Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters

Reza Azimi
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
azimi@eecg.toronto.edu

Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
stumm@eecg.toronto.edu

Robert W. Wisniewski
IBM T. J. Watson Research Lab
Yorktown Heights
New York, USA
bobww@us.ibm.com

Abstract

Hardware performance counters (HPCs) are increasingly being used to analyze performance and identify the causes of performance bottlenecks. However, HPCs are difficult to use for several reasons. Microprocessors do not provide enough counters to simultaneously monitor the many different types of events needed to form an overall understanding of performance. Moreover, HPCs primarily count low–level micro–architectural events from which it is difficult to extract high–level insight required for identifying causes of performance problems.

We describe two techniques that help overcome these difficulties, allowing HPCs to be used in dynamic real–time optimizers. First, statistical sampling is used to dynamically multiplex HPCs and make a larger set of *logical* HPCs available. Using real programs, we show experimentally that it is possible through this sampling to obtain counts of hardware events that are statistically similar (within 15%) to complete non-sampled counts, thus allowing us to provide a much larger set of logical HPCs. Second, we observe that stall cycles are a primary source of inefficiencies, and hence they should be major targets for software optimization. Based on this observation, we build a simple model in real–time that speculatively associates each stall cycle to a processor component that likely caused the stall. The information needed to produce this model is obtained using our HPC multiplexing facility to monitor a large number of hardware components simultaneously. Our analysis shows that even in an out–of–order superscalar micro–processor such a speculative approach yields a fairly accurate model with run–time overhead for collection and computation of under 2%.

These results demonstrate that we can effective analyze on–line performance of application and system code running at full speed. The stall analysis shows where performance is being lost on a given processor.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ICS'05, June 20-22, Boston, MA, USA. Copyright ©2005, ACM 1-59593-167-8/06/2005...$5.00

1 Introduction

Hardware Performance Counters (HPCs) are an integral part of modern microprocessor Performance Monitoring Units (PMUs). They can be used to monitor and analyze performance in real time. HPCs allow counting of detailed micro–architectural events in the processor [15, 24, 13, 2], enabling new ways to monitor and analyze performance. There has been considerable work that has used HPCs to explore the behavior of applications and identify performance bottlenecks resulting from excessively stressed micro–architecture components [1, 8, 25]. However, exploiting HPCs at run–time for dynamic optimization purposes has proven to be challenging for a number of reasons:

Limited Hardware Resources: PMUs typically have a small number of HPCs (e.g. up to 8 in IBM PowerPC processors, 4 in Intel Itanium II, and 4 in AMD Athlon processors). As a result, only a limited number of low-level hardware events can be monitored at any given time. Moreover, only specific subsets of hardware events can be programmed to be counted together due to hardware–level programming constraints. This is a serious limitation considering that detecting performance bottlenecks in complex superscalar microprocessors often requires detailed and extensive performance knowledge of several processor components. One way to get around this limitation is to execute several runs of an application, each time with a different set of events being captured. Such an approach can become time–consuming for offline performance analysis, and is completely inappropriate for online analysis. Merging the traces for offline analysis generated from several application runs is not straightforward, because there are asynchronous events (e.g. interrupts and I/O events) in each run that may cause significant timing drifts.

Complex Interface: The events that can be monitored by HPCs are typically low–level and specific to a micro–architecture implementation and, as a result, they are hard to interpret correctly without detailed knowledge of the micro-architecture implementation. In fact, in the processors we have considered most high–level performance metrics such as *Cycles Per Instruction* (CPI), cache miss ratio, and memory bus contention, can only be measured by carefully combining the occurrence frequencies of several hardware events. At best, this makes HPCs hard to use by average application developers, but even for seasoned systems programmers, it is challenging to translate the frequency of particular hardware–level events to their actual impact on end performance due to the complexity of today's micro–architectures.

High Overhead: Because PMU resources are shared among all system processes, they can only be programmed in supervisor

mode. Thus, whenever a user process needs to change the set of the events being captured, it must call the operating system. These expensive kernel boundary crossings can happen frequently when a wide range of hardware events needs to be captured for a single run of the application.

In this paper, we describe techniques to address the above problems. First, to overcome the limitation in the number of HPCs, we use multiplexing of HPCs in a fine–grained way, instead of counting them fully. This method allows us to provide a much larger set of logical counters to the user, making it is possible to count the occurrences of many micro-architectural events during a single application run. A similar technique is implemented in PAPI [7, 20], but in our approach multiplexing HPCs is done in the operating system kernel so as to reduce run–time overhead significantly. As a result it is possible to multiplex HPCs at a much finer granularity (up to tens of thousands of HPC switches per second). Such a fine multiplexing granularity enables us to capture short–lived fluctuations in the hardware events occurrence rate. Moreover, we present a statistical analysis to show that our multiplexing approach provides sufficient accuracy for performance tuning and optimization purposes.

Secondly, we use our multiplexing approach to interpret the impact of different hardware events on the applications end–performance. We developed an aggregate model called *Statistical Stall Breakdown(SSB)* that provides accurate and timely information on which micro–architecture components are most stressed. SSB categorizes the sources of stalls in the microprocessor pipeline, and quantifies how much each hardware component (e.g. the caches, the branch predictor, and individual functional units) contributes to overall stall in a way that is simple and easy to understand for the user. SSB information is collected as the program runs and can be used, for example, by a dynamic optimizer to apply effective optimizations. The results of our analysis show that the SSB model can accurately identify and quantify hardware bottlenecks. Furthermore, we show that the run–time overhead of collecting the SSB information is small.

In the next section, we describe the design of our HPC–based performance monitoring facility and the features it provides. We follow this by describing the details of the SSB model and the computations it requires at run–time. We have implemented our ideas on a real system and evaluate the implementation under realistic workloads in terms of accuracy and run–time overhead. We present the platform we used for our implementation and describe the implementation issues we faced. We describe how we evaluate the sampling accuracy, the accuracy of the SSB model, and the run–time overhead of our facility. Finally, we discuss related work and then present our conclusions and directions for future work.

2 System Design

We designed and implemented an HPC–based performance monitoring facility that can be used with sampling and instrumentation. The key features of this facility are (i) it provides an easy–to–use interface to the hardware PMU features, and (ii) it uses statistical sampling to continuously identify microprocessor bottlenecks. Figure 1 shows the block diagram of our facility. Users are provided with a programming interface through a user–level library. Thus, an application can be instrumented by manually–inserted library calls or by dynamic instrumentation tools. Users' calls are received by the operating system component consisting of the programming interface module, and the sampling engine.

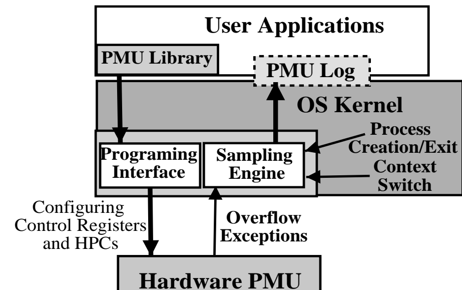

Figure 1. The block diagram of the HPC–based performance monitoring facility.

The sampling engine implements both HPC multiplexing and the the SSB model which we discuss in detail later. The programming interface module allows for both programming the PMU directly and configuring the sampling engine. In the former case, it receives the specification of a set of hardware events to be counted and automatically configures the hardware PMU. The values of the HPCs can be read directly by the user program, or logged in a per–process *PMU Log* by the sampling engine. To minimize the cost of user–kernel boundary crossings, the sampling engine operates as a component inside the kernel.

The sampling engine can obtain counter values either periodically or after a designated number of a hardware event occurrences. In both cases, we use PMU overflow exceptions. For periodic sampling we use one of the HPCs as the *cycle counter*, allowing sampling intervals accurate down to the CPU cycle.

The sampling frequency is a critical parameter. Sampling too infrequently may result in inaccuracies because changes in system behavior might be missed. On the other hand, too fine-grained sampling may result in unnecessarily high overhead. Our experience shows we can afford to take samples every 200,000 cycles (100 microseconds on a 2GHz CPU) with approximately 2% overhead. This rate is our default sampling frequency, although it can be overridden by the user.

In order to be able to isolate measurements of individual applications and the operating system, the sampling engine maintains a set of *HPC contexts*. HPC contexts are switched whenever the operating system switches processes. For this, the operating system must notify the sampling engine of all process creations and exits, as well as context switches.

For each process, there are three modes of operations: *kernel only, user only,* and *full system*. In kernel–only mode, hardware events are only counted when the hardware is in supervisor mode. This mode is appropriate if we are interested in monitoring operating system activities incurred by a particular target process. We assume kernel activities that occur in a process time slice are related to the target process. This assumption may not be valid when several operating system intensive applications tightly share the CPU. This, kernel–only mode is best suitable when a given application runs in isolation for a long time (on the order of several seconds). In user–only mode, logical HPCs (including the cycle counters) are suspended when the processor switches into the kernel. Finally, in full–system mode, HPCs count all hardware events whether due to kernel or application code. When a context switch occurs, the hardware events occurring both in the kernel and user mode will be counted by the HPCs of the new process.

We use the notion of an address space as the main indicator of a context. Therefore, the sampling engine is capable of reporting performance numbers for individual processes as well as the operating system. At this time, we do not differentiate between the user–level threads that share the same address space. One possible way of addressing this issue is to send a performance monitoring upcall to the user process when a hardware exception occurs so that a user–defined handler can associate the recorded HPCs with the current user-level context (e.g. user-level thread ID). Such a technique seems to be plausible only if there is a fast (low perturbation) upcall delivery mechanism. We are currently working on supporting this.

3 Multiplexing HPCs

Most microprocessors' PMU offer a limited number of HPCs. For instance, the IBM POWER4 and PowerPC970 provide 8 HPCs, the POWER5 6 per SMT (2 of which are hard-wired), Intel Itanium II has 4 generic HPCs and 5 registers for holding instruction and data address samples, and AMD Athlon has 4 generic HPCs. In addition to the limited number of HPCs, there are often restrictions on the combinations of hardware events that the HPCs can count due to restrictions on how the PMU is interconnected to the CPU components. For instance, although Intel P4 and Xeon have 18 HPCs, but they are divided in 9 groups, each of which can count events only from a specific subset of 48 available hardware events.

In many performance monitoring scenarios, several low–level hardware events must be counted simultaneously to obtain information of interest. For instance, to obtain the L1 data cache miss rate on an IBM POWER4 processor, at least 4 separate events must be counted (L1 load misses, L1 store misses, L1 loads, and L1 stores). Also, two HPCs are often reserved to count cycles and instructions retired. The two remaining HPCs are not sufficient to count L2 cache misses, L3 cache misses, branch mispredictions, TLB misses, instruction mix (integer/floating point), ICache misses, or other events that are important for obtaining a complete picture of an application's performance.

To address this issue, we dynamically multiplex the set of hardware events counted by the HPCs using fine–grained time slices. The programming interface component takes a large set of events to be counted as input and automatically assigns them to a number of HPC groups such that in each group there are no conflicts due to PMU constraints. The sampling engine assigns each group a fraction of g cycles out of a sampling round R that is the time period that all groups are guaranteed to have been given a PMU context. At the end of each HPC group's time slice, the sampling engine automatically assigns another HPC group to the PMU. The value that is read from an HPC after g cycles is scaled up linearly as if that group it had counted during the entire R–cycle period. As a result, the user program (e.g. a run–time optimizer) is presented with N *logical* counters on top of n *physical* HPCs where N can be an order of magnitude larger than n.

The system can easily be programmed to favor certain HPC groups by counting them for longer periods of time. This is accomplished by allocating multiple g-cycle time slices to the group. In fact, one can treat a period of g cycle as a unit for PMU time allocation. This PMU multiplexing scheme is analogous to the time–sharing of a CPU among processes. Figure 2 shows an example of four event groups, where each is given a time share (one or more time slices) of the sampling period. The share size of each group depends on the desired accuracy of the hardware events that are included in the group and on the expected rate of fluctuation of such events. Moreover, the accuracy may differ for different hardware events with the same share size. A default share assignment scheme might be overridden by explicit requests from the user that is interested in closely monitoring a specific hardware event.

Without loss of generality, for the rest of the paper, we assume all groups are given equal time shares, which is one time slice (g cycles). We call $\frac{R}{g}$ the *Sampling Ratio*. Larger sampling ratios allow a larger number of logical HPCs. For instance, a sampling ratio of 10 can provide roughly 80 logical HPCs on an 8-HPC processor. This has to be traded–off with the fact that sampling accuracy decreases as the sampling ratio increases.

An issue that must be addressed is the fact that a sampling period may happen to coincide with loop iterations in the program. If the order of HPC groups within a period is fixed and a sampling period happens to coincide with a loop iteration, then an HPC group always counts the events occurring in a fixed part of the iteration. To avoid this scenario, we randomize the order of the HPC groups in each sampling period. As a result, each HPC will have an equal chance of being located at any given spot of the iteration.

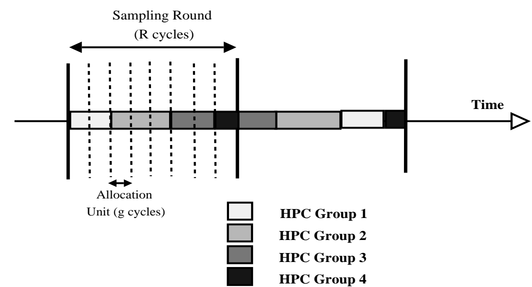

Figure 2. Time-Based Multiplexing example: There are four HPC groups in this example. Each HPC group is a collection of events that are counted simultaneously. An HPC group is counted in a number of time slices of g cycles within sampling period of R cycles. The order of the HPC groups is changed randomly in different sampling periods to avoid accidental correlations.

With multiplexing, time is usually measured in terms of CPU cycles. Therefore, one counter in each HPC group is reserved to count the CPU cycles. The use of cycle counters as timers allows us to define arbitrary fine time–slices down to a few thousand cycles. Another metric that can be used to define HPC group share sizes is the *number of instructions retired*. The main advantage of instruction–based multiplexing is that the HPC group share sizes are aligned more closely with the progress of the application. Share sizes will differ in terms of real time depending on the available instruction level parallelism (ILP) and the frequency of the miss events.

A pathological case for the multiplexing engine is the existence of a large number of short–lived bursts of a particular hardware event. If the burst time is shorter than R cycles, the HPC that counts that hardware event might be inaccurate because the PMU actually counts the event only in a fraction of R, and thus it may miss short–lived bursts. However, we expect the execution of most applications to go through several phases, each longer than R, in which the occurrence rate of hardware events is stable in the common case. In Section 6, we provide experimental evaluation that demonstrates that the statistical distance between the sampled and real rates of hardware events is small in most cases.

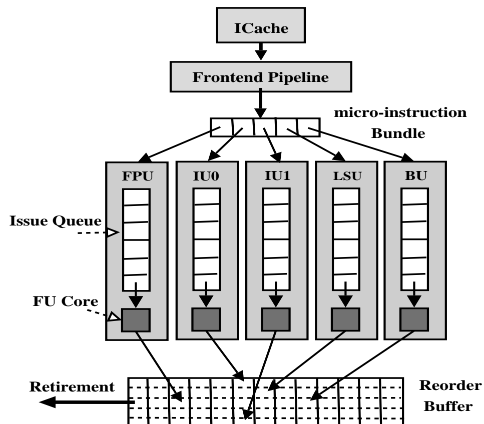

Figure 3. The basic hardware model for a super-scalar multi-dispatch out–of–order processor. FPU stands for Floating-Point Unit, IU stands for Integer Unit, LSU stands for Load/Store Unit, BU stands for Branch prediction Unit, and FU stands for Functional Unit.

4 The SSB Model

Traditionally, CPI breakdown has been used to describe the performance characteristics of hardware components and program behavior [11]. In CPI breakdown, each hardware component h accounts for CPI_h cycles per instruction out of the real CPI on average. However, in the context of complex superscalar out–of–order microprocessors with multi–level memory hierarchies, it is extremely difficult to obtain accurate CPI breakdown because many micro–architectural events overlap with each other.

We propose an alternative to the CPI breakdown model which we call *Statistical Stall Breakdown* (SSB). By *stall*, we mean a processor cycle in which no instruction retires. In SSB, each stall is speculatively attributed to a particular hardware event, where if that hardware event had not occurred, the stall would not have occurred. The key observation is that, in most cases, the latency of hardware components in processing instructions will result in stalls. In Section 6.3 we show that stalls are one of the primary contributions to the often dramatic gap between the measured CPI and the *ideal* CPI that is defined by (i) available ILP in the applications and (ii) the system pipeline width defined by the micro–architecture. Therefore, an accurate breakdown of stall sources can be used as an approximation for hardware bottleneck analysis.

The basic hardware support required for computing SSB is to have a way to assign a source to each stall. While such an assignment may be speculative (mainly due to the fact that stalls from different sources may overlap), our analysis in Section 6.3 shows that such speculation is sufficiently accurate in most cases.

We collect SSB statistics at run–time using a simple model for a superscalar, out–of–order processor. Our model is influenced by the architecture of the IBM PowerPC processor, but it is sufficiently generic to be used for other modern microprocessors with minor modifications.

Figure 3 depicts the hardware model used. Instructions are fed from the ICache to the front–end pipeline in program order (Figure 4 depicts the state–transition diagram for each instruction.). Up to W ISA (Instruction Set Architecture) instructions can be fetched from the ICache in each cycle. These instructions are decoded and possibly translated into μ–instructions. The front-end pipeline generates *bundles* of B μ–instructions, associated with one or more ISA instructions. μ–instructions within a bundle may depend on each other. That means the output of one may be used as an input for another. In RISC architectures, we expect most ISA instructions to be translated into a single μ–instruction, and hence, we assume at most B ISA instructions can co–exist in a bundle.

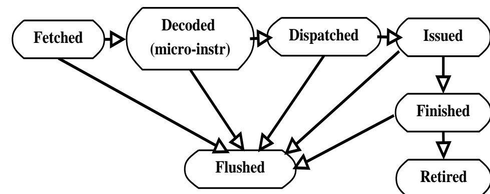

Figure 4. The state transition diagram for instruction execution.

At most one bundle can be *dispatched* in a cycle and each μ–instruction within the bundle is dispatched to a different functional unit (FU). The instruction bundles are dispatched in the program order. At most one μ–instruction is dispatched to an FU at a time, although may be several FUs of the same type. The total number of FUs may exceed the number of μ–instructions in each bundle, so some FUs may not receive new μ–instructions every cycle.

Before a μ–instruction bundle can be dispatched to the functional units, the following resources must be available for each μ–instruction in the bundle:

1. **Rename Buffer Entries**: Rename buffers are logical registers that are used to eliminate *Write-After-Read* and *Write-After-Write* dependencies.
2. **A Reorder Buffer Entry**: The reorder buffer is a queue that keeps track of the status of the dispatched bundles. Instruction bundles retire from the reorder buffer in the order they were dispatched after all of their μ–instructions have *finished*, and all the older bundles have retired.
3. **Load/Store Buffer Entries**: Load/Store buffers are used to buffer the values read by the load instructions or written by the store instructions.
4. **FUs Issue Queue Entries**: Each FU has a separate issue queue. Each μ–instruction in the bundle needs an entry in the corresponding FU's issue queue.

If any of these resources are not available, the instruction dispatch will be delayed until they become available. Typically, this only occurs when there are long latencies in the FUs so that one of the structures mentioned above becomes full.

Once a μ–instruction bundle is dispatched, each μ–instruction in it will be queued in the corresponding FU issue queue. The instruction remains in the issue queue until all the data it depends on becomes available, after which it can be *issued*. An issued μ–instruction will be processed by the FU core to produce the result. Once the result is ready, the instruction's state becomes *finished*. The FU core may reject a μ–instruction for a number of reasons, in which case the instruction will be put back in the FU issue queue and will be re–issued later. Instruction issue occurs out–of–order with respect to program order. Once the μ–instruction bundle retires

Miss Event	Effect	Description
ICache Miss	Empty Reorder Buffer	Instructions must be brought into the ICache either from L2 or memory.
Branch Misprediction	Empty Reorder Buffer	All in–flight instructions older than the mispredicted branch are flushed.
Data Cache Miss	Retirement Stops	A delay in the LSUs to finish a load or store instruction due to a data cache miss.
Address Translation Misses	Retirement Stops	A miss occurs in the hardware accessed address translation structures (e.g. TLB). The miss either delays processing a load/store instruction in the LSU, or results in the rejection of the instruction from the LSU.
Rejections	Retirement Stops	Any of the FUs rejects an instruction for any (e.g. hitting a resource limit). The instruction must be reissued after some delay or reordering.
FPU Latency	Retirement Stops	A delay in FPU to finish the computation for an issued instruction.
IU Latency	Retirement Stops	A delay in IU to finish the computation for an issued instruction.

Table 1. Miss event types with their potential effect.

(completes) all resources allocated to it, including the entries in the rename buffers, the reorder buffers, and the load store buffers, will be released. An instruction may be *flushed* for different reasons, including branch mispredictions.

A finished μ–instruction may *retire* only if , and (i) all other μ–instructions in the instruction's bundle have also finished and (ii) all older (with respect to the program order) bundles in the reorder buffer have already retired. Bundle retirement happens in program order. At most one bundle can retire per cycle. Therefore, the maximum number of ISA instructions that in theory can retire in a cycle is equal to B.

The key idea behind the SSB model is that most bottlenecks can be detected by speculatively attributing a *source* to each stall, i.e., a cycle in which no bundle from the reorder buffer can retire. There are two major categories of such stalls:

- The reorder buffer is empty. This implies that the front–end has not been able to feed the back–end in time. Assuming the micro–architecture is designed and tuned properly, such situations happen mostly when there is an ICache miss, or when a branch misprediction occurs. We assume the hardware designates the most recent event (an ICache miss, or a branch mispredication) as the source of the stall.
- The reorder buffer is not empty, but the oldest bundle in the reorder buffer cannot retire because one or more of its μ–instructions have not yet finished (i.e. they are waiting for an FU to provide the results). We assume in this case that once all μ-instructions of a bundle finish and the retirement resumes, the hardware will designate the source of the stall as the last FU that finished a μ–instruction so that the instruction retirement could resume.

We call the hardware events that can cause a stall *miss events*. The miss events we consider in this study are listed in Table 1 along with the type of stalls they cause and the potential effect they may have.

The association between a stall and a miss event is not necessarily precise because of the dependencies among instructions within the same bundle. For instance, an instruction i may depend on the output of another instruction, j, of the same bundle. In this case, stalls caused by miss events during the execution of j are charged to i because it is the last μ–instruction in the bundle to finish.

Finally, even if a stall is identified as being caused by a particular event, removing that event does not necessarily translate into an elimination of the stall. This is because of the highly concurrent nature of superscalar out–of–order microprocessors and the fact that events may overlap so that removing one of them may not regain all the performance lost because of the stall. This issue is discussed extensively in other work [9, 10, 26]. Addressing this issue in the general case is complex because in today's out–of–order processors hundreds of instructions may be in–flight simultaneously. To solve the problem in its generality it is necessary to consider all possible interactions of any subset of concurrently executing instructions, which is beyond the scope of an on–line tool.

In our SSB model, we justify ignoring both intra–bundle dependencies and miss event concurrencies by statistically comparing certain miss events captured when running workloads in three different processor modes (a) out–of–order multi–dispatch mode (*real*), (b) out–of–order single–dispatch mode (*single*) and (c) inorder single–dispatch mode (*inorder*). When running the processor in single mode, only one ISA instruction is dispatched in a μ–instruction bundle, and therefore, the instruction can independently finish whenever all its μ–instructions finish. When running the processor in inorder mode, only one ISA instruction is dispatched in a bundle, and also, the instruction is not issued until the preceding instruction bundle retires. In this mode, most overlaps between the miss events caused by separate ISA instructions are removed, because their execution is serialized. Comparing the operation of the processor when running in real mode (with full concurrency) to when the processor runs in single or in inorder mode yields interesting information. As we show in Section 6.3, the statistical difference between the three modes above is not significant for most of the applications we examined.

Based on this observation, we propose the following formula to detect bottlenecks at each phase in the program execution:

$$CPI_{Real} = \sum_{i=0}^{n} Stall_i + CPI_C$$

where, $Stall_i$ is the number of stalls caused by miss event i in the monitoring period, and CPI_C is an estimate for the CPI that can be achieved by idealizing all miss events. CPI_{Real} is easily computed by dividing the number of elapsed cycles by the number of ISA instructions retired at any period of time. We also rely on hardware PMU features to provide values for $Stall_i$. As a result, we can accurately show how much gain is potentially achievable by reducing the miss events of a certain type.

5 Experimental Environment

5.1 Hardware

We have implemented and evaluated our HPC–based performance monitoring facility on the IBM PowerPC970 processor [14], in an Apple PowerMac G5 workstation. Evaluating sampling on a real microprocessor as opposed to a cycle–accurate simulator offers two advantages. The higher speed allows us to collect considerably

more data, making the analysis more complete and accurate. Further, simulators may not reflect some of the limitations of a real microprocessor. For example, in a simulator we can easily assume that certain hardware events can be counted at no cost. But on a real microprocessor, both the number and the type of the events that can be counted simultaneously is constrained. The downside of using a real microprocessor is that because of its complexity, understanding the semantics of the hardware events requires significant internal (and potentially proprietary) knowledge of the processor implementation.

The PowerPC970 processor we used operates at 1.8GHz. It features a 64-KByte ICache and a 32-KByte L1 data cache. The level 1 caches are backed by a 512-KBtye unified L2 cache. The processor has two IUs, two FPUs, two LSUs, and one BU. The PowerPC 970 can dispatch a bundle of up to five μ-instructions (including one branch) per cycle. Each FU can independently issue one instruction per cycle. One bundle (up to five μ-instructions) can complete in a cycle (i.e. $B = 5$). Because the PowerPC architecture is RISC–based, most ISA instructions are converted to one μ–instruction. Therefore, the ideal CPI is approximately 0.2.

The PowerPC970 processor has 8 HPCs and can count a large variety of hardware events. The processor has counters that count the number of stalls caused by miss events including the ones listed in Table 1. When a stall occurs, a counter starts counting the number of cycles the CPU is in stall. The first event that causes the resumption of instruction retirement (i.e. a data item is received from the data cache), the value of the counter will be charged to this event as the number of stall cycles caused by the event. The assignment of stall to cause is speculative, since (i) several events can happen in a single cycle and the PMU chooses one of them to attribute the just–ended stall, and (ii) multiple stall causes may overlap, yet the stall length is attributed to just a single cause. Such hardware support for identifying stall sources is not specific to PowerPC970; for example, Intel Itanium II processor provides similar features [15].

5.2 Operating System

Our performance monitoring facility is implemented in the K42 operating system [12]. K42 is an open–source research operating system designed to scale well on large, cache–coherent, 64-bit multiprocessor systems. It provides compatibility with Linux API and ABI. The K42 kernel is designed in an object–oriented fashion, a feature that allows for easier prototyping.

The kernel module we built in K42 operates independently from the kernel. There are two major types of interactions with the kernel. First, the sampling engine must be able to dynamically install custom exception handlers for the PMU overflow exceptions. Secondly, the kernel must be modified to notify the sampling engine of all process creations, exits, and context switches. We did not modify K42 for this purpose. Instead, we exploit the fact that in K42, all major process management events (along with other the operating system events) are recorded in performance monitoring trace buffers. Upon each overflow exception, the sampling engine checks whether a context switch has recently occurred by consulting the trace buffer. Using this scheme, there is a delay in detecting context switches, but because the granularity of context switches is usually around 10 milliseconds, which is two orders of magnitude larger than the sampling granularity, the imprecision added by a small delay in detecting context switches is insignificant.

In order to record the gathered HPC values, we used the K42 performance monitoring infrastructure that is already in place [28]. The infrastructure provides for an efficient, unified and scalable tracing facility that allows for correctness debugging, performance debugging and on–line performance monitoring. Variable–length event records are locklessly logged on a per processor basis in the trace buffer mentioned above. The infrastructure is uniformly accessible to the operating system and user programs. The recorded events are encoded using XML, and thus, much of the implementation of adding and processing new events is automated [29]. Both HPC and SSB values gathered by the sampling engine are added to the buffers and thus available to any interested party.

6 Experimental Results

We developed and ran a number of experiments to evaluate our statistical sampling approach. In this section we describe these experiments and present their results. First, we briefly describe how we validate the basic values we read from the HPCs for different hardware events. We then present the results of our statistical analysis of sampling accuracy and show how the sampling accuracy changes as a function of sampling granularity and sampling ratio. Finally, we analyze the accuracy and usefulness of computing SSB values.

We present the results of our analysis for four integer and four floating point applications from the SPEC2000 benchmark suite[1]. The integer benchmarks are `gcc`, `gzip`, `crafty`, and `perlbmk`. The floating point benchmarks are `mgrid`, `applu`, `art`, and `mesa`.

6.1 Basic Validation

Previous work [19] has shown that validating the numbers read from HPCs is not trivial. In today's processors, HPCs are so specific to the micro–architecture implementation that their semantics cannot be completely understood without detail knowledge of the micro–architecture. This information is often proprietary and not publicly available. For instance, the significance of L1 cache misses may differ depending on the effectiveness of the prefetch structures. Also, a load instruction that incurs an L1 cache load access may be rejected by the LSU a few cycles later due to a TLB miss, or some other LSU constraints, and as a result, literally reading the number of L1 cache accesses can be misleading.

To reach a clear understanding of the semantics of the hardware events and we created a large set of micro–benchmarks, each carefully designed to a produce a specific scenario in which a specific set of hardware events can be monitored in isolation. Such an approach may not be feasible for some hardware events, because it requires comprehensive knowledge of the internal implementation of the microprocessor. However, the results of our experiments (not shown here) gave us confidence that our infrastructure correctly captures most of the hardware events we examined.

6.2 Sampling Accuracy

In order to measure the accuracy of sampling versus fully counting the hardware events, we use a statistical approach. When counting events fully, we associate with each hardware event, e, a probability distribution $P_e(R_i)$ representing the probability of event e occurring in the time interval R_i. $P_e(R_i)$ can be simply calculated by dividing the frequency of e re-occurring in the interval R_i by the total number of e events during a monitoring session. That is if N_e is the total

[1] We also examined several other applications from the same suite with similar results.

Application	gzip	gcc	perlbmk	crafty	applu	mgrid	art	mesa
Instructions Retired	0.01	0.06	0	0	0	0.05	0	0.12
L1 DCache Loads	0.09	0.04	0	0	0.02	0.07	0.03	0.22
L1 DCache Stores	0.13	0.08	0	0	0	0.03	0	0.10
L1 DCache Misses	1.21	0.05	0	0	0.02	0.07	0.07	0.05
ICache Misses	N/A	0.12	0.02	0.09	0.02	N/A	0.03	N/A
TLB Misses	N/A	0.11	N/A	N/A	0.01	0.15	N/A	N/A
ERAT Misses	0.79	0.13	0.01	0.02	0.07	0.71	0.42	0.21
L2 Cache Misses (Data)	0.24	0.01	0.07	0.02	0.02	0.06	N/A	0.17
Branch Mispredicts	0.36	0.05	0.01	0	0.02	0.18	0	0.13

Table 2. KL-distance between probability distribution P, which is obtained by fully counting hardware events and P', which is obtained by sampling. The sampling ratio is 10 and the sampling round R is 2 million cycles. The value 0 stands for any value less than 0.01. The N/A implies the event is less frequent than once every 10,000 cycles on average.

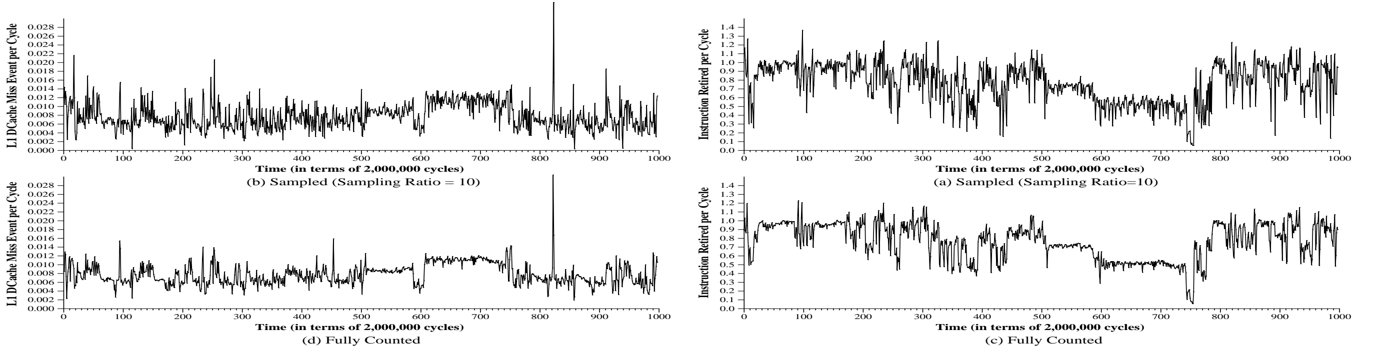

Figure 5. Comparing fully counted events with sampling when running `gcc`: Left: average instructions retired per cycle; Right: average number of L1 DCache misses per cycle.

number of occurrences of event e, and $N_e(R_i)$ is the number of occurrences of event e in interval R_i, the *probability* of event e occurring within R_i is calculated as $P_e = \frac{N_e(R_i)}{N_e}$ so that $\sum_{i=0}^{N} P_e(R_i) = 1$.

With sampling on the other hand, we count how many times e occurs in a subinterval of R_i, and linearly scale it to the entire interval, which will give us another probability distribution $P_e'(R_i)$. A key question is how the two distributions, P_e and P_e', corresponding to the actual counts and sampled counts, differ. To answer this question, we use *Kullback Leibler distance* (KL-distance), which is often used to measure similarity (or distance) between two probability distributions[5]. KL-distance is defined as:

$$K(P_e, P_e') = \sum P_e(x) \log \frac{P_e(x)}{P_e'(x)}$$

and computes the geometric mean over $P_e(x)/P_e'(x)$. The reason we use KL-distance (as opposed to, for instance, the mean over $|P_e(x) - P_e'(x)|$) is that in the context of a dynamic–optimizer the absolute values of the hardware event counts are often not a determining factor because there are many short transient states in the hardware. What is more important is whether there is a significant and stable shift in the rate of occurrences of a particular hardware event that lasts for a sufficiently long period of time to be worth considering. Therefore, although there may be sampling intervals in which the values of P_e and P_e' differ significantly, if such intervals are limited in number and isolated, they do not distort the distance measure due to the log factor in KL-distance.

In this study, we consider any value of $K(P_e, P_e')$ below 0.20 to be acceptable. Informally speaking, we consider the sampling adequate if the difference between the values of two probability distributions on average does not exceed 15%.

We measured $K(P_e, P_e')$ for a large number of hardware events for the selected SPEC2000 applications. Table 2 shows the results for several important hardware events. The 0 entries imply the actual value of $K(P_e, P_e')$ was less than 0.01. The N/A entries imply that the hardware event was on average less frequent than once per 10,000 cycles, and hence, insignificant. The samples are collected over 6-billion cycles (after skipping over the first billion instructions). The sampling interval R is 2 million cycles, and the sampling ratio is 10.

As it can be seen in Table 2, the KL-distance value is small for most hardware events in a majority of applications, with a few exceptions we discuss later in this section. In Figure 5 we depict graphically the rate of occurrences for two hardware events, instruction retired and L1 data cache miss, using the same setup as above except for the fact that only the first two billion cycles are included in the graphs. Note that we chose to show occurrence rate (i.e. the average number of occurrences per cycle) only for visualization purposes. The probability distributions P_e and P_e' can be directly derived from these graphs by dividing each point by N_e. It can be seen that the sampling distribution accurately follows all significant and steady changes in the real occurrence rate of the hardware events (even though there are differences over small periods of time).

There are a few cases in Table 2 with unacceptably high values. However, we note that these cases all correspond to fairly infrequent events (one per 100 cycles on average). Although infrequent events are unlikely to cause performance bottlenecks, we explored this issue further by varying the sampling granularity and sampling ratio for them. We ran several experiments with `gzip` for which at least three hardware events have a relatively large KL-distance: L1 data cache miss, ERAT (Effective to Real Address Table) miss, and branch misprediction. Figure 6 shows the results of the experiments. The graph on the left shows how the accuracy changes as a function of the sampling granularity. As a general rule, larger granularities have higher sampling accuracies for infrequent events.

Therefore, we change the sampling granularity from 200,000 to 500,000 cycles. We then wanted to know how sensitive the accuracy is to the sampling ratio in this sampling granularity. The graph on the right shows the results of our experiments. It appears that none of the three hardware events is highly sensitive to the sampling ratio. The general conclusion we draw from these experiments is that it is better to use larger granularities (with a fixed sampling ratio) for infrequent hardware events.

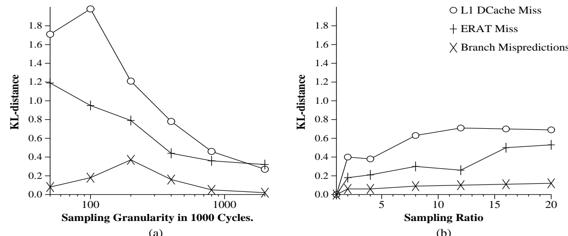

Figure 6. Tuning sampling ratio and sampling granularity for `gzip`**: (a) The KL-distance generally decreases as the sampling granularity increases. (b) Fixing the granularity to 500,000 cycles, all three hardware events seem to be fairly stable when changing the sampling ratio within a realistic range.**

6.3 Computing SSB

In Figure 7, we show the result of computing the SSB values for `gcc` over a period of 50 billion cycles. There are several observations that can be made from the graph. First, the entire run is divided into several fairly long phases in which either CPI is stable, or CPI changes in a regular fashion. In each phase, the SSB pattern is reasonably stable, so that it is possible to pinpoint one or more major sources of stalls. Secondly, there is often a large gap between the real, measured CPI and the ideal CPI, most of which can be explained by the stalls. Thirdly, in this particular example, data cache misses seem to be the single most important source of stalls. There are also a significant number of LSU rejections. An LSU rejection usually occurs if either an address translation miss occurs (which is accounted for separately), or if the LSU resources are exhausted. Therefore, many of the rejections are caused indirectly by data cache misses which keep the LSU busy with an instruction for a long time. There is also a significant number of stalls caused by ICache misses and branch mispredictions in the early phases of the run which disappear later on.

The SSB computed by our sampling engine can provide accurate and timely hints to a run–time optimizer, allowing it to focus, in this case, on techniques to reduce data cache misses for most of the program and preventing it from focusing on optimizations that might reduce the computation, branch mispredictions, or ICache misses as they will not have significant effect unless they manage to also reduce data cache misses. Also, the online availability of the SSB information allows the run–time optimizer to monitor the results of the applied optimizations, and measure their benefits and potential negative side effects in a feed-back loop.

In order to estimate the effect of intra–bundle dependencies and miss event concurrencies, we executed all of our benchmarks in three different processor modes: multi–dispatch out–of–order (real), single–dispatch out–of–order (single), and single–dispatch in-order (inorder). We used KL-distance to measure the statistical distance between the distribution of stalls for each particular source in the different execution modes. In order to make the comparison precise, we used instruction–based multiplexing, instead of time–based multiplexing. In instruction-based multiplexing, periods are calculated in terms of retirement of a certain number of ISA instructions. Therefore, all sampling points in the three execution modes are aligned to each other in terms of the number of instructions retired between the two sampling points. Of course, both the single and inorder modes take a significantly longer time to finish than the real mode. Also, by monitoring user–level code only, we make sure the exact same stream of instructions are executed in the three runs.

Table 3 shows the results of our analysis. KL_S is the the KL-distance between the real mode and the single mode, and KL_I is the KL-distance between real mode and inorder mode runs. Again, 0 implies any value less than 0.01, and N/A implies the average occurrence rate of the particular stall is less than once per 10,000 ISA instructions, and hence not significant for bottleneck analysis.

For the great majority of entries, both the KL_S and KL_I values are very small, suggesting that, in most cases, the distortions due to intra-bundle dependencies and overlapping miss events can be ignored in quantifying a particular source of stall. There are a few exceptions, most pronounced in `mgrid`. A closer investigation of `mgrid` shows that its stalls are dominated by data cache misses and the FPU latencies for both KL_I and KL_S is reasonably small. Other sources of stall, such as branch misprediction or rejections are in fact insignificant. We are currently exploring these anomalies further.

Sampling Frequency (kHz)	1	2	5	20	100	200	1000
Runtime Overhead (%)	0.2	0.5	0.8	2	12	22	63

Table 4. The runtime overhead of computing and logging SSB.

Table 4 shows the run–time overhead of continuously gathering the SSB values and recording them into the trace log. We measured the overhead for different sampling frequencies. The overhead increases linearly with the sampling frequency within the range we examined. At 20,000 samples per second, the run–time overhead is around 2%. We believe that with such a low run–time overhead our sampling engine is suitable for run–time optimization purposes.

7 Related Work

Related work closest to our work is DCPI which uses fine–grained sampling of the HPCs to identify system–wide hot spots at run–time [3]. It also attempts to identify pipeline stalls at the instruction level. There are some hints that event multiplexing is implemented in this system, but no details of the design nor statistical analysis is provided. Moreover, we argue that with the increase in the number of concurrent in–flight instructions, the imprecision of a software–only approach to attribute stalls to the instructions can be high.

PAPI [7] is a public domain tool that is implemented on many platforms. Its main emphasis is on platform–independence rather than efficiency. The portable interface is implemented in software, and as a result, it may incur significant overhead in some scenarios. PAPI also implements HPC multiplexing at user level [20]. A fine–grained timer is used as a means for HPC group switch. The timer will send a signal to the process that has requested a multiplexed set of hardware events. A major limitation of this approach is that due to the large overhead of HPC group switch (the cost of signal delivery plus the cost of kernel/user context switches), the sampling granularity must be large and as a result, the sampling error may become high for some applications. Another problem with switching HPC groups in user space is that there is a potentially large latency between the time when the timer expires and the time

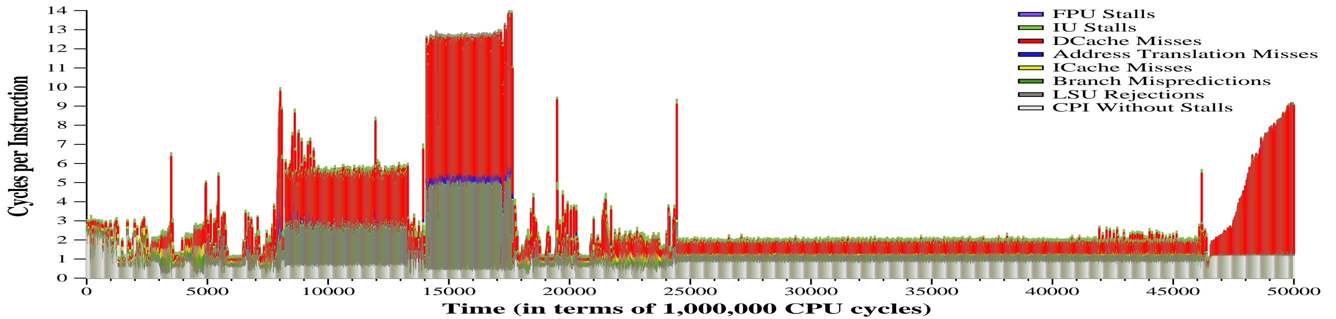

Figure 7. Statistical Stall Breakdown for an instance of `gcc` run for a 50 billion cycle period.

Application	gzip		gcc		perlbmk		crafty		applu		mgrid		art		mesa	
Stall Source	KL_S	KL_I	KL_S	KL_I	KL_S	KL_I	KL_S	KL_I	KL_S	KL_I	KL_S	KL_I	KL_S	KL_I	KL_S	KL_I
ICache Miss	0.01	0.01	0.05	0.05	0	0	0.01	0.01	0.06	0.06	N/A	N/A	0.28	0.19	0.15	0.15
DCache Miss	0.02	0.02	0.05	0.06	0.03	0.05	0	0.01	0.08	0.08	0.12	0.42	0.03	0.03	0.03	0.05
ERAT Miss	0.36	0.36	0.28	0.30	N/A	N/A	0.03	0.13	0.13	0.13	N/A	N/A	N/A	N/A	N/A	N/A
FPU Latency	N/A	N/A	N/A	N/A	0.01	0.02	0.13	0.13	0	0	0.25	0.25	0	0	0.01	0.02
Int. Latnecy	0.10	0.10	0.15	0.14	0	0	0	0	N/A	N/A	N/A	N/A	0.05	0.05	0.06	0.06
Misprediction	0.02	0.02	0.06	0.06	0	0.01	0.01	0.01	0.06	0.06	1.68	1.68	0.28	0.28	0.15	0.15
Rejection	0.47	0.22	0.17	0.21	0.13	0.14	0.03	0.03	0.15	0.15	0.40	1.47	0.02	0.02	0.50	0.50

Table 3. KL-distance between the stall distributions. KL_S is the KL-distance between the stall distribution when executing in real mode versus the stall distribution when executing in single dispatch mode. KL_I is the KL-distance between the stall distribution when executing in real mode and the stall distribution when executing in inorder mode.

when the signal is actually delivered and the signal handler (where the current HPC group is read and stored) is called. This delay adds to the imprecision of multiplexing. Finally, to the best of our knowledge, there is no quantitative study on the overhead and accuracy of PAPI's multiplexing engine. In fact, one can build PAPI's high–level platform–independent interface transparently on top of our low–level and efficient multiplexing scheme.

ProfileMe uses *instruction sampling*, where individual instructions are selected randomly to be monitored by the processor hardware when the instructions pass through the different stages of the system pipeline [6]. Their main goal is to gather accurate information on instructions that cause miss events or have long latencies. Although the instruction sampling can be effective, there is little analysis in the paper that shows the actual run–time overhead of building an instruction–level profile. We believe our approach can be complemented by approaches such as ProfileMe to search for bottleneck in a multi–level fashion.

Intel's VTune [16] is one of the most widely used tools to make the facilities of the PMU available to developers. It provides both sampling and binary instrumentation facilities, and outputs a graphical display of the programs hot spots as well as call graph. There are several other tools built for various hardware platforms with similar sets of features such as Apple's CHUD [4] and PCL [22]. They provide facilities to identify program hot spots and the frequency of important hardware events such as cache misses or branch mispredictions. To the best of our knowledge, none of these tools allows for profiling more events than the number of HPCs at the same time. Also, they often only expose the hardware PMU features directly to the user. It is up to the user to interpret the semantics of the low–level hardware events.

Wassermann et. al [27] presented an analysis of microprocessor performance where a model similar to SSB is used to characterize the effect of stalls caused by cache and memory latencies. Estimating the number of stalls caused by a source is done in software by multiplying the number of accesses to the source by its average access latency. Our approach extends this effort in two directions. First, we exploit hardware support to measure the stalls more accurately. Secondly, their approach mainly focused only on cache and memory stalls, while we include all possible sources of stall into our analysis.

SMARTS [30] and SimPoint [23] use sampling in a different context as a means to accelerate detailed micro–architecture simulations. In both approaches, the detailed simulation of the program is done only for small intervals during the program execution. There are strong evidences in both work that show their sampling techniques would result in accurate estimates of important hardware metrics such as CPI.

Studies have been done to model the behavior of superscalar microprocessors either through analytical modeling [17] or through statistical simulation [21]. In both cases, a simple and aggregate model is constructed that can provide accurate and high–level insight of hardware bottlenecks. These models are simple and efficient, even though, they often make unrealistic simplifying assumptions. Our SSB model was inspired by some of this work.

Slack [9] and Interaction Costs [10] are two models for accurately estimating how much performance gain can be achieved by idealizing latencies of individual instructions. Although these approaches provide accurate information about potential gain of idealizing individual instructions, they require additional hardware support and extensive postmortem analysis which make them difficult to use in the context of run–time optimization.

Lemieux [18] has extensively explored issues in designing the interface and also the implementation of the PMU of microprocessors with an emphasis shared memory multiprocessor systems. The main focus of the work is also to account for stalls (memory and non-memory) that that attribute to significant performance penalty.

8 Concluding Remarks

Hardware performance counters (HPCs) are useful for analyzing and understanding performance, but there are challenges in using them on line. There are a small number of HPCs available in most today's microprocessors. Moreover, the definitions of the hardware events that can be counted by HPCs are low–level and complex.

In this paper, we described two techniques that overcome the the limitations we identified of microprocessor HPCs. First, we provide larger set of logical HPCs by dynamically multiplexing HPCs using statistical sampling of the hardware events. Using real programs, we showed experimentally that counts of hardware events obtained through sampling is statistically similar (i.e. within 15%) to real counts of the events. Secondly, we proposed a technique that speculatively associates each stall cycle to a processor component that likely caused the stall, and built this technique using our HPC multiplexing engine. Our experiments showed that identification of the stalls is reasonably accurate for most of the applications we examined. The run-time overhead of our sampling engine is under 2% allowing it to be used online.

The facility we have implemented is useful for detailed on–line performance analysis of application and system code running at full speed with small overhead. It is also effective in reporting hardware bottlenecks to tools such as a dynamic optimizer that might guide dynamic adaptation actions in a running system. A number of outside groups have started using our statistical sampling tool. Our research group is interested in using the tool to help guide dynamic optimizations within the operating system. Much of this work has just started, and it is an open question how difficult it will be to map hardware behavior to the responsible software component.

Our work and other previous work has identified the challenges of correctly interpreting HPC values. Counter descriptions are proprietary, documented poorly, or designed for hardware architects. Our techniques and tools for extending the number of logical HPCs available and for providing a easy to understand characterization of sub-optimal processor performance alleviate some of the difficulties faced with HPCs.

9 References

[1] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proc. of Conference on Supercomputing*, Baltimore, Maryland, Nov. 2002.

[2] AMD. Athlon Processor X86 code optimization guide. *http://www.amd.com/us-en/assets/ content_type/white_papers_and_tech_docs/22007.pdf*.

[3] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandervoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. of the 16th ACM Symposium of Operating Systems Principles (SOSP)*, pages 1–14, Oct. 1997.

[4] Apple Computer Inc. Computer Hardware Understanding Development (CHUD) tools. *http://developer.apple.com/tools/performance/*.

[5] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., 2003.

[6] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. of the 30th Intl. Symposium on Microarchitecture (MICRO)*, pages 292–302, Dec. 1997.

[7] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *Proc. of Workshop Parallel and Distributed Systems: Testing and Debugging (PATDAT), joint with the 19th Intl. Parallel and Distributed Processing Symposium (IPDPS)*, Niece, France, Apr. 2003.

[8] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proc. of 12th Intl. Conference on Parallel Architecture and Compiliation Techniques (PACT)*, Dec. 2003.

[9] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Slack: Maximizing performance under technological constraints. In *Proc. of the 30th Intl. Symposium on Computer Architecture (ISCA)*, San Diego, CA, June 2003.

[10] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proc. of the 36th Intl. Symposium on Microarchitecture (MICRO)*, San Diego, CA, Dec. 2003.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA, 2003.

[12] IBM Corporation. K42 research Operating System. *http://www.research.ibm.com/k42*.

[13] IBM Corporation. The POWER4 processor introduction and tuning guide. *http://www.redbooks.ibm.com/pubs/pdfs/redbooks /sg247041.pdf*.

[14] IBM Corporation. PowerPC970: First in a new family of 64-bit high performance PowerPC processors. *http://www-3.ibm.com/chips/techlib/ techlib.nsf/producs/PowerPC_970_and_970FX_Microprocessors*.

[15] Intel Corporation. Intel Itanium 2 reference manual for software development and optimization. *http://www.intel.com/design/itanium2/ manuals/251110.htm*.

[16] Intel Corporation. VTune performance analyzers. *http://www.intel.com/software/products/vtune*.

[17] T. Karkhanis and J. E. Smith. Modeling superscalar processors. In *Proc. of the 31th Intl. Symposium on Computer Architecture (ISCA)*, Munchen, Germany, June 2004.

[18] G. Lemieux. Hardware performance monitoring in multiprocessors. Master's thesis, University of Toronto, 1996.

[19] M. Maxwell, P. Teller, L. Salayandia, and S. Moore. Accuracy of performance monitoring hardware. In *Proc. of the Los Alamos Computer Science Institute Symposium (LACSI)*, Santa Fe, NM, Oct. 2002.

[20] J. M. May. MPX: Software for multiplexing hardware performance counters in multithreaded systems. In *Proc. of the Intl. Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, Apr. 2001.

[21] S. Nassbaum and J. E. Smith. Modeling superscalar processor via statistical simulation. In *Proc. of the 10th Intl. Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 15–24, Sep. 2001.

[22] PCL. The Performance Counter Library: A common interface to access hardware performance counters on microprocessors. *http://www.fz-juelich.de/zam/PCL/ doc/pcl/pcl.html*.

[23] T. Sherwood, E. Perlman, G. Hamerly, and B. Caler. Automatically characterizing large scale program behavoir. In *Proc. of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002.

[24] B. Sprunt. Pentium 4 performance monitoring features. *IEEE Micro*, 22(4):72–82, July/Aug. 2002.

[25] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proc of 3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[26] E. Tune, D. M. Tullsen, and B. Calder. Quantifying instruction criticality. In *Proc. of the 11th Intl. Conference on Parallel Architecture and Compilation Techniques (PACT)*, Charlottesvill, VA, Sep. 2002.

[27] H. J. Wassermann, O. M. Lubeck, Y. Luo, and F. Bassetti. Performance evaluation of the SGI Origin2000: a memory-centric characterization of lanl asci applications. In *Proc. of ACM/IEEE Conference on Supercomputing (SC)*, San Jose, CA, Nov. 1997.

[28] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proc. of the Supercomputing Conference (SC)*, Phoenix, AZ, Nov. 2003.

[29] R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Cascaval, and R. Azimi. Performance and environment monitoring for whole system characterization and optimization. In *Proc. of the 2nd IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (PAC)*, Yorktown Heights, NY, Oct. 2004.

[30] R. E. Wunderlich, T. F. Wenish, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. of the 30th Intl. Symposium on Computer Architecture (ISCA)*, San Diego, CA, June 2003.