	2.8 GHz

Table 3: Machine configuration.

Benchmark	Improvement
blackscholes	7.46
bodytrack	5.81
canneal	3.02
vips	1.48
cholesky	4.45
fmm	1.02
ocean_cp	4.67
water_spatial	2.69

Table 4: Improvement (%) per benchmark, averaged over event types.

all CPUs run consistently at the same speed (four CPUs running at 2.8 GHz, in our case). The rate-of-change algorithm hooked into `rotate_ctx` (Sec. 2.2.2) and the final counts were exported as a `.csv` file from Perf Event subsystem.

4.1.1 MSE on final counts

We compared our multiplexing technique with the existing round-robin approach by applying a statistical estimator on the mean μ calculated from the single event runs. We used mean-squared error (MSE) for comparing each mean μ_{roc} and μ_{rr} from the multiplexing technique versus the mean μ_{base} from the non-multiplexed run. MSE is defined as follows:

$$MSE = E[(\hat{\theta} - \theta)^2] = B[\hat{\theta}]^2 + var[\hat{\theta}] \qquad (9)$$

4.2 Results

Results show significant improvements for all benchmarks and all events for our rate-of-change approach, when compared with round-robin. Figure 4 compares accuracy in improvement for the two multiplexing strategies. For *instructions*, performance improved with blackscholes (22.1%) when compared with round-robin (8.5%). *cache-misses* had the biggest gain, with close to 95.5% accuracy for both the blackscholes and water_spatial benchmarks. *ref-cpu-cycles* also performed well with rate-of-change (Fig. 4b). Figure 5 shows accuracy rates for *L1-data-cache* events, including *hits*, *misses*, and *prefetches*. For all three events, our rate-of-change approach outperformed round-robin.

Table 4 shows percentage improvements for each benchmark when averaged across event types. The positive improvement rates indicate that our rate-of-change methodology has facilitated in profiling these applications better than the round-robin scheme.

Table 5 shows the improvements for each event type when averaged across benchmarks. Some of the top performers include *instructions*, *cache-misses*, and *node-prefetches*. However, there were also some poor performers. For instance, *branch-misses* had -10.23, while *iL1-misses* had -8.99. System perturbation across different runs may have skewed these numbers. It is possible that round-robin may have been sufficient for scheduling those events, and that our algorithm may have had an effect on those results. These averages only provide an overview of how certain events might behave on a particular benchmark.

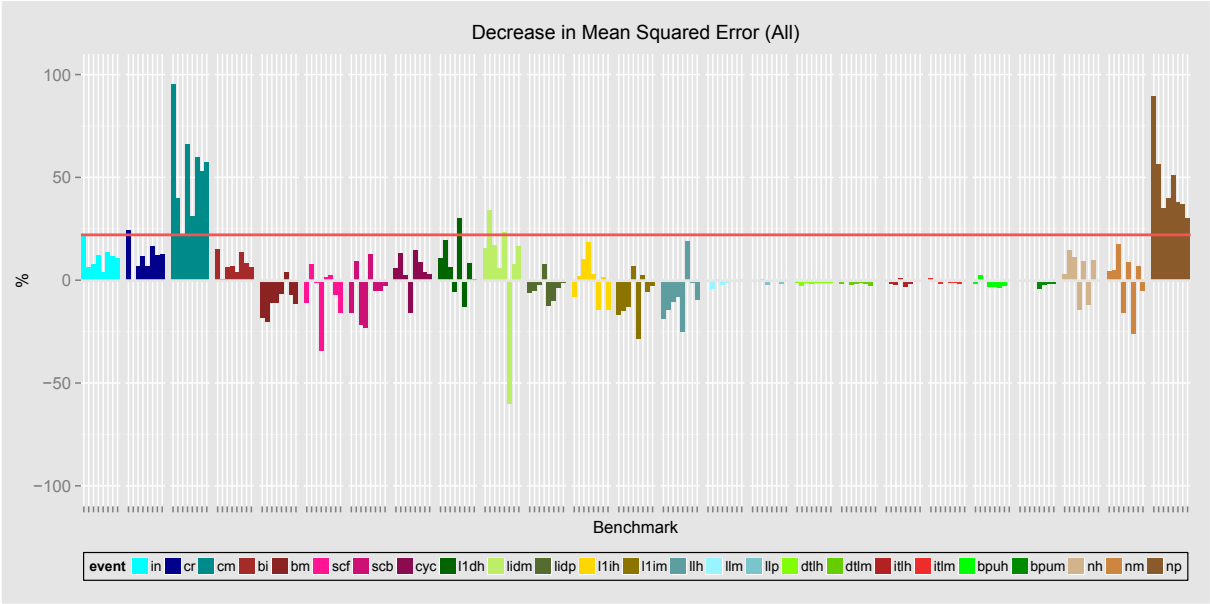

Figure 6: Decrease in MSE for all 25 multiplexed events, when comparing rate-of-change over the round-robin scheduling scheme. Each event set displays improvements for each of the eight benchmarks.

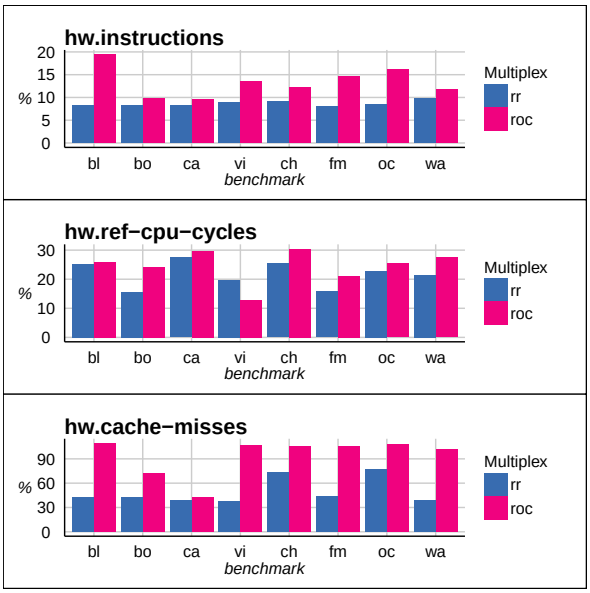

Figure 4: Accuracy in multiplexing strategies (round-robin, rate-of-change) with respect to baseline trace for select hardware events (more is better).

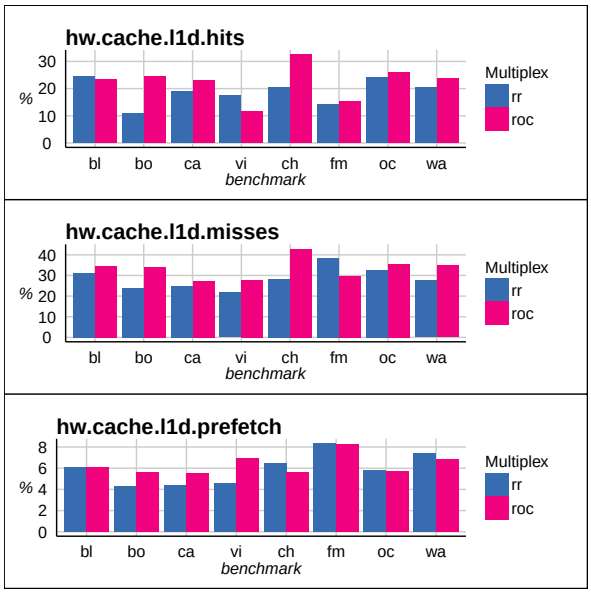

Figure 5: Accuracy in multiplexing strategies (round-robin, rate-of-change) with respect to baseline trace for *L1-data-cache* events (more is better).

instr	*c.ref*	*c.miss*	*br.in*	*br.mi*
10.96	11.35	53.04	7.48	-10.23
iTLB.h	*iTLB.m*	*st.cyc.f*	*st.cyc.b*	*ref.cyc*
-1.11	-0.82	-7.19	-6.44	4.4
dTLB.h	*dTLB.m*	*LLC.h*	*LLC.m*	*LLC.p*
-1.46	-1.59	-8.48	-0.95	-0.95
iL1.h	*iL1.m*	*dL1.h*	*dL1.m*	*dL1.p*
-0.26	-8.99	7.09	7.46	-4.01
bpu.h	*bpu.m*	*node.h*	*node.m*	*node.p*
-1.6	-1.58	2.71	-0.62	47.04

Table 5: Improvement (%) per event type, averaged over benchmark.

Figure 6 shows the proposed methodology's improvement as a decrease in mean-squared error for round-robin versus rate-of-change with respect to the baseline trace. Each event set shows performance for each of the eight benchmarks. The red horizontal line indicates that on average, performance gains of 22% were witnessed when comparing our rate-of-change approach with round-robin. With the exception of vips (*cycles*), cholesky (*iL1-misses*), and fmm (*dL1-misses*), most of the events profiled have shown substantial improvements. Some notable standouts include *cache-misses*: blackscholes (96%), vips (65%), cholesky (32%), fmm (59%); *cache-references*: blackscholes (24%), vips (12%), cholesky (7%), fmm (17%); and *node-prefetches*: blackscholes (89%), vips (40%), cholesky (51%), fmm (37%).

5 Future Work and Conclusion

5.1 Future Work

Our scheduling approach has demonstrated that performance measurement accuracy can increase when incorporating information about the behavior of a program task. Since architectural events are highly correlated in the same run (e.g. hardware interrupts, system calls), one extension to our multiplexing technique would be to incorporate correlated information into calculating the scheduling cost. Information about execution phases, in addition to temporal locality, can facilitate in creating an even more robust scheduler, which can lead to improved profiled accuracy rates. In addition, our multiplexing methodology can serve as an alternative to round-robin scheduling in other areas that utilize real-time decision-making, including task scheduling and decision-support systems.

5.2 Conclusion

Multiplexing multiple performance events is necessary because event counts of the same runs are more correlated to each other than among different runs. In addition, the limited number of m hardware counters provided by architecture manufacturers makes scheduling n performance events impractical, where $n > m$. Our rate-of-change scheduling algorithm has provided an increase of accuracy over the existing round-robin method, improving the precision of the total counts while allowing multiple events to be monitored.

6 References

References

[1] C. Bienia, "Benchmarking Modern Microprocessors," Princeton University, Jan. 2011.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," International Journal of High Performance Computing Applications, 14(3):189-204, June 2000.

[3] L. Campos and I. Scherson, "Rate of Change Load Balancing in Distributed and Parallel Systems," IEEE Symposium on Parallel and Distributed Processing. Apr 1999, doi:10.1109/IPPS.1999.760552.

[4] S. Eranian, "Perfmon2: A Flexible Performance Monitoring Interface For Linux," Proceedings of the Linux Symposium, vol.1, July 2006.

[5] T. Gleixner and D. Niehaus, "Hrtimers and Beyond: Transforming the Linux time Subsystems," Proceedings of the Linux Symposium, vol.1, July 2006.

[6] Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software\-developer-manual-325462.pdf. Intel Corporation, Sept, 2013.

[7] hrtimer for Event Multiplexing. https://lkml.org/lkml/2012/9/7/365. Linux Kernel Mailing List. Sept, 2012.

[8] W. Mathur and J. Cook, "Improved Estimation for Software Multiplexing of Performance Counters," 13th IEEE Intl. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05), 2005.

[9] J. May, "MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs," IEEE International Parallel and Distributed Processing Symposium, 2001.

[10] T. Mytkowicz, "Time Interpolation: So Many Metrics, So Few Registers," 40th IEEE/ACM International Symposium on Microarchitecture, 2007, doi:10.1109/MICRO.2007.27.

[11] NVIDIA nvprof Profiler. http://docs.nvidia.com/cuda/profiler-users-guide/#axzz33xUbGBm0. NVIDIA CUDA Toolkit v6.0.

[12] AMD Athlon Events. http://oprofile.sourceforge.net/docs/amd-athlon-events.php. OProfile website.

[13] Linux Kernel Profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial. Creative Commons 3.0, 2010.

[14] Qualcomm Adreno Profiler. https://developer.qualcomm.com/mobile-development/maximize-hardware/mobile-gaming-graphics-adreno/tools-and-resources. Qualcomm Developer Network.

[15] S. Shende and A. D. Malony, "The TAU Parallel Performance System," International Journal of High Performance Computing Applications, 20(2):287-311, Summer 2006.

[16] V. Weaver, D. Terpstra and S. Moore, "Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations," ISPASS Workshop, April 2013.

[17] V. Weaver, "Linux perf_event Features and Overhead," 2013.

[18] S. Woo, M. Ohara, E. Torrie, J. Singh, A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," International Symposium on Computer Architecture, 1995, doi:10.1145/223982.223990