Processor Hardware Counter Statistics As A First-Class System Resource *

Xiao Zhang Sandhya Dwarkadas Girts Folkmanis Kai Shen
Department of Computer Science, University of Rochester

Abstract

Today's processors provide a rich source of statistical information on program execution characteristics through hardware counters. However, traditionally, operating system (OS) support for and utilization of the hardware counter statistics has been limited and ad hoc. In this paper, we make the case for direct OS management of hardware counter statistics. First, we show the utility of processor counter statistics in CPU scheduling (for improved performance and fairness) and in online workload modeling, both of which require online continuous statistics (as opposed to ad hoc infrequent uses). Second, we show that simultaneous system and user use of hardware counters is possible via time-division multiplexing. Finally, we highlight potential counter misuses to indicate that the OS should address potential security issues in utilizing processor counter statistics.

1 Introduction

Hardware counters are commonplace on modern processors, providing detailed information such as instruction mix, rate of execution (instructions per cycle), branch (control flow) prediction accuracy, and memory access behaviors (including miss rates at each level of the memory hierarchy as well as bus activity). These counters were originally provided for hardware verification and debugging purposes. Recently, they have also been used to support a variety of tasks concerning software systems and applications, including adaptive CPU scheduling [2, 6, 11, 15, 18], performance monitoring/debugging [1, 5, 19], workload pattern identification [4, 7], and adaptive application self-management [8].

Except for guiding CPU scheduling, so far the operating system's involvement in managing the processor counter statistics has been limited. Typically the OS does little more than expose the statistics to user applications. Additional efforts mostly concern the presentation of counter statistics. For instance, the PAPI project [5] proposed a portable cross-platform interface that applications could use to access hardware events of interests, which hides the differences and details of each hardware platform from the user.

In this paper, we argue that processor hardware counters are a first-class resource, warranting general OS utilization and requiring direct OS management. Our discussion is within the context of the increasing ubiquity and variety of hardware resource-sharing multiprocessors. Examples are memory bus-sharing symmetric multiprocessors (SMPs), L2 cache-sharing chip multiprocessors (CMPs), and simultaneous multithreading (SMTs), where many hardware resources including even the processor counter registers are shared.

Processor metrics can identify hardware resource contention on resource-sharing multiprocessors in addition to providing useful information on application execution behavior. We reinforce existing results to demonstrate multiple uses of counter statistics in an online continuous fashion. We show (via modification of the Linux scheduler) that on-line processor hardware metrics-based simple heuristics may improve both the performance and the fairness of CPU scheduling. We also demonstrate the effectiveness of using hardware metrics for application-level online workload modeling.

A processor usually has a limited number of counter registers to which a much larger number of hardware metrics can map. Different uses such as system-level functions (*e.g.*, CPU scheduling) and user-level tasks (*e.g.*, workload profiling) may desire conflicting sets of processor counter statistics at the same time. We demonstrate that such simultaneous use is possible via time-division multiplexing.

Finally, the utilization of processor counter statistics may bring security risks. For instance, a non-privileged user application may learn execution characteristics of other applications when processor counters report combined hardware metrics of two resource-sharing sibling processors. We argue that the OS should be aware of such risks and minimize them when needed.

2 Counter Statistics Usage Case Studies

We present two usage case studies of processor hardware counter statistics: operating system CPU scheduling and online workload modeling. In both cases, the

*This work was supported in part by NSF grants CNS-0411127, CCF-0448413, CNS-0509270, CNS-0615045, CNS-0615139, and CCF-0621472.

processor counter statistics are utilized in a continuous online fashion (as opposed to ad hoc infrequent uses).

2.1 Efficient and Fair CPU Scheduling

It is well known that different pairings of tasks on resource-sharing multiprocessors may result in different levels of resource contention and thus differences in performance. Resource contention also affects fairness since a task may make less progress under higher resource contention (given the same amount of CPU time). A fair scheduler should therefore go beyond allocating equal CPU time to tasks. A number of previous studies [2, 6, 10, 11, 15, 18] have explored adaptive CPU scheduling to improve system performance and some have utilized processor hardware statistics. The case for utilizing processor counter statistics in general CPU scheduling can be strengthened if a counter-based simple heuristic improves both scheduling performance and fairness.

In this case study, we consider two simple scheduling policies using hardware counter statistics. The first (proposed by Fedorova *et al.* [11]) uses instruction-per-cycle (IPC) as an indicator of whether a task is CPU-intensive (high IPC) or memory-access-intensive (low IPC). The IPC scheduler tries to pair high-IPC tasks with low-IPC tasks to reduce resource contention. The second is a new policy that directly measures the usage on bottleneck resources and then matches each high resource-usage task with a low resource-usage task on resource-sharing sibling processors. In the simple case of SMPs, a single resource — the memory bus — is the bottleneck.

Our implementation, based on the Linux 2.6.10 kernel, requires only a small change to the existing CPU scheduler. We monitor the bus utilization (or IPC) of each task using hardware counter statistics. During each context switch, we try to choose one ready task whose monitored bus utilization (IPC) is complementary to the task or tasks currently running on the other CPU or CPUs (we use last-value prediction as a simple yet reasonable predictor, although other more sophisticated prediction schemes [9] could easily be incorporated). Note that our implementation does not change the underlying Linux scheduler's assignation of equal CPU time to CPU-bound tasks within each scheduling epoch. By smoothing out overall resource utilization over time, however, the scheduler may improve *both* fairness *and* performance, since with lower variation in resource contention, tasks tend to make more deterministic progress.

Experimental results We present results on CPU scheduling in terms of both performance and fairness using two sets of workloads — sequential applications (SPEC-CPU2000) and server applications (a web server workload and TPC-H). The test is performed on an SMP

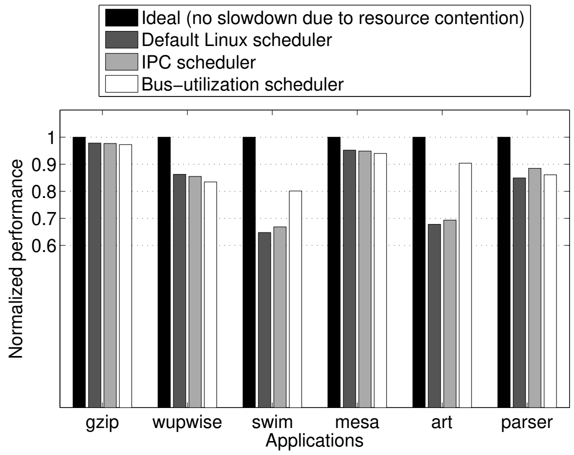

Figure 1: Normalized performance of individual SPEC-CPU2000 applications under different scheduling schemes.

system consisting of 2 Intel Xeon 3.0 GHz CPUs with Hyper-Threading disabled.

For experiments on SPEC-CPU2000 applications, we run gzip, parser, and swim (low, medium, and high bus utilization, respectively) on one CPU, and mesa, wupwise, and art (again, low, medium, and high bus utilization, respectively) on the other CPU. In this scenario, ideally, complementary tasks (high-low, medium-medium) should be executed simultaneously in order to smooth out resource demand. We define the normalized application performance as "$\frac{\text{execution time under ideal condition}}{\text{execution time under current condition}}$". The ideal execution time is that achieved when the application runs alone (with no processor hardware resource contention). Figure 1 shows the normalized performance of SPEC-CPU2000 applications under different schedulers.

We define two metrics to quantify the overall system performance and fairness. The *system normalized performance* metric is defined as the geometric mean of each application's normalized performance. The *unfairness factor* metric is defined as the coefficient of variation (standard deviation divided by the mean) of all application performance. Ideally, the system normalized performance should be 1 (*i.e.*, no slowdown due to resource contention) and the unfairness factor 0 (*i.e.*, all applications are affected by exactly the same amount). Compared to the default Linux scheduler, the bus-utilization scheduler improves system performance by 7.9% (from 0.818 to 0.883) and reduces unfairness by 58% (from 0.178 to 0.074). Compared to the IPC scheduler, it improves system performance by 6.5% and reduces unfairness by 55%. The IPC scheduler is inferior to the bus-utilization scheduler because IPC does not always accurately reflect the utilization of the shared bus resource.

We also experimented with counter statistics-assisted CPU scheduling using two server applications. The first

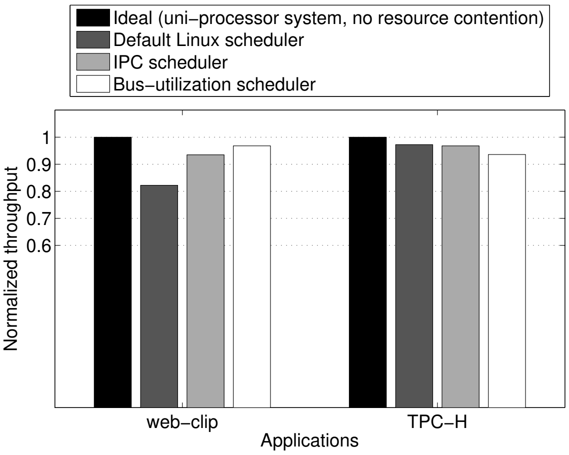

Figure 2: Normalized throughput of two server applications under different scheduling schemes.

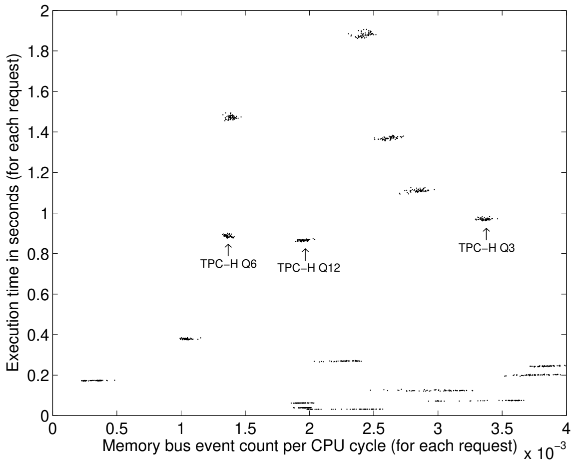

Figure 3: TPC-H request modeling using execution time and memory access intensity (measured by the Intel Xeon processor counter metric FSB_DATAREADY per CPU cycle).

is the Apache 2.0.44 web server hosting a set of video clips, synthetically generated following the file size and access popularity distribution of the 1998 World Cup workload [3]. We call this workload *web-clip*. The second application is the TPC-H benchmark running on the MySQL 5.0.17 database. We choose a subset of 17 relatively short TPC-H queries appropriate for an interactive server workload. The datasets we generated for the two workloads are at 316 MB and 362 MB respectively. For our experiments, we first warmup the server memory so that no disk I/O is performed during performance measurement. Figure 2 presents server throughput normalized to that when running alone without resource contention (ideal). Compared to the default Linux scheduler, the bus-utilization scheduler improves system throughput by 6.4% (from 0.894 to 0.952) and reduces unfairness by 80% (from 0.118 to 0.024). In this case, the IPC scheduler's performance is close to that of the bus-utilization scheduler, with system throughput at 0.951 and an unfairness factor of 0.025.

2.2 Online Workload Modeling

In a server system, online continuous collection of per-request information can help construct workload models, classify workload patterns, and support performance projections (as shown in Magpie [4]). Past work has mostly focused on the collection of software metrics like request execution time. We argue that better server request modeling may be attained by adding processor hardware counter statistics. We support this argument using a simple experiment. We ran our TPC-H workload for 10 minutes and monitored around 1000 requests. For each request, we collected its execution time and memory bus utilization (measured using processor hardware counters). The per-request plot of these two metrics is presented in Figure 3.

As can be seen from our result, the counter statistics can assist request classification. For instance, while TPC-H query Q3, Q6, and Q12 all exhibit similar execution time, they vary in their need for memory bandwidth, making it easy to distinguish them if this statistic is used. Further, hardware statistics can also help project performance on new computing platforms. For example, by migrating to a machine with a faster processor but identical memory system, a memory-access-heavy request (TPC-H Q3) would show less performance improvement than one with lower bandwidth requirements (TPC-H Q6). Software metrics alone may not provide such differentiation.

3 Managing Resource Competition

Although many hardware metrics can be configured for observation, a processor usually has a limited number of counter registers to which the hardware metrics must map. Additionally, the configurations of some counter metrics are in conflict with each other and thus these metrics cannot be observed together. Existing mechanisms for processor counter statistics collection [1, 17, 20] do not support competing statistics collections simultaneously (*i.e.*, during the same period of program execution).

Competition for counter statistics can result from several scenarios. Since system functions typically require continuous collection of a specific set of hardware metrics over all program executions, any user-initiated counter statistics collection may conflict with them. In addition, on some resource-sharing hardware processors (particularly SMTs), sibling processors share the same set of counter registers. Possible resource competition may arise when programs on sibling processors desire conflicting counter statistics simultaneously. Finally, a single application may request the use of conflicting counter statistics, potentially for different purposes.

In such contexts, processor counter statistics should be multiplexed for simultaneous competing uses and carefully managed for effective and fair utilization. The OS's basic mechanism to resolve resource competition is time-division multiplexing (alternating different counter register setups at interrupts) according to certain allocation shares. Time-division multiplexing of counter statistics has already been employed in SGI IRIX to resolve internal conflicts among multiple hardware metrics requested by a single user. Our focus here is to manage the competition between system-level functions and application-level tasks from multiple users.

The time-division multiplexing of hardware counter statistics is viable only when uses of counter statistics can still be effective (or sustain only slight loss of effectiveness) with partial-time sampled program execution statistics. Our allocation policy consists of two parts. First, the fraction of counter access time apportioned to system-level functions must be large enough to fulfill intended objectives but as small as possible to allow sufficient access to user-level tasks. Since the system-level functions are deterministic, we can statically provision a small but sufficiently effective allocation share for them. Second, when multiple user applications desire conflicting hardware counter statistics simultaneously (*e.g.*, when they run on sibling processors that share hardware counter registers), they divide the remaining counter statistics access time using any fair-share scheduler. We recognize that in order to be effective, certain applications might require complete monopoly of the hardware counters (*e.g.*, when a performance debugger must find out exactly where each cycle of the debugged program goes [1]). The superuser may change the default allocation policy in such cases.

Experimental results We experimentally validate that, as an important system-level function, the counter statistics-assisted CPU scheduler can still be effective with partial-time sampled counter statistics. Our CPU scheduler, proposed in Section 2.1, uses processor counter statistics to estimate each task's memory bus utilization. We install a once-per-millisecond interrupt which allows us to change counter register setup for each millisecond of program execution. In this experiment, the scheduling quantum is 100 milliseconds and we collect processor counter statistics only during selected millisecond periods so that the aggregate collection time accounts for the desired share for CPU scheduling.

Figures 4 and 5 show the scheduling results for SPEC-CPU2000 applications and server applications respectively, as the share of counter statistics collection time is varied. The experimental setup, metrics used, and applications are the same as in Section 2.1. In general, scheduling effectiveness depends on application behavior variability. Applications with high behavior variability may be difficult to predict even with more sophisticated non-linear table-based predictors [9]. However, for SPEC-CPU2000 applications, our results suggest that with only a 5% share of counter statistics collection time, the CPU scheduling performance and fairness approximates that attained with full counter statistics collection. For server applications, a 33% share is needed since the behavior of individual requests fluctuates and is more variable. These results indicate that the effectiveness of a counter statistics-assisted CPU scheduler may be fully realized with only a fraction of counter statistics collection time.

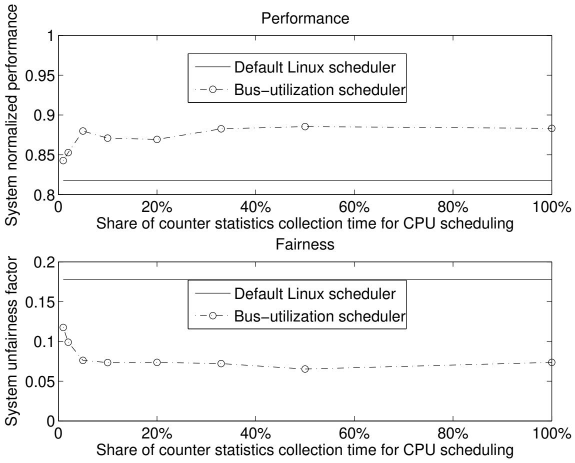

Figure 4: CPU scheduling performance and fairness of six SPEC-CPU2000 applications under partial-time sampled processor counter statistics. The "system normalized performance" and "unfairness factor" metrics were defined in Section 2.1.

4 Security Issues

Exposing processor counter statistics to non-privileged users and utilizing them in system functions may introduce new security risks. We examine two such issues.

Information leaks On resource-sharing multiprocessors, a program's execution characteristics are affected by behaviors of programs running on sibling processors. As a result, the knowledge of a program's own hardware execution characteristics may facilitate undesirable covert channel attacks [13] and side channel attacks for

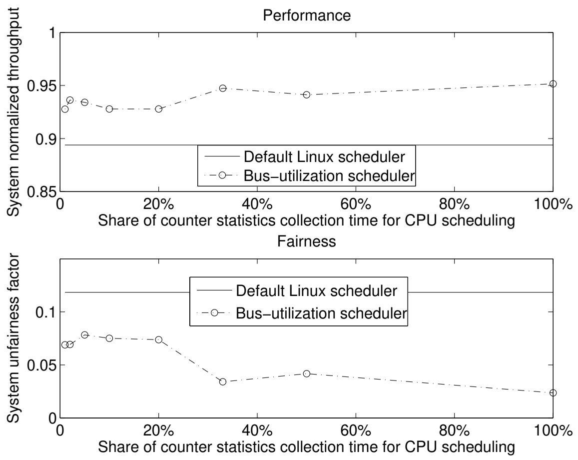

Figure 5: CPU scheduling performance and fairness of webclip and TPC-H applications under partial-time sampled processor counter statistics.

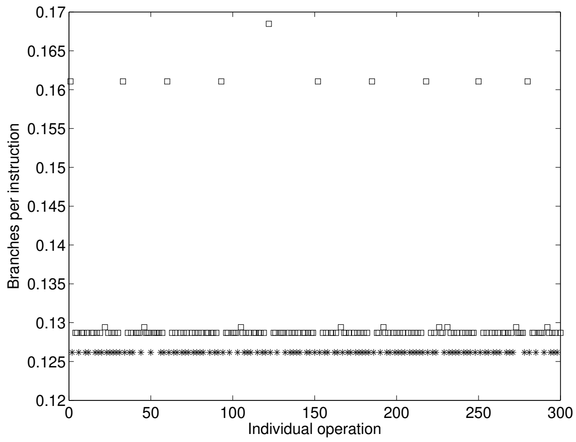

Figure 6: X^d is decomposed into a series of multiplications and squares in OpenSSL. Here X=1 and all d's are randomly generated with bit length no more than 6. Each point represents either a square (square symbol) or a multiplication (star symbol). The Y axis represents branches per instruction.

cryptanalysis [12].

Here we show a simple example of how the private RSA key in OpenSSL [14] may be stolen. One vital step in the RSA algorithm is to calculate "$X^d \bmod p$" where d is the private key and X is the input. In OpenSSL, modular exponentiation is decomposed into a series of modular squares and modular multiplications. By knowing the sequence of squares and multiplications, one can infer the private key d with high chance. For example, X^{11} is decomposed into $((X^2)^2 * X)^2 * X$. If one knows the execution order "sqr, sqr, mul, sqr, and mul", one can easily infer that the key is 11. Percival [16] showed that when a carefully constructed microbenchmark runs together with the RSA operation on the Intel Hyper-Threading platform, it may distinguish an RSA square from a RSA multiplication based on observed cache miss patterns. We show that this can be done more easily if certain hardware counter statistics are available. To demonstrate this, we run X^d exponentiation for many random private keys d and record the processor counter values for each individual multiplication and square operation. As we can see from Figure 6, the metric of branch instruction count divided by total instruction count (branches per instruction) provides a clear differentiation between the two types of operations.

The direct exposure of hardware counter statistics to non-privileged user applications may exacerbate existing risks in the following ways.

- Some statistics reported by hardware counters are combined event counts from multiple sibling processors (typically when the statistics concern shared hardware resources). Such statistics provide a direct way for a malicious program to learn information about other applications.
- Among those counter statistics that do not include event counts from sibling processors, some concern program execution characteristics that are affected by behaviors of programs running on sibling processors. An example is the L2 cache miss rate on L2 cache-sharing multiprocessors. These statistics can still be used to infer behavior patterns of other applications. Although such information may be learned at the software level (*e.g.*, through software sampling or probing), processor hardware counters expose the information with significantly higher accuracy while requiring almost no runtime overhead.

To prevent such undesirable information leaks, the system must be judicious in exposing hardware statistics to non-privileged user applications. Relevant hardware statistics can be withheld due to the above security concerns. Note that a fundamental tradeoff exists between security and versatility in information disclosure. In particular, withholding contention-dependent hardware statistics may impair an application's ability to employ adaptive self-management. Such tradeoffs should be carefully weighed to develop a balanced policy for hardware statistics exposure.

Manipulation of system adaptation When system functions utilize program execution characteristics for adaptive control, an application may manipulate such adaptation to its advantage by skillfully changing its execution behavior. For example, consider an adaptive CPU scheduler (like our bus-utilization scheduler proposed in Section 2.1) that uses partial-time sampled program execution characteristics to determine task resource usage levels and subsequently to run complementary tasks on resource-sharing sibling processors. A program may increase its resource usage only during the execution statistics sampling periods in order to be unfairly classified as a high resource-usage task. A possible countermeasure for the adaptive scheduler is to collect program execution characteristics using randomized sample periods. In general, the OS should be aware of possible manipulations of system adaptation and design counter-measures against them.

5 Conclusion

With the increasing ubiquity of multiprocessor, multithreaded, and multicore machines, resource-aware policies at both the operating system and user level are becoming imperative for improved performance, fairness, and scalability. Hardware counter statistics provide a simple and efficient mechanism to learn about resource requirements and conflicts without application involvement. In this paper, we have made the case for direct OS management of hardware counter resource competition and security risks through demonstration of its utility both within the operating system and at user level. Ongoing work includes development of the API and policies for hardware counter resource management within the kernel.

References

[1] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Trans. on Computer Systems*, 15(4):357–390, November 1997.

[2] C.D. Antonopoulos, D.S. Nikolopoulos, and T.S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *Proc. of the 32nd Int'l Conf. on Parallel Processing*, October 2003.

[3] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35, HP Laboratories Palo Alto, 1999.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modeling. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[5] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proc. of the IEEE/ACM SC2000 Conf.*, Dallas, TX, November 2000.

[6] J.B. Bulpin and I.A. Pratt. Hyper-threading aware process scheduling heuristics. In *Proc. of the USENIX Annual Technical Conf.*, pages 103–106, Anaheim, CA, April 2005.

[7] I. Cohen, J.S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, pages 231–244, San Francisco, CA, December 2004.

[8] M. Curtis-Maury, J. Dzierwa, C.D. Antonopoulos, and D.S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proc. of the 20th Int'l Conf. on Supercomputing*, Cairns, Australia, June 2006.

[9] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, New Orleans, LA, September 2003.

[10] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proc. of the USENIX Annual Technical Conf.*, pages 395–398, Anaheim, CA, April 2005.

[11] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer. Chip multithreading systems need a new operating system scheduler. In *Proc. of the SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[12] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2-3):141–158, 2000.

[13] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[14] OpenSSL: The open source toolkit for SSL/TLS. http://www.openssl.org.

[15] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, Department of Computer Science and Engineering, University of Washington, May 2000.

[16] C. Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, Canada, May 2005. http://www.daemonology.net/papers/htt.pdf.

[17] M. Pettersson. Linux performance counters driver. http://sourceforge.net/projects/perfctr/.

[18] A. Snavely and D. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *Proc. of the 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Cambridge, MA, November 2000.

[19] P.F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behaviors of java applications. In *Proc. of the Third USENIX Virtual Machine Research and Technology Symp.*, pages 57–72, San Jose, CA, May 2004.

[20] Intel VTune^{TM} performance analyzers. http://www.intel.com/software/products/vtune.