Predicting Program Phases and Defending Against Side-Channel Attacks using Hardware Performance Counters

Junaid Nomani   Jakub Szefer
Yale University
{junaid.nomani,jakub.szefer}@yale.edu

Abstract

Sharing of functional units inside a processor by two applications can lead to to information leaks and micro-architectural side-channel attacks. Meanwhile, processors now commonly come with hardware performance counters which can count a variety of micro-architectural events, ranging from cache behavior to floating point unit usage. In this paper we propose that the hardware performance counters can be leveraged by the operating system's scheduler to predict the upcoming program phases of the applications running on the system. By detecting and predicting program phases, the scheduler can make sure that programs in the same program phase, i.e. using same type of functional unit, are not scheduled on the same processor core, thus helping to mitigate potential side-channel attacks.

1. Introduction

Today, the scheduler has limited insights into the operation of the applications that it is scheduling, and detailed software-based analysis of programs may be prohibitively expensive. Thus, a scheduler has to make best-effort decisions about how to schedule applications.

With regard to security, scheduling can impact which applications run concurrently on which processor and thus can impact potential side-channels between the applications. With better knowledge of the operation of the applications, the scheduler could mix and match the applications to processor cores so that there is the least temporal sharing of specific functional units among applications running on the same processor. To achieve this, however, there needs to be a fast and low-overhead method for obtaining detailed information about the operations that the application is performing. In addition, once an attacker and victim application execute on the same processor and share a functional unit, there is already potential for side-channels. The scheduler thus needs to predict the upcoming operations and then use that information in scheduling to prevent side channels from occurring.

HASP '15 Portland, OR, USA
Copyright 2015 ACM. ISBN 978-1-4503-3483-9/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2768566.2768575

1.1. Security and Performance Benefits

In addition to having potential security benefits, non-sharing of functional units can give performance improvements as well. Using synthetic memory, integer and floating point benchmarks, we have analyzed how much performance can be gained, when comparing applications that fight for same functional unit, and when they are scheduled such that each is running on separate processor or core and thus using different functional units. We can observe improvements up to 25% in execution time when contention for hardware functional units is minimized. Today, however, the scheduler is not able to detect program phases of an application and may, for example, schedule two memory intensive applications on same the core, thus they will share the same cache. If instead it could detect the memory phases of the applications, it could schedule them on different cores, giving the 25% or more performance improvement. Similarly, integer, floating-point and other applications can gain performance if they are scheduled so as to minimize contention for specific hardware units.

1.2. Leveraging Hardware Performance Counters

We propose that the existing, and continually improved upon, hardware performance counters in processors can be used by the scheduler to learn and classify operations and phases of different applications to use for scheduling decisions. In particular, the scheduling decision may focus on how to help defend against temporal sharing of functional units by different applications, and thus help reduce side channels. Many hardware side-channels exist due to sharing of different functional units inside the processor [17]. The most well-known among these are the cache side-channels [14]. With our proposed new hardware performance counter enhanced scheduling, the scheduler could detect that two processes are memory intensive and attempt to schedule them on different processor cores. Now the attacker does not share the processor (and cache) with the victim at the same time and chances of a side-channel are reduced. Similarly, side-channel attacks based on sharing of floating point units, branch predictors, and others have been proposed [2]. Conveniently, there exist today performance counters that count exactly such events.

1.3. Performance Counters and Program Phases

Hardware performance counters, also called hardware performance monitors, are a set of registers which contain information about counts of different events occurring inside the processor. On Intel processors [1], performance-monitoring counters can be set, and then read based on the counter index, or event number. For example, `0x43` is the `data_mem_refs` counter which counts any memory loads and stores. Another counter, `0xC1` is the `flops` counter which counts a number of floating point operations. These events, however, can not all be measured at the same time. Usually, there are 2 or 4 performance counter registers, corresponding to the number of measurements can be collected in parallel [3]. There are also configuration registers that are used to specify which events should be counted in each counter.

Program phases represent the different periods of a program's execution, and have been shown to repeat as the application runs [10]. Program phases can include memory intensive phases, floating-point computation intensive phases, idle phases, etc. The program phases change as the program executes, e.g., memory phase to read in data, followed by integer computation phases when data is computed on. Each program phase also necessarily corresponds to some hardware in the processor, e.g., the floating point unit is only used in the floating point phase. Understanding program phases can be very powerful for scheduling, but also for detecting which piece of hardware inside the processor is being used by a particular application.

2. New Scheduler Architecture

This work proposes to augment the OS scheduler with the ability to read performance counter data, and then use machine learning approaches to detect and predict program phases. A high-level architecture is shown in Figure 1.

2.1. Threat Model

This work aims to protect systems against malicious applications, in particular ones which aim to exploit micro architectural side-channels. While applications are untrusted, this work assumes that the OS or hypervisor scheduler is trusted. The scheduler code should be also free of bugs and correctly implemented. The core Linux scheduler code is less than 10000 lines of code [7] and along with other scheduler code should be amiable to some level of software verification. The machine learning portion of the code in our prototype is done with help of the FANN C library [5] and the machine learning code is around 19000 lines of code, of which we only use a small percentage, which could be verified as well. The machine learning portion is a trusted user-level program that interfaces with the kernel. The hardware performance counters are assumed to correctly count events for which they are configured and too report these values acurately, as specified in the processor manufacturers' manuals.

2.2. Scheduler Architecture

The scheduler, show in figure Figure 1, collects performance counter data as each application runs. Right before an application is scheduled, the performance counters are reset, then they are configured to count desired events (different events can be counted for different applications), and the application runs. When the time quantum expires, or application yields, the scheduler first reads the performance counter values and stores them in the task structure for that application. After that it can select another application to run, and process repeats.

Asynchronously, the PMC module runs and reads the performance counter values for the applications, by accessing the task structures from the kernel. When it reads counters for an application, it communicates, via a netlink socket, to the user-level machine learning module.

The machine learning module implements a neural network that uses as inputs the history of performance counter values for the application. The neural network outputs a prediction of what is the next expected program phase.

Once the prediction is made, it is communicated back to the PMC module, which in turn can access the task structure and write the prediction into a new location in the task structure dedicated for that purpose. With the information stored in the task structure it can use the prediction in its scheduling decisions. The prediction is used only for placing the thread on the appropriate processor, and does not tamper with the CFS (Completely Fair Scheduler) in Linux which deals with the actual time slicing of the programs. Dissimilar tasks are migrated to be on the same processor, but there is a threshold to how many migrations can occur to prevent over-migration due to malicious or heavy loads. This maintains a balance between migration overhead, and security.

2.3. Machine Learning for Program Phases Prediction

The learning algorithm used was a 7 layer (5 hidden layers) feed forward neural net with basic gradient descent back propagation training [12]. The input layer received counter data for one counter for the last 15 context switches. This counter data was simplified to 5 clusters per counter using the k-means algorithm [6]. The centroids for each of the 5 clusters were unique to each program and based on that program's long term counter data. The input layer was 5*15 neurons, and each hidden layer was kept as 5*15 neurons as well while being fully connected. The output layer is 5 neurons, corresponding to the next predicted cluster for the next context switch for that counter. The training dataset was taken from running SPEC2006 benchmarks [11]. As a benchmark ran, the performance counters were recorded for an entire run and then used to train the neural net offline. This approach takes advantage of the fact that neural nets only need to be trained once per program, and although they have slow training, they have fast activation. Thus, although training time is in the order of minutes, prediction can be done in about 210us on our setup.

2.4. Timing Considerations

Machine learning has been considered for scheduling related approaches before, however, they have been mostly offline approaches and timing was assumed to be too long to be feasible. In Linux, on average an application runs for 2500us between context switches. In comparison, communication between the task structure and the machine learning module is on order of 5um. Inside our machine learning module, we use a feed-forward [12] type of neural network, which requires about 210us to make a prediction. Thus in one time quantum, predictions can be made for about 10 applications,

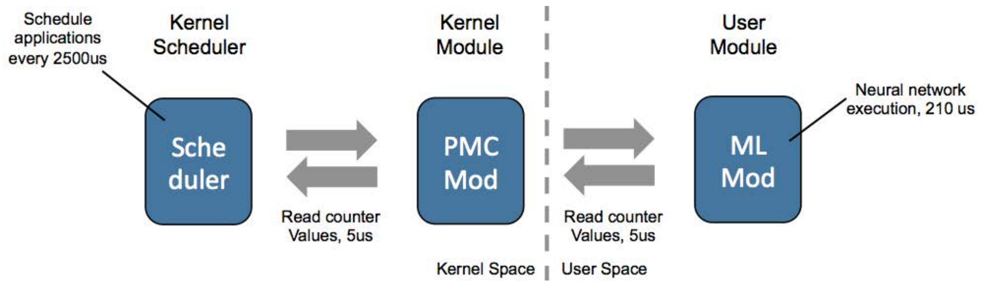

Figure 1: **Hardware performance counter enhanced scheduler with there components: base scheduler modified plus two new components. Arrows show data communication among components, while timing for select operations is shown for reference.**

and they all should finish before scheduler runs again. This is sufficient for commodity processors with up to 8 cores or threads. As more and more cores and programmable GPUs are available, our design allows to run prediction on multiple programs in parallel for even better performance.

3. Evaluation

The evaluation was carried out on Dell Precision T7600 workstation with Intel's Xeon E5-2609 running Linux 3.16.1. The various tests used synthetic benchmarks, as well as the SPEC2006 benchmarks [11].

3.1. Interference Evaluation

When applications use same functional units, there is interference that can be observed as changes in the performance of the application. Thus, observing changes in application performance is a proxy for observing the interference, which is in turn related to the capacity of the side-channels. A set of synthetic benchmarks was designed that included a memory based program which only has heavy random memory accesses, an integer-arithmetic only program that is not memory demanding, and a floating-point-arithmetic only program that also is not memory demanding.

We ran two groups of tests to check for interference. Programs had three categories: FP, INT, and INT-MEM, where FP is floating point intensive programs, INT are integer intensive programs, and the MEM are programs that are also very memory intensive. One group were very simple programs that ran single instructions in a loop to determine maximum possible interference, and the other were SPEC 2006 benchmarks (bzip2, mcf, milc) to determine more realistic interference from benchmarks that approximate real user applications.

Programs were ran alone to determine base speeds, and then ran in pairs for every combination. Programs ran roughly 25 percent slower in the simple group when two memory intensive programs were ran on the same core. Additionally, FP programs tended to interfere with other non-FP programs, for about 35 percent degradation. The SPEC group had similar but diluted results with around a 10-20 percent decrease in speed when there was memory contention or interference from FP programs. This shows possibilities for large interference from poor scheduling.

3.2. Predicting Program Phases

For fine grain prediction we use a neural network as described above, to make predictions at the context switch granularity. We train on all data points from counters for an entire `size=ref` run on the SPEC benchmarks.

As a comparison we used a simple last-only predictor that predicts the next context switch to have the same counter values as the last context switch. This last-only approach is actually quite common in literature. It works somewhat well because program phases usually span multiple context switches, meaning much of the time programs don't change behavior much from the last context switch. However, our approach has a noticeable decrease in mispredictions for the 12 SPEC benchmarks tried (astar, bzip2, dealII, gobmk, hmmer, lbm, libquantum, mcf, milc, namd, perlbench, povray). On average predicting memory with our predictor had an error rate of 30 percent, while the simple last-only predictor had an error rate of 50 percent, showing a significant reduction in error. For FP prediction, average results were roughly the same between predictors at 22 percent error rate. However, this is likely because most benchmarks had either no FP activity and so could be easily predicted as 0 FP operations throughout, or had very long and easy to predict program phases which skewed the results. So its not that our predictor is poor for FP operations, simply that for these benchmarks FP operations are usually in a phase that is exceptionally easy to predict.

4. Minimizing Side-Channels

Given a prediction about upcoming program phase, the scheduling algorithm can attempt to schedule applications such that different functional units on each processor are shared among applications least often. With a pool of applications to choose from, the scheduler can user the information about predicted program phase. For example, if two applications have an upcoming memory phase, they should be scheduled on different processor cores, so that they do not interfere with each other. But they can be scheduled on same core as an application with upcoming integer phase. This is done by assigning processes initially to spread memory intensive programs as evenly as possible, hopefully so that they each get their own core. This process then reoccurs periodically when the scheduler calls its normal load balance function, but thresholds are in place to prevent over-migration.

There is, however, a potential performance penalty from the fact that applications will be re-scheduled from processor core to processor core. For example, a memory intensive application may be moved among different cores, causing increased cache missies. This performance aspect needs to be further evaluated, however we hope by putting thresholds on frequency of migration this issue can be minimized in the common case.

5. Related Work

Earlier work [10] has shown that programs have reoccurring behavior identified as program phases. There are many ways to identify the phases, some requiring binary rewriting or analysis. Performance counters however are transparent and non-invasive while being accurate enough to discern phases. It has been shown by [15] that performance counters only have a variance up to 1% in the worst scenario which is very low for the purpose of discovering a program's phases. The precision of the performance counters allows them to be used for a wide variety of purposes such as to predict future behavior of applications [18]. Other papers either have prediction algorithms very similar to the last-only predictor we mentioned above [19] [8] [20], or use overly complex algorithms such as [9] that require offline analysis using complex models and even some by-hand optimization and could not be scaled to an online approach. Additionally no algorithm incorporated itself directly into the current Linux CFS scheduler, as we aim in our ongoing prototype work; most closely, [20] did modify an older and much simpler single queue scheduler. Performance counters have been also used for security purposes to detect malicious disturbances in program behavior [4], but have not been incorporated with a scheduler in this context. One example of scheduler modification for security includes work that shows that changing the minimum runtime guarantees in the scheduler thwarts a range of side channel attacks [13]. Others have also suggested that resource aware scheduling, similar to our proposed algorithm, could be useful at levels other than that of a program, e.g., at the cluster level or machine level [16]. Overall these other scheduling algorithms based on performance counters were effective, but our approach is generally more accurate compared to other online algorithms, is predictive rather than reactive, and by leveraging offline training, can be used for prediction online while maintaining its high accuracy.

6. Conclusion

Processors now commonly come with hardware performance counters which can count a variety of events. In this paper we showed that the hardware performance counters can be leveraged by the operating system's scheduler to predict the upcoming program phases of the applications running on the system and use the information to schedule applications such as to mitigate side-channel attacks. Our insight was to leverage the hardware performance counters to detect and predict different program phases inside the applications executing on a computer. By profiling the applications in real-time with the help of performance counters, the operating system's scheduler can learn what type of operations each application performs, and also learn to predict the phases of that application. We proposed to apply this prediction to schedule applications in a way such that may help defend against side-channel attacks by minimizing temporal sharing of functional units by different applications.

7. Acknowledgements

This work was supported in part through National Science Foundation's grant number 1419869.

References

[1] "Intel architecture software developer's manual volume 3: System programming," https://communities.intel.com/servlet/JiveServlet/previewBody/5061-102-1-8118/Pentium_SW_Developers_Manual_Vol3_SystemProgramming.pdf.

[2] O. Acimez, C. K. Koc, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Topics in Cryptology*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, vol. 4377, pp. 225–242.

[3] "Basic Performance Measurements for AMD Athlon 64, AMD Opteron and AMD Phenom Processors," http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Basic_Performance_Measurements.pdf.

[4] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the International Symposium on Computer Architecture*. ACM, 2013, pp. 559–570.

[5] "Fast artificial neural network library," http://leenissen.dk/fann/wp/.

[6] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani, *The elements of statistical learning*. Springer, 2009, vol. 2, no. 1.

[7] "Kernel scheduler and related syscalls," http://lxr.free-electrons.com/source/kernel/sched/core.c.

[8] A. Merkel and F. Bellosa, "Task activity vectors: a new metric for temperature-aware scheduling," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 1–12.

[9] M. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design, alexandra fedorova," 2005.

[10] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *Micro, IEEE*, vol. 23, no. 6, pp. 84–93, Nov 2003.

[11] "SPEC CPU 2006," https://www.spec.org/cpu2006/.

[12] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, pp. 43–62, 1997.

[13] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-vm side-channels," in *Usenix Security*, 2014.

[14] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the International Symposium on Computer Architecture*, ser. ISCA '07, 2007, pp. 494–505.

[15] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 141–150.

[16] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi 2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 379–391.

[17] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012, pp. 305–316.

[18] Z. Zhang and J. Chang, "A cool scheduler for multi-core systems exploiting program phases," *Computers, IEEE Transactions on*, vol. 63, no. 5, pp. 1061–1073, May 2014.

[19] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 129–142.

[20] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 4, 2012.