USENIX Association

Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Using Performance Reflection in Systems Software

Robert Fowler†, Alan Cox†, Sameh Elnikety‡, and Willy Zwaenepoel‡
† Department of Computer Science, Rice University, Houston, Texas, USA.
‡ School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland

Abstract

We argue that systems software can exploit hardware instrumentation mechanisms, such as performance monitoring counters in modern processors, along with general system statistics to reactively modify its behavior to achieve better performance. In this paper we outline our approach of using these instrumentation mechanisms to estimate *productivity* and *overhead* metrics while running user applications. At the kernel level, we speculate that the scheduler can exploit these metrics to improve system performance. At the application level, we show that applications can use these metrics as well as application-specific productivity metrics to reactively tune their performance. We give several examples of using reflection at the kernel level (e.g., scheduling to improve memory hierarchy performance) and at the application level (e.g., server throttling).

1 Introduction

Traditional processors have real-time clocks and interval timers. Today cycle counters and event counters are universally available in modern processors and chipsets. For example, the AMD Athlon processor has four performance monitoring counters (PMC) that can be programmed to count specific performance-related events such as TLB and cache misses. Moreover going beyond simple counters, the Alpha EV67 processor and its successors includes a ProfileMe [5] facility that can record the execution history of a single instruction as it passes through the pipeline. We expect that future processors and chipsets will have even more hardware instrumentation mechanisms.

The most common use of these mechanisms is by programmers to do performance debugging for application code. Also, these mechanisms have been successfully applied to the analysis and tuning of application code through compiler [8] and link-time optimizations, and applied to architecture evaluation. Despite the long history of using hardware instrumentation mechanisms, few studies focused on using them to reactively change kernel and application behavior.

We advocate using currently-existing hardware instrumentation mechanisms as the foundation of a *kernel performance reflection* facility designed to collect real time performance information to reactively modify operating system and application behavior.

The rest of the paper is structured as follows: Section 2 describes our approach of using performance reflection. We discuss examples of using performance reflection in OS kernels and in applications in sections 3 and 4, respectively. Section 5 presents related work. Finally, we present our conclusions in section 6.

2 Our Approach

We propose adding a performance reflection facility to the OS kernel to collect performance metrics using timers, event counters, and some programmed hooks. These metrics can estimate *overhead* and *productivity.*

First, some metrics represent costs: For example, the TLB and data cache miss rates measure the overhead that the system incurs in running the applications. We use these metrics to estimate *overhead.*

Second, some metrics count useful work: For example, the number of instructions executed, the floating point operation (FLOP) rate, bytes transferred to I/O devices, and the percentage of time the CPU spends in user mode are measures of useful work done. We use these metrics to estimate *productivity.*

We use the relationship between overhead and productivity to determine if there is a need to tune the system. Figure 1 shows three schematic plots that represent different relationships between overhead and productivity. In the first plot, both overhead and productivity are increasing, indicating that the load on the system is increasing and the system is behaving well. The second plot shows that the productivity is decreasing whereas the overhead is increasing. This corresponds to an undesired condition, such as thrashing when the system is in overload. Finally in the third plot, both overhead and productivity are decreasing, indicating a normal behavior as the system load decreases.

There are different ways of estimating productivity and overhead. It is not necessary to use any specific metric for these estimates. It is also possible to compute some metrics indirectly [1] if they are not available from the hardware. For example, the Cycles Per Instruction (CPI), which is a common measure of processor productivity, can be computed over some interval by taking the ratio of the number of cycles to the number of instructions graduated.

While many different kinds of events can be counted, the number of distinct measures of productivity counted either by hardware counters or the OS is small, perhaps including instructions, FLOPs, and bytes/packets transferred over I/O devices. Similarly, a small set of cost/overhead measures (cycles, L2 cache misses, TLB misses, interrupts) is sufficient. The kernel can use its own heuristics, or it can be guided by application advice provided through an interface similar to `madvise(3)`.

Productivity estimates can be enhanced with application cooperation. A variable shared between the application and kernel can be used by the application to inform the kernel of its rate of progress [6]. For example, a multi-threaded network server, such as a Web server or a file server, can use the number of requests served as its measure of progress and requests per unit time as its productivity metric.

3 Use of Performance Reflection in OS Kernels

The OS kernel can use the overhead and productivity metrics to reactively change its policies, such as changing the quantum size or the scheduling policy. In this section, we discuss a few applications of performance reflection in scheduling.

3.1 Memory Hierarchy Performance

Memory hierarchy performance can be very sensitive to competition on shared resources. For example, the standard configuration of IBM Regatta node has modules containing two Power4 processors that share a common cache and interface to main memory. Since it is known that many large scientific programs are memory-bandwidth bound, there is also an HPC variant of the hardware that contains only a single processor per module. For bandwidth-limited applications the second processor adds little or no additional performance and eliminating it saves cost while further eliminating possible cache interference. While it would not save the cost of the extra processors, monitoring miss rates of the shared cache of a standard node would enable the system to either schedule only one thread per module or to possibly identify “compatible” threads to co-schedule.

Similar scheduling strategies [9, 11] have been proposed for use with Simultaneous Multi-threading (SMT) [13].

The performance of non-uniform memory access (NUMA) machines is dependent on the assignment of threads to processors. The kernel can monitor memory behavior by, depending on the level of architectural support, measuring remote references, cache miss behavior, or cycles per instruction (CPI). Thread rescheduling decisions can then be based on this feedback.

4 Use of Performance Reflection in Applications

Applications can use the overhead and productivity metrics provided by the kernel, as well as

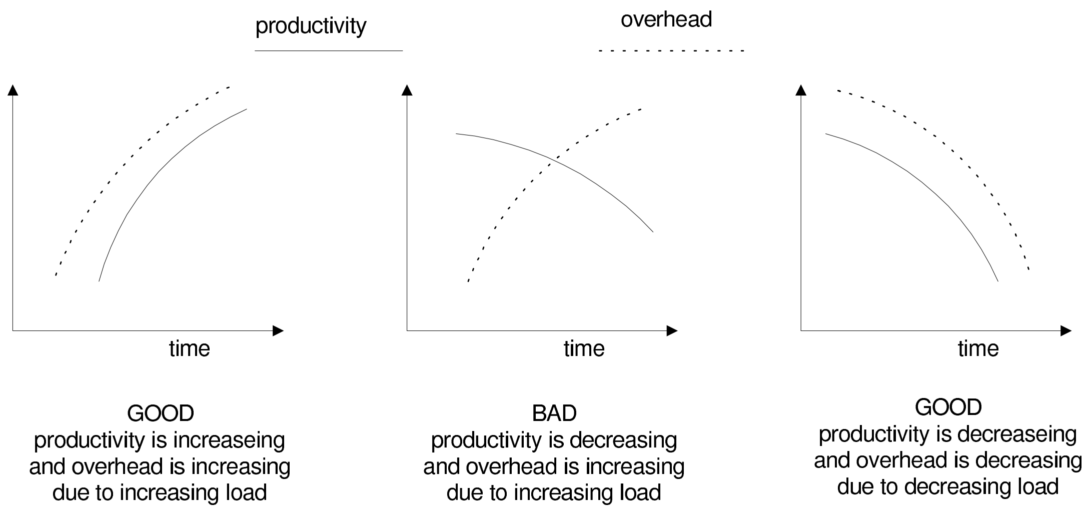
productivity
overhead
time
time
time
GOOD
productivity is increaseing
and overhead is increasing
due to increasing load
BAD
productivity is decreasing
and overhead is increasing
due to increasing load
GOOD
productivity is decreaseing
and overhead is decreasing
due to decreasing load

Figure 1: The relationship between overhead and productivity.

application-specific productivity metrics to reactively change their policies.

4.1 Server throttling

To demonstrate the feasibility of using reflection in applications, we show that our techniques can be used to do server throttling. Our experimental setup contains a MySQL database server running under Red Hat Linux 7.0 with the 2.4.18 kernel on an AMD Athlon 1.3 GHz processor. We used the shopping mix of the TPC-W [12] workload to drive the database server. Under this workload, the database server is the bottleneck. The database server thrashes when the number of concurrent queries is too high.

We developed a simple controller that uses the overhead and productivity metrics to dynamically determine the concurrency level of the database by controlling the number of active database connections. The controller receives all requests for database connections. It queues excess requests if the demand exceeds the number of available connections. When connections are released or more connections become available, queued requests are satisfied. The controller uses the PerfCtr [10] kernel module to read the number of L1 DTLB misses, L2 DTLB misses, and L1 and L2 data cache misses from the Athlon AMD processor [4].

The controller uses feedback from the kernel including the DTLB and data cache miss rates to estimate the overhead metric. It uses the percentage of user-mode CPU utilization and the throughput rate to estimate the productivity metric. The controller reads the input values every second, and keeps an exponential moving average of these metrics that spans the last 60 seconds to prevent transient oscillations. The controller uses a simple heuristic: It increases the number of database connections whenever both the productivity and overhead metrics increase, which corresponds to the situation where the CPU has idle time and low DTLB and data cache miss rates. The controller decreases the number of connections whenever the overhead metric increases and the productivity metric stagnates, which corresponds to the situation where the CPU is saturated and the DTLB and data cache miss rates are high.

Figure 2 shows the performance of the baseline system (without the controller) and of the system using reflection (through the use of the controller). The performance of the baseline system increases with the load until it reaches the peak plateau. Then, the performance degrades because of thrashing due to DTLB misses and data cache misses. As for the configuration that uses reflection, the database server is able to sustain peak throughput throughout the overload region by controlling the number of active connections to prevent thrashing.

Although, the controller prevented thrashing in the overload region, the dynamic behavior of the system is still not satisfactory. The controller uses that simple heuristic in an ad-hoc manner rather than a

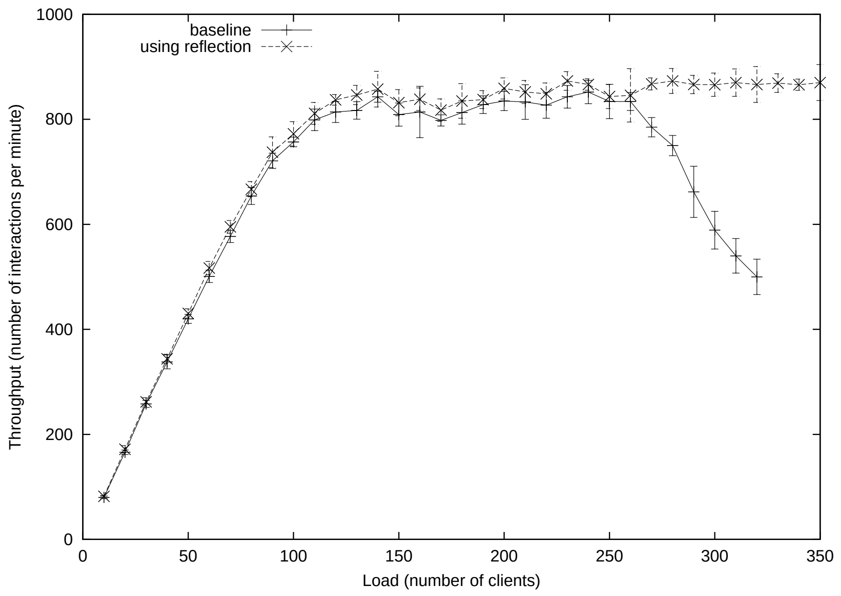

Figure 2: Server throttling for MySQL database server under TPC-W workload.

control-theoretic approach, which would guarantee stability and responsiveness. We believe that it is feaible to apply a control-theoretic approach that prevents thrashing and substantially improves the dynamic behavior of the system as the load changes.

5 Related Work

In this section we briefly mention representative work in areas where similar approaches are used.

Long term schedulers in batch systems have used page fault frequency (PFF) as the objective function for decisions to increase or decrease the multiprogramming level.

Our approach is complementary to Morph [17]. Instead of optimizing the performance by rewriting the binary code, we change the behavior of the system software without rewriting its binary code. We argue that it possible for both approaches to applied simultaneously because Morph-like optimizations optimize the code for a specific hardware or end-user pattern, whereas our approach addresses other performance issues, such as thrashing, which should be handled by specific policies (e.g., limiting the level of concurrency or changing the scheduling policy) rather than binary code optimization.

Douceur and Bolosky [6] used similar techniques to regulate low-importance processes. Our approach strives to maintain maximal performance by adapting the system behavior using both productivity and overhead metrics. This is in contrast to their approach where only productivity (progress) metrics are used to regulate low-importance processes such that they do not affect the execution of other processes.

SEDA [15, 14] presents a staged architecture for Internet servers. Our approach is another point in the design space of building systems that change their behavior adaptively to improve performance. SEDA offers greater control within each stage of a server; however, it requires a complete rewrite of the software. In contrast, our approach gives less control and requires far fewer modifications to the software.

The MAGNET [7] tool tracks OS events and exports information on them to user level. It has been used to identify Linux kernel problems (e.g., Ethernet driver, scheduler anomalies, overheads) and it has been used to analyze and tune applications, including creation of a reflective application.

In the Atlas [16] project, empirical techniques are used to tune the performance of some BLAS and LAPACK routines to provide portable performance.

Bershad et al. [3] used feedback from special hardware to dynamically avoid conflict misses in large direct-mapped caches by reassigning the conflicting virtual memory pages.

In the AppLeS [2] project, an application-level scheduler is used to adaptively and dynamically schedule individual applications on distributed, heterogeneous systems.

6 Summary and Conclusions

We discussed the use of hardware instrumentation mechanisms that are universally available on modern processors and chipsets as a basis for a performance reflection facility. Using this facility, it possible to estimate productivity and overhead metrics. Systems software can use these two metrics to improve its performance. We showed several potential uses: The OS kernel can use the metrics to tune its scheduling decisions. Applications can use these metrics to determine the concurrency level. Finally, we provided a working example for using reflection in server throttling to prevent thrashing.

References

[1] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[2] Fran Berman and Rich Wolski. The AppLeS Project: A Status Report. In *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, May 1997.

[3] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[4] AMD Corporation. AMD Athlon Processor x86 Code Optimization Guide. `www.amd.com`.

[5] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th Annual Symposium on Microarchitecture*, Research Triangle Park, North Carolina, December 1997.

[6] John R. Douceur and William J. Bolosky. Progress-based Regulation of Low-importance Processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999.

[7] Mark K. Gardner, Wu chun Feng, M. Broxton, A Engelhart, and G. Hurwitz. MAGNET: A Tool for Debugging, Analysis and Reflection in Computing Systems. In *Submitted to the third IEEE/ACM International International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003.

[8] John Mellor-Crummey, Robert Fowler, and Gabriel Marin. HPCView: a tool for top-down analysis of node performance. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, New Mexico, October 2001.

[9] Sujay Parekh, Susan Eggers, and Henry Levy. Thread-Sensitive Scheduling for SMT Processors. Technical report, University of Washington, 2002.

[10] Mikael Pettersson. PerfCtr home page. `http://user.it.uu.se/\~{}mikpe/linux/perfctr`.

[11] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[12] The Transaction Processing Council (TPC). TPC-W. `http://www.tpc.org/tpcw`.

[13] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[14] Matt Welsh and David Culler. Adaptive overload control for busy Internet servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2003.

[15] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.

[16] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):2–35, January 2001.

[17] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automatic Profiling and Optimization. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.