The overhead of profiling using PMU hardware counters

July 2014

Authors:

Georgios Bitzes, Andrzej Nowak

CERN openlab Report 2014

Abstract

Run-time profiling of executable binaries can offer valuable insight into the performance characteristics and behaviour of a program. Some methods, such as instrumentation, are invasive and involve modifications of the profiled binary. This can significantly impact performance, to the point that an instrumented binary runs many times slower than the original. The Performance Monitoring Unit found in many modern processors offers the possibility of low-overhead profiling through a plethora of performance events. In this report, we investigate and quantify this overhead for a variety of tests and configurations, using the "perf" tool of the Linux kernel. Results for four main usage modes of the PMU are included: counting, sampling, PEBS events, and Last Branch Record (LBR).

Table of Contents

Abstract....................2

1 Introduction....................3

2 The Perf tool and the Linux kernel....................3

3 Test setup....................4

4 Results....................5

4.1 Counting mode....................5

4.2 Sampling mode....................6

4.3 Precise Event-Based Sampling (PEBS)....................10

4.4 Last Branch Record (LBR)....................12

5 Conclusions....................14

6 References....................14

7 Appendix A – Events....................15

8 Appendix B – Benchmarks....................16

1 Introduction

Modern processors are equipped with a Performance Monitoring Unit (PMU) which allows programmers to peek into the inner workings of the processor during the execution of their binaries [1]. From an extensive list, the user can select the performance-related events to monitor, such as cache misses, branch mispredictions or various other kinds of hardware stalls. The list of supported events varies depending on the manufacturer and the model of the processor.

Whenever any of the selected events occurs during execution, the processor increments the respective event counter inside the PMU. It is possible to instruct the processor to issue an interrupt whenever a counter exceeds a user-specified threshold – this enables recording the instruction pointer and examining the location inside the code that an event occurred. Using statistical sampling, we can now determine which piece of code consumes the most cycles or causes the most cache misses, among other things.

An important thing to note is the skid of the instruction pointer – by the time the interrupt is issued and caught, the instruction pointer is likely to have progressed and thus give a slightly inaccurate location of the code that triggered the event. This is possible to mitigate by having the processor itself store the instruction pointer (along with other information) in a designated buffer in memory – no interrupts are issued for each sample and the instruction pointer is off only by a single instruction, at most. This needs to be supported by the hardware, and is typically available only for a subset of supported events – this capability is called Precise Event-Based Sampling (PEBS) on Intel processors. The skid will generate a shadow, which will "hide" any occurring events. Such behaviour can be a particular problem in regular loops, where a recurring scenario could mask a large portion of events.

2 The Perf tool and the Linux kernel

Starting from version 2.6.31, the Linux kernel provides a formal interface for managing hardware counters through the "perf_event_open" system call, as well as a user-space tool which utilizes the kernel interfaces and acts as a profiler – called "perf".

Two main modes of usage are available – counting and sampling. Counting measures the overall number of events during the entire execution without offering any insight regarding the instructions or functions that generated them. On the other hand, sampling gives a correlation of the events to the code through captured samples of the Instruction Pointer. When sampling, the kernel instructs the processor to issue an interrupt when a chosen event counter exceeds a threshold. This interrupt is caught by the kernel and the sampled data – including the Instruction Pointer value – are stored into a ring buffer. The buffer is polled periodically by the user-space perf tool and its contents written to disk. In post-processing, the Instruction Pointer is matched to addresses in binary files, which can be translated into function names and such.

It's easy to see that while counting mode collects less information, it also incurs less overhead – there's no need for regular interrupts or writing to disk, and two value samples per counter should be enough to obtain the result.

Another important consideration is the number of counters that are available inside the PMU unit – as an example, in Intel Ivy Bridge micro-architecture there are three fixed counters that measure core cycles, reference cycles and core instructions as well as four programmable counters that can be assigned to any of the available events that the processor offers. Simultaneously measuring more than four events requires multiplexing by regularly rotating the set of events that are active at each moment. This incurs some further overhead.

When Hyper-Threading is disabled on modern Intel CPUs, however, the programmable counters of the unused hardware thread become available to the other – so there are eight available in total, per core.

A novel capability of recent processors is the Last Branch Record (LBR) with which it is possible to sample the last 16 branches, recording their source and target addresses. All this information is stored by the processor into a ring buffer. This feature on its own incurs no overhead, but due to the limited size of the ring buffer regular interrupts are still needed to sample and retrieve data from it.

3 Test setup

We used dual-socket systems equipped with Intel Ivy Bridge E5-2695 v2 @ 2.40GHz having 12 physical cores on each socket, running Scientific Linux 6.5 and version 3.11.6 of the Linux kernel. The systems were each equipped with 3 Intel SSDSC2CW240A3 SSDs in an LVM/stripping configuration.

All tests were run with Hyper-Threading enabled, except those where it is explicitly stated otherwise.

The benchmarks were chosen to mimic scientific workloads. Code was taken from the well-known SPEC 2006 suite [2], as well as the ROOT toolkit [3]. As all the applications are single-threaded, as most scientific code ran at CERN, multiple instances were launched in order to utilize all cores. Each process was pinned with `taskset` to its own core, ensuring an even distribution across the two sockets and utilizing SMT only when having run out of independent hardware threads. A separate perf process was used to monitor each instance – we observed no significant difference when all instances were monitored under a single perf process.

Another benchmark used was the multithreaded Geant4 prototype [4] – in this case, no pinning was applied and all threads were monitored under a single perf process. More information about the benchmarks can be found in Appendix B.

We benchmarked for a varying number of processes or threads, measuring each time the percentage difference between a monitored and unmonitored workload – this gives the overhead. As an example, on a workload with 12 processes, if the sum of the execution times of all processes is 100 without perf and 110 seconds while being profiled, the overhead is at 10%.

Turbo mode was disabled. Perf’s automatic throttling mechanism through `/proc/sys/kernel/perf_cpu_time_max_percent` was turned off – max sampling rate `perf_event_max_sample_rate` remained at its default value, 100000.

The list of events used is a mixture of hardware and resource stalls, as well as a few internal processor events. Not all of these events are supported in PEBS, so a different set was used with PEBS measurements. Both can be found in Appendix A. Event set #1 was used in all cases except where indicated otherwise. The periods employed were those that are recommended by expert tools [5] or commonly used in the industry [6].

Perf does not support supplying the symbolic event names as defined by the CPU manufacturers – using `UNHALTED_CORE_CYCLES` instead of "`0x3c`", for example. Libpfm [6] was used to aid in the translation of the event names to the raw hex codes that the CPU understands. This should not have any effects on performance as it's only done once during initialization, nevertheless we mention this as it is a part of the setup.

To sum up, we monitored overheads in several dimensions:

- Different profiling methods: counting, sampling, sampling with PEBS, LBR sampling
- Varying number of events: 1, 4, 8, 16
- Varying number of cores: 1, 4, 12, 24, 48
- Hyper-threading on or off
- Varying sampling periods (for LBR)
- Multiple nodes
- Different benchmarks from the SPEC, ROOT and Geant4 packages

Since there were many configurations, in some cases we chose to look for the most important data points. Overall, we tried to identify the most common usage scenarios and to quantify their performance. The penalties presented are cumulative penalties of the hardware and the perf subsystem that drives it. While it is not easily possible to distinguish the two sources, results suggest that the software system is the major source of overheads.

4 Results

The most important obstacle encountered in measurements was a relative instability of some of the runtimes, which would vary between runs on the same configuration. Some benchmarks varied more than others, and those with the greatest variations (+10% between runs on the same configuration) were altogether excluded from this report.

While great care was taken to ensure the stability and reliability of the results presented, there's still some uncertainty due to the inherent variations between runs, which could typically reach 0.5-2%. We ran each configuration three times and averaged, which should limit such effects – readers are however reminded that a small fraction of such measurements relies on chance.

Having said that, clear overhead trends, depending on the configuration, do emerge.

4.1 Counting mode

Figure 1 presents the average overhead incurred in counting mode depending on the number of instances and the number of events, averaged across all benchmarks. As can be seen this mode incurs a particularly low overhead, which for most intents and purposes can be considered negligible.

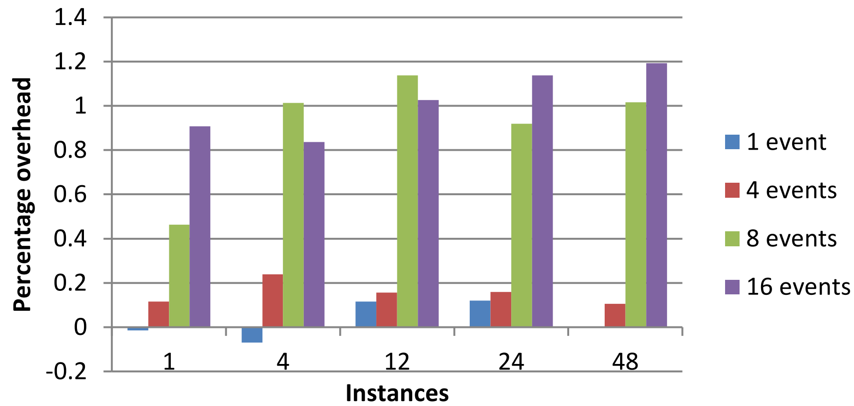

Figure 1. Average overhead in counting mode

The jump in overhead due to multiplexing is evident – Figure 2 presents a comparison between having Hyper-Threading on and off for a single benchmark, "444.namd". When disabled, the jump does not occur until 16 events (per core) as there are more hardware counters available and there's no need to multiplex at 8 events.

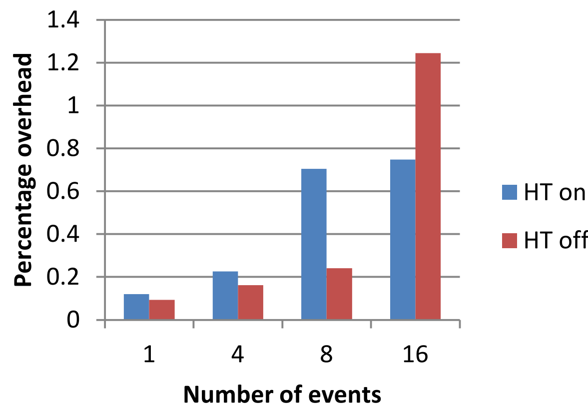

Figure 2. Effects of turning off Hyper-Threading – counting mode for 444.namd using 24 instances

4.2 Sampling mode

In this mode, we observed that the workload being profiled and the choice of events strongly influence the amount of overhead incurred. Figure 3 provides measurements for our most intensive case – sampling for 16 events when running 48 simultaneous instances.

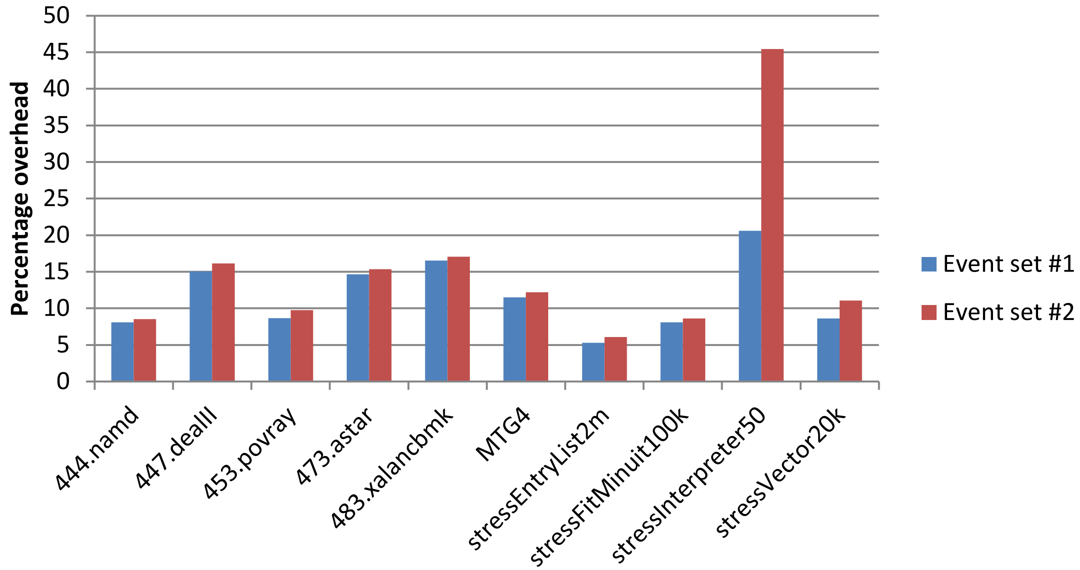

Figure 3. Overhead from sampling mode – 48 instances using 16 events

Figure 4 shows how the average overhead across all benchmarks behaves as we increase the number of instances.

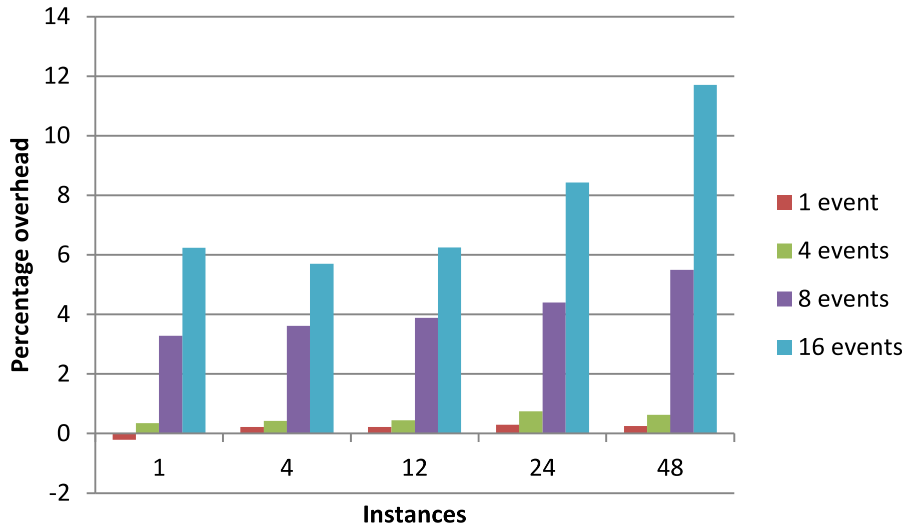

Figure 4. Average overhead across all benchmarks

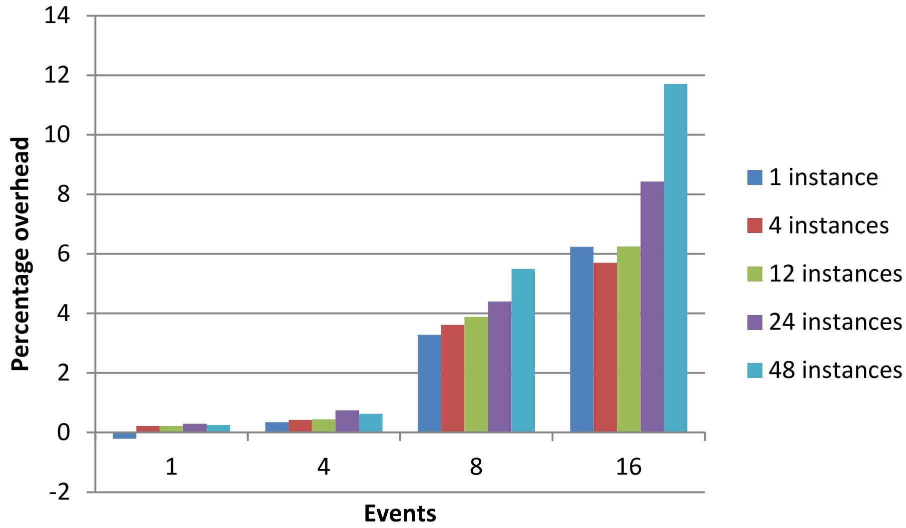

Figure 5. Average overhead across all benchmarks

Once again we see that the cost of multiplexing is very high, increasing the penalty by an order of magnitude. We find that turning off Hyper-Threading sharply reduces the overhead suffered at 8 events.

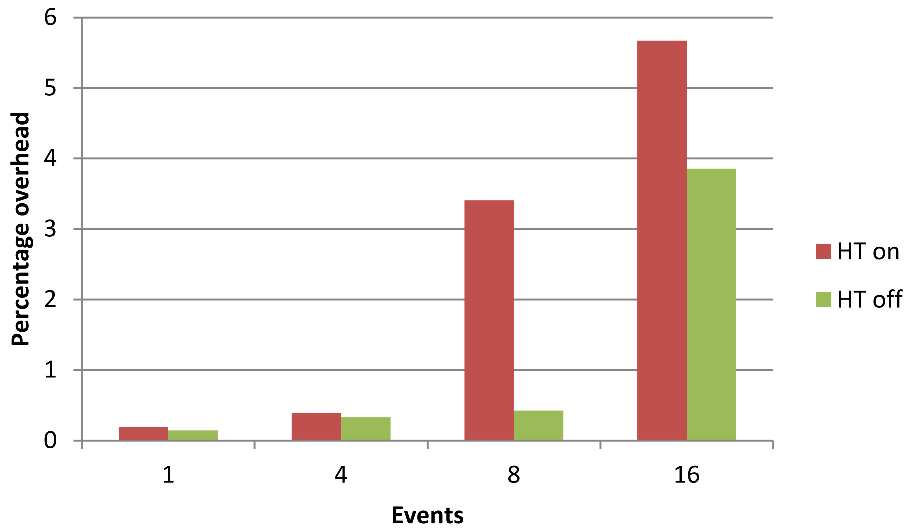

Figure 6. The influence of Hyper-Threading – sampling mode for 444.namd using 24 instances

Figure 7 presents a comparison across the number of events and instances just for a single benchmark, “447.dealII”.

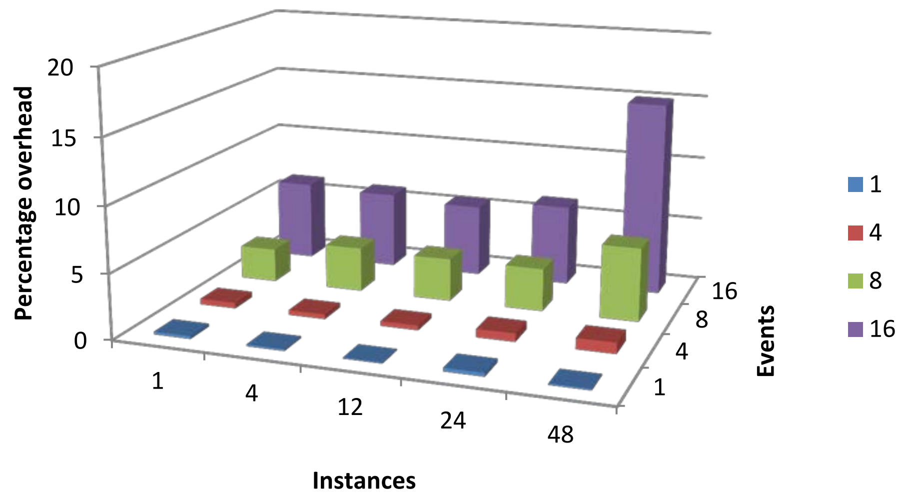

Figure 7. Sampling overhead for 447.dealII

Of course, the performance impact depends on the events being monitored and in particular, their periods. An event whose counter overflows often and for which many samples are collected will contribute to a higher overhead than a less commonly occurring event.

Figure 8 shows this clear trend – dividing the periods of all events by 10 and 100 increases sampling overhead.

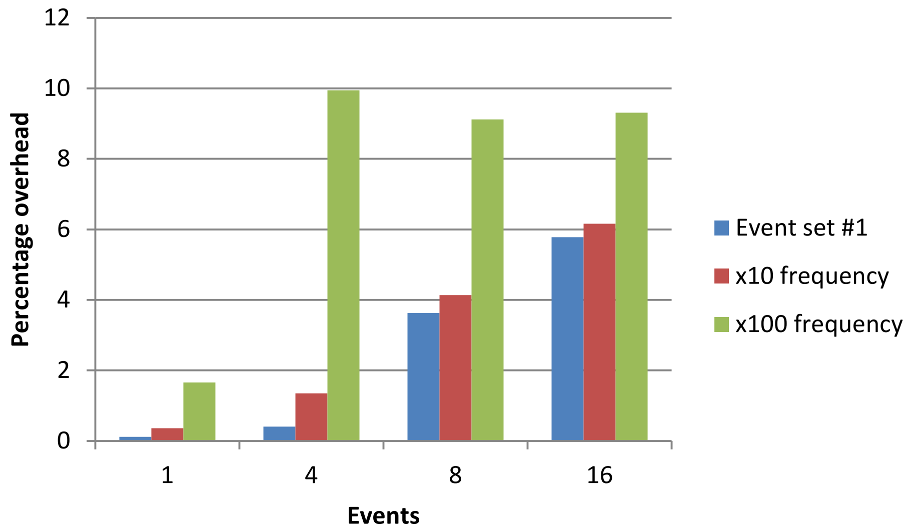

Figure 8. The influence of period on overhead – 12 instances, 444.namd

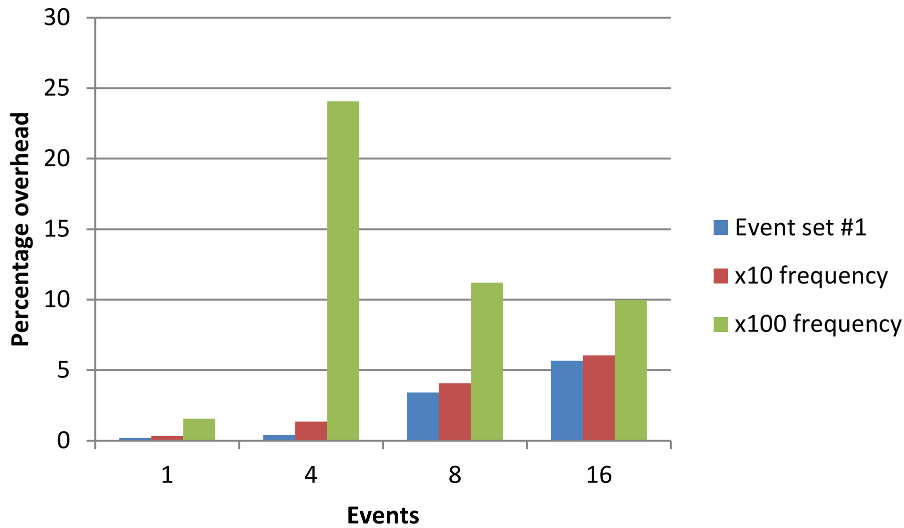

Figure 9. The influence of period on overhead – 24 instances, 444.namd

We encountered a peculiarity during 24 instances and 100x frequency – overhead dramatically increases to 24% only for 4 events. This is a strange result we could not adequately explain.

4.3 Precise Event-Based Sampling (PEBS)

Profiling using PEBS seems to incur identical overhead as regular sampling – the following figures show the average overhead across all benchmarks on the same event set both when utilizing the processors' PEBS mechanism and without.

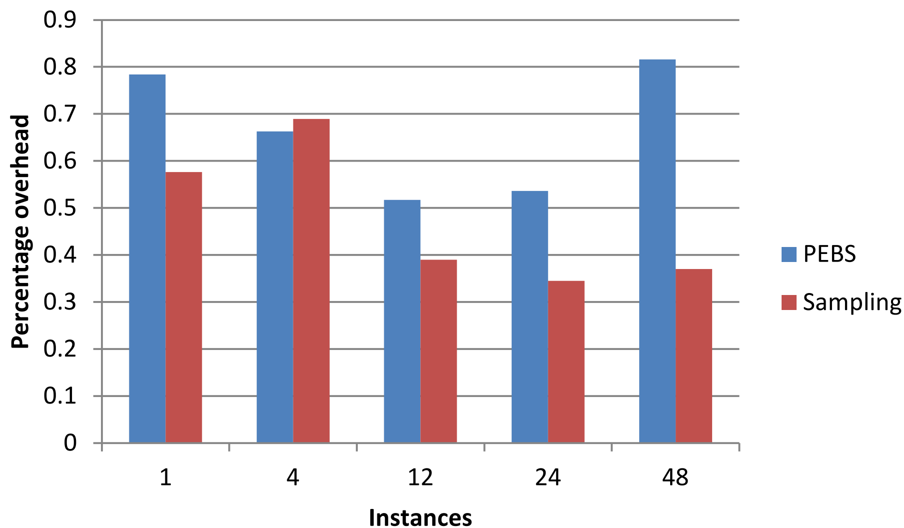

Figure 10. Sampling for 1 event on event set #2

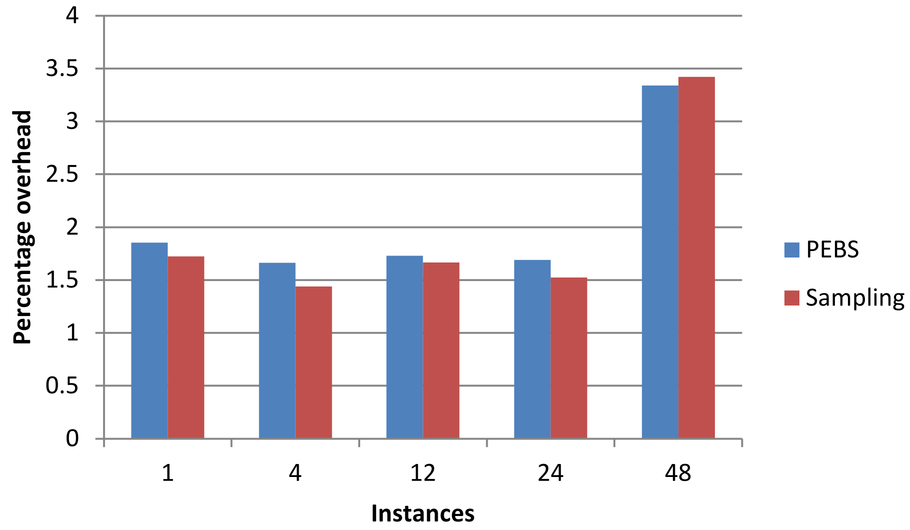

Figure 11. Sampling for 4 events on event set #2

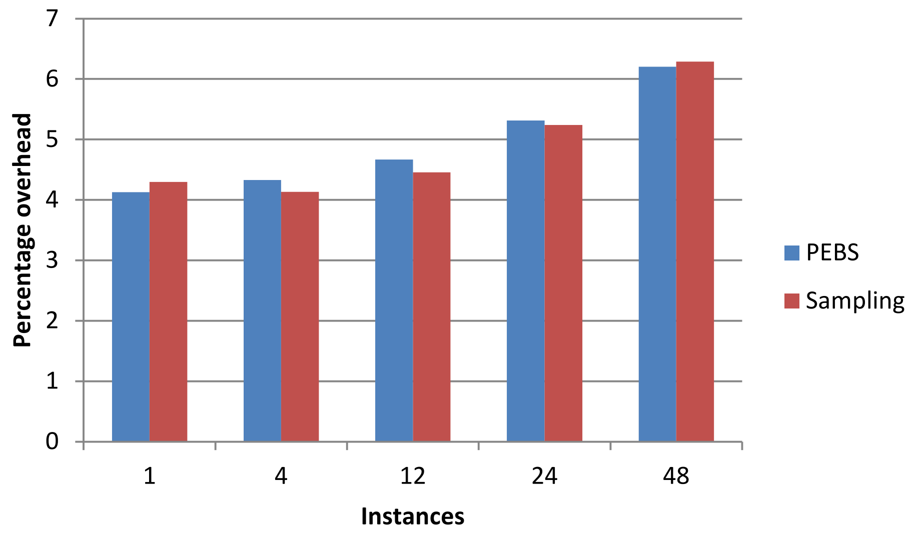

Figure 12. Sampling for 8 events on event set #2

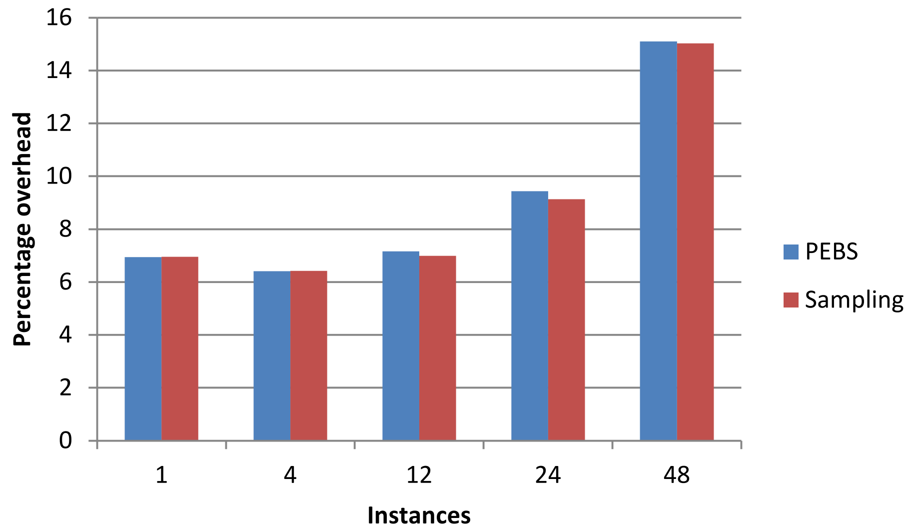

Figure 13. Sampling for 16 events on event set #2

This behaviour can be explained by the fact that the kernel, even in PEBS mode, still copies from the PEBS buffer on each sample by setting the overflow threshold to 1 record.[1] We predict that having the kernel copy samples in batch, rather than one at a single time, would significantly lower the impact of profiling with PEBS.

Therefore, these results do not necessarily reflect the inherent cost of sampling with PEBS, but rather the limitations of the current implementation in the Linux kernel.

4.4 Last Branch Record (LBR)

Last Branch Record was tested using only a single event each time – this is because of issues and limitations in perf. Average overhead of all benchmarks is presented below.

[1] See functions intel_pmu_drain_pebs_core and alloc_pebs_buffer in arch/x86/kernel/cpu/perf_event_intel_ds.c of the Linux 3.11.6 kernel

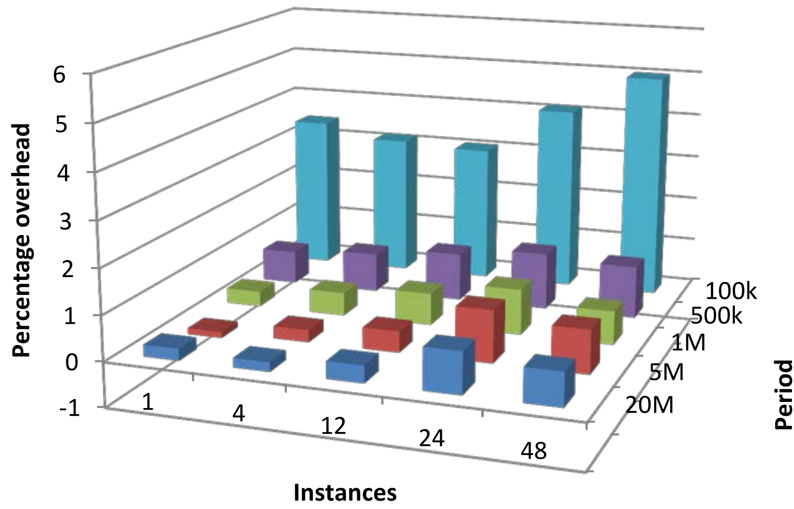

Figure 14. Average overhead for event rob_misc_events:lbr_inserts

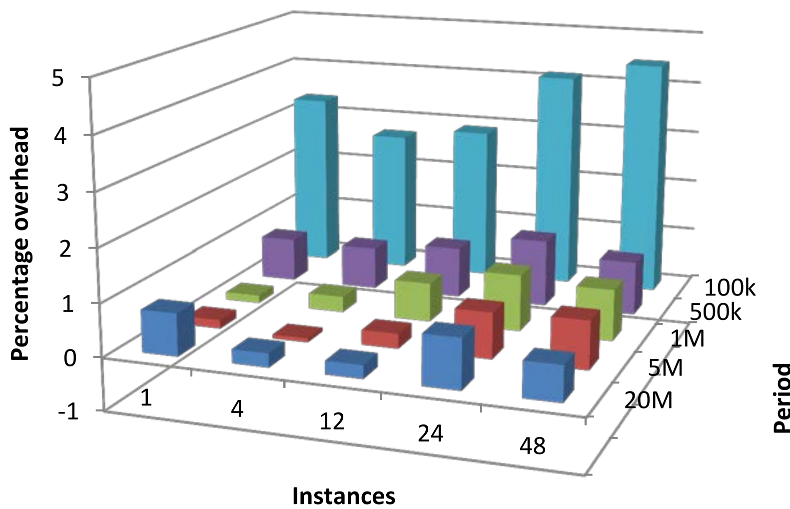

Figure 15. Average overhead for event br_inst_retired:near_taken

While this mode seems to incur only a little overhead, there's a jump when using a period of 100k – this is likely due to the very large amount of data that is collected during this configuration. As an example, Figure 13 presents the volume of data collected depending on the period per every 473.astar instance. At such point, the overhead incurred from I/O would be significant.

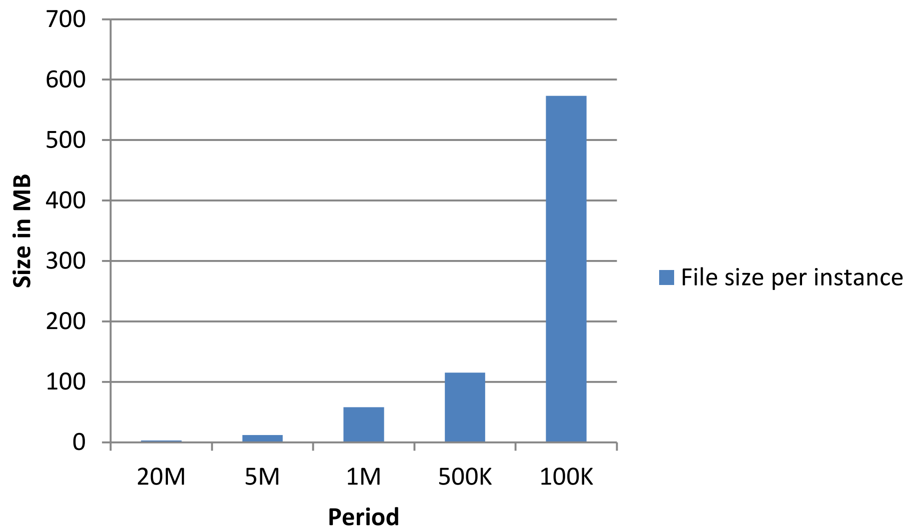

Figure 16. File size of sampled data per each instance of 473.astar (LBR)

5 Conclusions

We set out to measure the overhead incurred when profiling with PMU hardware counters. Results show how it can vary in many dimensions – the events selected, their frequencies, the workload, the number of events and in particular, the profiling method. An exact measurement of the hardware (vs. software) overheads would require more detailed instrumentation of the kernel – our results only demonstrate the overheads a performance tuner would see from their perspective.

One of the biggest sources of overheads seems to be multiplexing – users concerned with performance are advised not to use more events than there are hardware counters available, in particular in sampling mode. When multiplexing and with a very demanding set of counters of configurations, overheads can reach as much as 25% - a number more commonly seen when using software instrumentation to monitor workloads. This result suggests that there still might be room for optimization in the perf subsystem.

6 References

[1] “Intel 64 and IA-32 Architectures Optimization Reference Manual.” Intel Corporation, July 2013. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[2] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[3] R. Brun and F. Rademakers, “ROOT — An object oriented data analysis framework,” *Nucl. Instrum. Methods Phys. Res. Sect. Accel. Spectrometers Detect. Assoc. Equip.*, vol. 389, no. 1–2, pp. 81–86, Apr. 1997.

[4] X. Dong, G. Cooperman, and J. Apostolakis, "Multithreaded Geant4: Semi-automatic Transformation into Scalable Thread-Parallel Software," in *Euro-Par 2010 - Parallel Processing*, P. D'Ambra, M. Guarracino, and D. Talia, Eds. Springer Berlin Heidelberg, 2010, pp. 287–303.

[5] Google, *Gooda - a pmu event analysis package*. 2012.

[6] Intel Corporation, "Intel VTune Amplifier XE 2013," 2012. [Online]. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe. [Accessed: 22-Nov-2012].

[7] Libpfm library, http://perfmon2.sourceforge.net

7 Appendix A – Events

Event set #1

1. uops_retired:stall_cycles:period=20000000
2. itlb_misses:stlb_hit:period=2000000
3. uops_retired:any:period=20000000
4. resource_stalls:any:period=20000000
5. mem_load_uops_retired:l1_hit:period=20000000
6. rs_events:empty_cycles:period=20000000
7. itlb_misses:walk_completed:period=2000000
8. mem_load_uops_retired:l2_hit:period=2000000
9. itlb_misses:walk_duration:period=20000000
10. l2_rqsts:code_rd_hit:period=2000000
11. br_inst_retired:all_branches:period=2000000
12. baclears:any:period=2000000
13. cpl_cycles:ring0_trans:period=200000
14. l2_rqsts:code_rd_miss:period=1000000
15. arith:fpu_div_active:period=20000000
16. arith:fpu_div:period=2000000

Event set #2

All events in this list support PEBS.

1. uops_retired:stall_cycles:period=2000000
2. uops_retired:any:period=2000000
3. mem_load_uops_retired:l1_hit:period=2000000
4. mem_load_uops_retired:l2_hit:period=2000000
5. br_inst_retired:all_branches:period=2000000
6. br_inst_retired:cond:period=2000000
7. br_inst_retired:near_taken:period=2000000
8. br_misp_retired:all_branches:period=2000000
9. br_misp_retired:cond:period=2000000
10. br_misp_retired:near_taken:period=2000000
11. inst_retired:all:period=2000000
12. mem_uops_retired:all_loads:period=2000000

13. mem_uops_retired:all_stores:period=2000000
14. uops_retired:total_cycles:period=2000000
15. mem_uops_retired:stlb_miss_loads:period=2000000
16. mem_uops_retired:stlb_miss_stores:period=2000000

8 Appendix B – Benchmarks

Benchmarks taken from SPEC 06 suite: 444.namd, 447.dealII, 453.povray, 483.xalancbmk.

Benchmarks taken from ROOT framework tests:

Test	Command that was run
stressEntryList2m	`./stressEntryList 2000000`
stressFitMinuit100k	`./stressFit Minuit 100000`
stressInterpreter50	`./stressInterpreter 50`
stressVector20k	`./stressVector 20000`

Finally, the multi-threaded Geant4 prototype (version 9.6-ref09a) was run on geometry benchmark from the CMS experiment at CERN, simulating 100 pi- events per thread. Only the time spent in the scalable event loop was measured – initialization and finalization were excluded.