Hardware Performance Monitoring For the Rest of Us
A Position and Survey

Tipp Moseley[1,2*], Neil Vachharajani[2,3*], and William Jalby[1]

[1] Université de Versailles Saint-Quentin-en-Yvelines
[2] Google Inc.
[3] Pure Storage Inc.
* Current affiliation.

1 Introduction

Microprocessors continue to make great strides in performance and scalability, yet hardware performance monitoring remains an area of dissatisfaction amongst those interested in better understanding the interactions of hardware and software. HPM technology has, at best, maintained the status quo for over a decade, though hope for better answers still remains. As it is, HPM is well-suited for some purposes, and everyone else tries to make the most of what is available. HPM will never be everything to everyone, and as new features are added, new users will adopt them in unforeseen ways.

HPM means different things to different people, so before proceeding we will give it a definition. Hardware performance monitoring refers to any hardware mechanism that enables (not necessarily by design) insight into how software performs on a microprocessor. This definition includes features as simple as timer-based interrupts, but also a broad range of things like event counters, last branch buffers, instruction-based samples, and many more.

Defining perspective is equally important. Performance tuning for any party involves identifying hot and representative code and determining whether the code is satisfactory relative to some notion of peak performance. When code does not achieve peak performance, the perspective of a hardware architect and software developer are quite different. From a hardware architect's perspective, some hardware unit performs poorly due to an instruction which cannot be changed. From a software developer's perspective, some instruction performs poorly due to some hardware unit which cannot be changed. Understanding this distinction is important for deciding how information is to be collected and presented. We describe HPM from a software perspective in hopes of influencing hardware architects to consider this viewpoint in future designs.

When tuning an application, either manually or automatically, there comes a point when further performance gains can only be achieved by truly understanding the minute details of the microarchitecture. Due to the effort required, however, this is one of the last stages of application tuning. Figure 1 illustrates this phenomenon.[4] Of course, some strategies can fit into multiple regions. Conveniently, the graph has no scale. The curve is segmented into three stages:

1) Application Insight. The biggest performance gains come from identifying and understanding the behavior of the hot portions of the code to apply better algorithms, refactor code, or choose the best data structures [1].

Simply knowing where an application spends its time is the most valuable piece of information available. Yet this seemingly simple task is still an area of active research, even for natively compiled programs [2, 3]. In the managed world, the state of things is comical (though here, HPM is not at fault) – one of our colleagues tested several different Java profiling tools and found the relative execution time attributed to specific functions can differ by an order of magnitude!

Software engineering can expedite development, but opacity can lead to terrible performance for even small programs. The authors of a binary decision diagram (BDD) based potential parallelism detection tool discovered a 12X speedup simply by tuning parameters of their BDD

[4] This figure comes completely from the authors' imaginations.

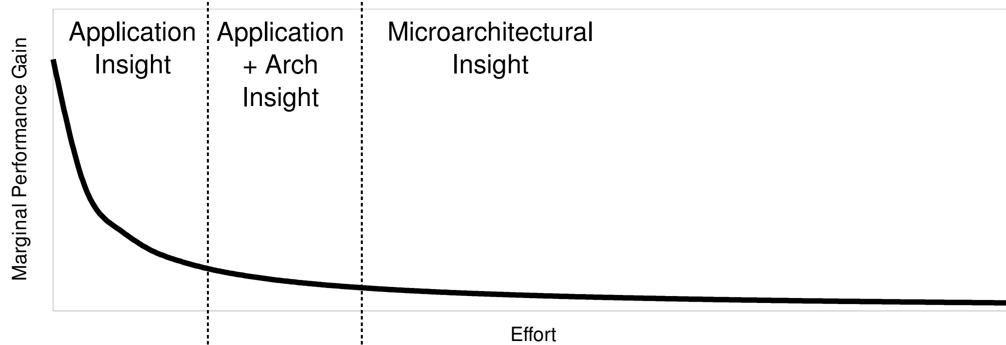

Fig. 1. Additional tuning effort versus marginal performance improvement.

library's garbage collector [4]. For linear solvers, choosing the correct software package can be of similar importance. A performance engineer at a large company helped application developers achieve a 1000X speedup by integrating a library call that used fork/exec repeatedly. None of these programmers are stupid; they are simply ignorant to the entire behavior of their program.

Though we somewhat disagree with his conclusion, Zilles's "Benchmark Health Considered Harmful" [5] is a must-read for compiler and architecture researchers. The `health` benchmark is often used to tout significant gains from compiler and architectural optimizations targeting linked lists. Instead, Zilles modifies the program's algorithms and data structures to produce the same result with 200X better performance. Zilles concludes `health` probably does not represent any real benchmark, while we fear it may be more common than anyone would like to admit.

The purpose of this group of anecdotes is not to argue that HPM does not matter. To the contrary, HPM can provide useful information to help uncover these types of *hardware independent* performance sinks and assist the developer in fixing them.

2) Application + Architectural Insight. Here, we make a distinction between architecture and microarchitecture. We use the term architecture to describe any aspect of performance that requires some knowledge of how modern processors are designed (i.e., with branch predictors, TLBs, and caches), and microarchitectural for things that pertain to a specific processor implementation (e.g., Core i7's unaligned vector operations will perform better when given aligned data, but Core 2's vector operations perform the same in either case).

Since all high performance CPUs have caches, it is important to structure data access patterns with their behavior in mind. For example, on a Core 2, loop interchange for matrix multiplication provides a 10X performance improvement.

Though the complexity varies greatly, branch predictors are also a ubiquitous piece of hardware and can be a critical performance bottleneck. Moseley et al. show an example from `445.gobmk` where a clever software transformation to reduce branch mispredicts leads to an 8X performance improvement on a Pentium 4 [6].

The measurable impact of each of the previous anecdotes could vary in magnitude, but they represent common pitfalls that are significant on any piece of modern hardware. As such, HPM should provide simple and accessible features to access architectural performance information.

3) Microarchitectural Insight. HPM can be of assistance at every point on the curve, but without a simulator, it becomes *necessary* for microarchitectural insight. At this level, very subtle and often undocumented features such as queue sizes, specific latencies, alignment issues, μops, and coherency protocol idiosyncrasies come into play. Mytkowicz shows that simply altering a program's initial stack alignment by modifying the size of an environment variable can often affect performance by around 30%, and in one case, 300% [7].

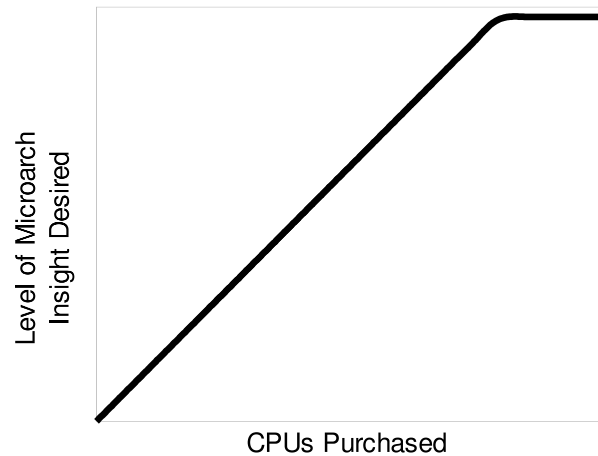

Fig. 2. CPUs purchased versus level of microarchitectural insight desired.

Because their scope is typically limited, microarchitectural optimizations are often automated by compilers, libraries, and runtime systems. Building these systems is left to the top experts, and they must have intimate knowledge of processor-specific features, techniques for measuring such features, and limitations of measurement.

Understanding microarchitectural interactions comes at a high cost of both education and tedious analysis, yet the potential gains are diminished. An increasingly popular solution is to simply guess a lot and choose the best solution [8–11], though the guessing process (also known as search or learning) can take a long time, and defining the parameters to use may still require an understanding of the underlying microarchitecture.

Interestingly, the three-stage model can be observed in compilers and operating systems as well. The first steps of the compiler are transformations like inlining, constant propagation, and common subexpression removal. Next are vectorization and loop unrolling, and finally instruction alignment, scheduling, and peephole optimizations are performed.

We believe that HPM capabilities should mirror the three-stage model. Today, low-level information, while necessarily concrete, does not cover everything compiler, OS, and VM designers care about. Stages 1 and 2 are even worse off because an expert-level of understanding is often required to collect and apply HPM data; hardware and software must improve to provide more abstract information for the earlier stages of performance tuning.

Motivation. The common desktop computing experience is enough to see that most programs do not get past stage 1. From that perspective, it is hard to motivate designing HPM with application tuning in mind. As it is, HPM is clearly designed to give hardware designers the feedback they need to understand their own decisions. However, Figure 2 highlights an important trend in hardware sales[5]; end customers who purchases more hardware have higher expectations of how well they will be able to understand and monitor the hardware. Note that the curve levels off at no more knowledge about the processor can be gained (though customers will still continue to buy more hardware after that point). Furthermore, a [real] study by HP [12] concluded that the performance increase from using counters is more effective than simply waiting for a faster processor.

For instance, when AMD's Lightweight Profiling [13] is released, we believe it alone will drive some percentage of our colleagues cluster purchasing decisions, if only for research potential. For supercomputing and datacenter customers, even the smallest gains can translate to millions of dollars of savings on energy costs. HPM that enables developers to realize these savings adds real value.

[5] This figure is also imaginary.

The rest of this paper is organized as follows. Section 2 gives an overview of modern HPM hardware. Section 3 discusses issues facing current and future designers and users. Section 4 proposes some alternatives and opportunities for improvement. Section 5 proposes a paradigm shift from the current course. Finally, Section 6 concludes.

2 Background

This section gives a brief overview of the types of hardware that exist today (or have existed in the past). It is far from complete, but it covers most of what is currently available.

2.1 Event-based Sampling (EBS)

Event-based sampling involves configuring a counter to interrupt every Nth time an event occurs. When the interrupt occurs, the program counter (PC) can be recorded. Ideally, EBS can be used to identify instructions that most frequently cause a specific event. However, due to the out-of-order (OoO) nature of modern processors, the reported PC may be quite far (tens of instructions) from the instruction that actually triggered the interrupt.

Time-based Sampling (TBS) Logically, time-based sampling is event-based sampling where the event is time. It may use a high-frequency interrupt (or an event like CPU_CLK_UNHALTED) to collect information at specific time intervals. TBS can be used to get an estimate of where time is spent in the system [14, 15], though more weight will be given to long-latency instructions. In conjunction with other counters, it can be used to get an overall sense of bottlenecks in a program [16].

TBS is the most basic form of HPM and needs no special hardware support (for low fidelity).

Precise Event Based Sampling (PEBS) Several generations of Intel x86 machines have included a feature called Precise Event-Based Sampling [17]. For a small subset of possible events (and until Core i7, only a subset of counters), PEBS will report the entire architectural state (i.e., register contents), and the PC is claimed to be within one instruction of the actual culprit.

POWER processors have a mode called random sampling which is similar to PEBS, but it gives the actual PC associated with an event.

2.2 Instruction Based Sampling (IBS)

Due to the imprecise nature of EBS, Digital introduced ProfileMe [18] in the Alpha 21264, which has since inspired Instruction-based Sampling included on AMD's x86 offerings since 2007 [19]. IBS works by tagging random instructions and recording useful properties of their journey through the pipeline. Unlike ProfileMe, which did not have to deal with μops, IBS has two modes: fetch and op sampling. Fetch sampling reports the instruction's address, whether a fetch was completed or aborted and how long it took to complete, and flags for which levels of cache or TLB were missed. Op sampling reports the instruction's address, tag-to-retire cycles, completion-to-retire cycles, and flags for branch misprediction and many memory-system related flags.

2.3 Event Address Registers (EAR)

Debuting in Intel Itanium Processors, event address registers [20] can be configured to sample detailed information including precise PC and linear address for fetches or loads (but not stores) that miss in specified caches or TLBs.

Load Latency Filtering (LLF) Load latency filtering [17] is a mechanism introduced with Intel Core i7 processors and works in conjunction with PEBS. LLF is similar to EARs, though it is more flexible and in some aspects more informative. It works by randomly tagging load (but not store) ops, and when a load exceeds a specified threshold, it reports the linear address along with fields denoting where the data came from (e.g., local L2/L3, remote cache, local/remote DRAM). LLF is more powerful than EARs because it provides more details on the source of the data. Since it is a PEBS facility, the PC may be off by up to one instruction.

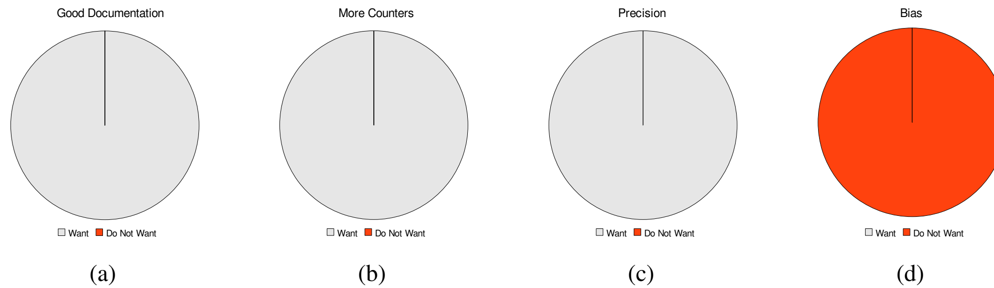

Fig. 3. Survey of HPM users (sample size = 3).

2.4 Last Branch Record (LBR)

A last-branch record is a buffer containing the most recently seen branch instructions and targets (ranging from 4 to 16 branches) [17]. It can also be configured to filter based on branch type (e.g., call, return, conditional). In conjunction with PEBS, it can be used to give the path leading up to a cache miss.

2.5 Lightweight Profiling (LWP)

Though not yet available at the time of writing, AMD has released an official specification for Lightweight Profiling [13], which represents a refreshing and radical departure from currently available hardware. LWP is configurable entirely from userspace and can be configured to generate interrupts or simply be polled. Instead of interrupting every time an an event threshold is reached, LWP uses a ring buffer so events can be processed in batch. An event record is precise; the record contains the PC responsible for generating an event. Each type of event has a unique record format (e.g., branch events have information about predictions and directions).

LWP can currently be configured to count instructions retired, branches retired, dcache misses, CPU clocks unhalted, and CPU references clocks unhalted – all simultaneously, if desired. The main limitation at this point for LWP appears to be coverage. Even events as important as TLB misses, icache misses, and FPU operations are not yet monitored.

3 Issues Facing HPM

HPM has long been an issue of contention between users and hardware designers, and all of the issues are well known (if not well quantified). In addition to discussions with our colleagues, we found the "Hardware Performance Monitor Design and Functionality" workshop [21] associated with HPCA in 2005 as well as a recent Linux kernel thread [22] to be excellent sources of information. One of the most unsettling trends we notice is that the HPM solutions have remained relatively unchanged for over a decade, yet machines have grown significantly in complexity. Since none of the insights here are new, we will instead try to motivate them with examples.

Inherent Complexity. Modern processors are inherently complex, and many interacting factors can significantly impact performance. We previously described how stack alignment can affect performance by up to 300% [7], but instruction alignment can be a significant factor as well:

- Across weekly revisions of a compiler, the variance in performance for 176.gcc on Core 2 is much higher than for Pentium 4, largely due to an increased importance in instruction alignment.
- Aligning a loop such that it will be used by the loop-stream detector (LSD) can result in 2X performance improvement.
- Branch prediction fetches are larger than instruction fetches, so in some cases a bigger alignment boundary for loops with multiple branches can reduce branch prediction bubbles for a 50% speedup on Core 2.

– The data prefetcher can be confused by IP aliasing, resulting in 8% performance drop on 189.lucas.
– Branch aliasing can result in 5-10% performance differences across an application, and 40% for a small kernel.

With subtle issues like these, analysis can take days for each one, and some can only be uncovered using a simulator. Considering that these examples only cover alignment issues in the front-end, the number of ways a program can encounter performance corner cases is frightening.

Examples like this are uncommon except when the working set is small because a few long latency memory operations usually overshadow these issues. Thanks to this, average and even advanced users rarely deal with them. In fact, most users probably do not know they exist. Even we were surprised to learn the magnitude of some of the anomalies. Because of the knowledge required for a relatively small speedup, even if HPM did make understanding these types of issues easier, it is unlikely users would spend more time focusing on them. It is still the more common issues that matter, and HPM should cater to optimizations in stages 1 and 2 first and foremost.

Precision. From an manual performance tuning standpoint, there are two critical pieces of information: the location of problem, and what it is. **Knowing the location of a problem is more important than knowing what the problem is.** IBS is nice. PEBS is not. Knowing the location of a long-latency instruction is usually enough to reason about the cause. However, simply knowing the cause of performance problems gives the programmer no indication of where to look. Think of it this way: if you manage a datacenter with thousands of nodes and dozens of applications, would you rather know the instruction that spends the most time stalled, or that cache misses are a problem for an application?

Just the name Precise Event-based Sampling (PEBS) is surreptitiously misleading. Under most conditions, a PEBS sample gives you precisely the next instruction. On variable-length ISAs, it is non-trivial just to figure out where the previous instruction actually starts. What if the previous instruction was a branch? Prior to load-latency filtering, PEBS was less trustworthy for non-scientific applications for address profiling because for instructions like `mov eax, [eax]`, the source operand is clobbered and its memory contents are given in the PEBS buffer. With the advent of the load latency record, the correct linear address is always provided. Furthermore, PEBS sampling itself can be asymmetric during bursts of retirement events – the distribution of events is not uniform, so for frequently occurring events, some instructions can receive orders of magnitude more samples than others. Furthermore, under rare conditions, PEBS is not always precise. We have learned, through a paper review of all places, that there is an undocumented mechanism to know when PEBS will be precise. The potential for nonuniform distribution and imprecision makes PEBS quite difficult to trust. Figure 3c shows users are extremely dissatisfied with imprecision.

ProfileMe was developed for Alpha to address this problem, and in our opinion it remains the most elegant solution. It may not always be helpful in solving extremely complex microarchitectural performance problems, but it addresses the simple ones in a natural way.

In many cases, the processor is a black box, and performance tuning is like playing a game of Clue; there is a room with a dead body and 10 people who could be guilty. Imprecise profiling tells us the murder weapon but never who committed the crime. Precise profiling will always give us the perpetrator and many times (but not all) the murder weapon as well. Programmers need to know the culprit, but hardware designers are much more concerned with knowing the weapon so they can devise ways to make it less potent. For programmers, simply knowing where stalls are occurring should be easy (but is not on all hardware). Knowing that, programmers can intelligently (or blindly [9]) tweak parameters until it works.

Bias. One of the trickiest issues with event-based sampling is bias, which can come in many forms. Bias is most deceptive because to even measure it we need multiple ways to measure the same data (or be able to predict it), and that is often not an option. Suppose we want to use HPM to estimate the relative frequency of execution for each basic block. One approach might be to use EBS to count INST_RETIRED, but this approach does poorly for many reasons. Sampling

skid means the interrupt will report a PC that occurs some variable number of instructions after the instruction that triggered it. *Synchronization* effects with constant sampling periods (even when prime relative to a loop body!) can result in sample counts 3X different from those expected, and random sampling still yields sample counts with up to 50% from what they should be. Instructions that trigger long-latency stalls are subject to over counting via *aggregation effect* because they sit at the head of the instruction queue for more cycles. Consequently, the instructions in the *shadow* of a stall are under sampled. Note that these effects are common to all OoO machines.

PEBS hardware on Intel chips is an enticing alternative, but in some cases it actually has worse bias. PEBS does not support randomization of sampling intervals, so it is more susceptible to synchronization effects, and, as we previously mentioned, PEBS is not always precise and PEBS samples are not evenly distributed.

Another appealing approach is IBS, but it, too, is insufficient. Using op-sampling mode would require the user to know how many μops each instruction decodes to, and fetch-mode is biased to wrong-path instructions.

Of course, the best approach would be to use the last-branch record, but the omission of an event for taken branches retired on Core i7 prohibits this. The LBR does generally work well on Core 2, where the event exists, but there are still anomalies where some branches get significantly oversampled.

Figure 3d shows surveyed users find bias not to be a desirable feature of HPM.

Number of Events and Counters. When we developed a driver for the Pentium 4 performance counters, we cursed its arcane restrictions. We beg for forgiveness. Having 18 counter registers was great for accelerating the measurement process and allowed much better correlation of time-based samples since there was no need for interpolation. Exacerbating the problem, the number of events is growing faster than the number of counters. Two counters on Core 2 and four counters on Core i7 simply is not enough, and Figure 3b shows that 100% of users agree. However, 100% of designers say adding more counters is very challenging, so we may be at a stalemate. Perhaps a distributed approach like the Pentium 4 and proposed by others [23] is worth revisiting.

The increasing number of events can be viewed as a good thing, but it steepens the learning curve and often means multiple complex events must be counted to obtain one aggregate, more meaningful value. With so many hundreds of events, the user needs a good idea of what the problem is a priori. An oft-heard quote from internal performance tuners on more complex problems is, “we used a simulator to find the problem, but we now know could have used counter X”. Machine learning is becoming increasingly popular to deal with the abundance of information available [24, 16].

Common advice is that multiplexing the counters across time will yield statistically valid results. In the absence of more counters, hardware support for multiplexing would be extremely helpful. Yet, as we have found, even a well-chosen constant sampling period results in strong bias for a single event. **There is a disturbing lack of research describing best practices for sampling and multiplexing.**

Design Philosophy. Users have long suspected that HPM features are at best a second-class citizen of processor design. Events are what is convenient to connect to a wire, rather than what is useful to understand the program. It is hard to argue, given that the designer of the Pentium Pro and Pentium 4 counters endorses this claim as well [25], although others disagree [12]. IBM’s POWER4 PMU was allegedly “verified” by an intern, though its importance has increased dramatically in subsequent revisions. Either way, the fact remains that HPM will never satisfy all desires.

Perhaps it is the fault of those who are dissatisfied that are to blame; the dearth and difficulty still associated with software drivers indicates there is not a huge demand for advanced features. When users do not fully utilize current hardware, the case for return on investment may not yet be strong enough to dedicate the resources required for more robust HPM.

Verification. Also relating to design philosophy, verification of performance counters seems to be an afterthought. Even on Intel’s newest x86 processor, the Core i7, most of the events

relating to the L1 data cache are incorrect. There is an event on Core i7 which over counts by a factor of 3, and another on POWER5 which over counts by a factor of 4! A previous version of VMWare's replay system attempted to rely on a counter for branches retired to schedule when to intervene in native execution, but the counter was so inconsistent across processor revisions the approach was abandoned. If the standard ISA features had defects like those found in HPM, processors would be unusable. Perhaps this is the reason more features are not architected. As it is, revealing unverified features or broken features without proper errata is worse than not revealing the features at all. Verification can be costly, so it should be focused on the most worthwhile events.

Features and Use Cases. Despite much research in the area, the two most widely used applications of HPM continue to be architectural characterization [26, 27] and application performance tuning. This is likely because the hardware best supports these applications. In fact, with respect to these two applications, we have no *fundamental* complaints about the assortment of hardware currently available, though no single processor has everything we would like and the list of secondary complaints is long.

Even with current hardware, research applications for hardware introspection are plentiful; researchers have used HPM for profile-guided optimization [24], dynamic optimization [28], adaptive power management [29], thread scheduling for shared resources [30], path profiling [31], user-aware design [32], and others.

Researchers propose going much further. Adaptive thread coscheduling would benefit from duplicated performance counters for SMT contexts (but so would everyone else). Coscheduling could also benefit from per-thread utilization metrics for each shared resource. Others call for cache-line monitors to measure locality and contention in caches, buses, and NUMA systems [33–35], using bits for memory state checking [36], using branch predictor history for path profiles [37]. Eyerman et al. propose a fundamentally different HPM architecture to collect more meaningful CPI stacks for OoO machines. Shotgun profiling [38] proposes an approach to measure the interaction cost of parallel instructions, though with the abundance of cores now available, better core-to-core communication mechanisms would probably be sufficient.

From a cynical standpoint, an academic motivation for HPM is to enable writing more papers, even if there is little rationale for the proposal to actually be implemented in hardware. Some have even proposed a specialized coprocessor to analyze ProfileMe-like data [39]. We wonder how the coprocessor's performance will be modeled. With so many divergent proposals that all look compelling in research papers, we pose this question: how are architects to decide which direction to pursue?

Parallel Programs. With respect to HPM, we do not believe parallel programs have any fundamental disadvantage relative to sequential programs. The biggest factor in parallel performance is sequential performance, but some events begin to be bigger factors. For example, cache misses limit the performance of sequential programs, but their effects are amplified for parallel programs when shared caches, buses, and interconnects become contended. The same is true for any shared resource and becomes increasingly important in SMT processors.

Issues specific to parallel programs include bus contention, false sharing, true sharing, and lock contention. With features like IBS, precise attribution of cache misses is helpful, but the real problem can be difficult to identify without value profiling. Here, we would expect load latency filtering to be quite powerful; for a cache miss, it reports the linear address accessed along with where the data came, which can be used to distinguish between true and false sharing and identify other instructions that are competing for the same cache line. Interestingly, an Intel performance expert informed us that regular PEBS sampling configured with the proper coherency-related events is more useful for identifying the false sharing that most hurts performance.

Portability and Abstraction. Aside from a small, yet increasing, set of architected counters, subsequent generations of Intel processors are deceptively diverse. For example, we can count cache misses and references on Core 2, but hits and misses on Core i7. In conjunction with the last-branch record, it is necessary to use a counter for taken branches retired (because the LBR only stores taken branches) to get unbiased samples. This exists on Core 2, but the event does

not exist (they forgot?) on Core i7! On the Pentium 4, `fldcw` counts as two instructions, but on all other x86 implementations, it counts as one [40].

When counting system-wide (user and kernel) events, there may be many instructions executed between a counter interrupt and actually reading the counter value. Some systems support an atomic freeze that stops counting on an interrupt until it is reset, yet some do not. For those that do not, tools must estimate the perturbation error due to over counting.

Users are very rarely interested in monitoring exactly one platform. Typically, they are trying to collect the same type of data across a heterogeneous fleet of machines. This is true in supercomputing, this is true in data centers, and this is true for client applications. Even on a single machine, there is a steep learning curve and validation phase to gather event counts. The diversity of counters and subtle semantic differences across generations make data collection many times more challenging. IBM has recognized this problem and architected 32 counters in POWER6 that will be available on subsequent revisions.

Software Support. VTune and PTU are decent if you want to profile few applications/inputs on a single machine, but they do not scale well. Most performance tuning for Intel architectures probably occurs on Linux machines, but the vanilla kernel only supports OProfile [15]. OProfile is good for system-wide event sampling, but it does not support any features beyond basic EBS. Perfmon is a very nice interface, but it has been struggling to become part of the vanilla kernel for years. Patching and recompiling the kernel is an involved process, and it is a showstopper if the machine administrators are unwilling to do the work.

Even with good interfaces like Perfmon installed, you have to be an expert to do most things. PAPI [41] does provide some nice abstractions and an API to poll performance counters from within an application. There are group of Linux kernel developers who are trying to make the world simple and uniform with the `perf_event` project, but that approach does not capture the subtleties in the counters or expose advanced PMU features.

From another perspective, Linux users have all the luxury. FreeBSD has no support for Core i7. Windows tools for Intel x86 start and end with VTune and PTU.

We need tools that scale from the novice to the expert, and we need more than one for each layer. Lower level interfaces like Perfmon should be flexible and enable programming of every HPM feature, and they need to be open source. Low-level controls are by far the most important aspect, and it is a shame they are not better supported by the processor manufacturers. Higher level tools should abstract features so we can ask, for example, how many L2 cache misses were there? It should be able to respond with a breakdown of misses caused by instructions, by page table walks, DMA traffic, coherence traffic, etc. Some of these things may be measurable, but some may have to be guessed based on the available counters. If such a tool exists, it might provide guidance for future PMU designers to see what is most important to get right.

Virtualization. Virtualization is becoming increasingly commonplace, especially in markets like datacenters where performance analysis is paramount. However, currently, other than the LWP specification, no system we are aware of has support for virtualization.

Overhead. To read performance counters, at least one context switch to the kernel is required and may cost 1000 cycles itself. Beyond that, the cost of reading a single counter register varies greatly between microarchitectures: Opteron 1.4: 14 cycles, Athlon64: 20 cycles, Pentium IV (model 3/2): 226/146 cycles, PentiumPro: 33 cycles, PPC750: 2 cycles [42]. If they are to be read frequently, there is a high potential for perturbation and bias simply from reading the counters.

Some HPM features like LWT and LBR support buffering, but most do not. It is rarely the case that the user wants to react when a specific event happens. More often, the interrupt handler just buffers information somewhere else. Sampling overhead in general could be reduced with more HPM features supporting buffering.

User-level Access and Management. Needing root access to measure performance (or at least, install suid performance-measuring software) is a significant barrier to entry, as is needing to compile modules or recompile the Linux kernel for support for many features. AMD's LWT avoids these issues entirely by allowing full userspace control over its features. Some hardware

currently allows user processes to read performance counters if they are mmapped into user memory, but user processes cannot perform any configuration or modification. Some may argue that hardware events are a security threat and can be used to exploit timing attacks in other processes. We envision few systems where such a threat really is a concern, and suspect those which are do not give users shell access to begin with. If they in fact do, features like LWT could be disabled by the kernel or BIOS. Other metrics, like instruction mix counters or last branch records, offer no threat if virtualized within a process.

Documentation. HPM documentation is critical, and the documents need to specify exactly what is getting counted. However, the documentation (and possibly, the HPM itself), seems to be outsourced to the legal department. Many times, documentation is intentionally vague to avoid liability and expectations from users. Any time we want to count something, we first develop a microbenchmark to see if the counter meets our expectations. Usually, our first attempt is wrong. It seems that many people do this, so the work is unnecessarily duplicated, and better documentation or open microbenchmarks might save everyone some effort. For Intel architectures, there are the official docs in Volume 3B of the x86 manuals, then there is the most up to date information which is hidden in a VTune. The disconnect is due to the lag time to update the official manuals. Sprunt proposes encoding HPM descriptions and specifications in XML to avoid this disconnect [43].

We have recently done some experiments trying to understand the effects of the `prefetchnta` instruction so that we can do some memory optimizations. Using some microbenchmarks, and we could not even make the cache miss numbers add up. Depending on how one counted L1 misses (there are multiple ways), different numbers are produced. One of those sets of numbers agreed with the number of L2 references, the other did not. However, the one that did not match our intuition about how many times the program should miss in the L1 cache. It seems one counter counts only misses caused by load instructions, while the other also includes misses caused by hardware page table walks. These kind of omissions are commonplace.

In another example documented by Sprunt [25], the architects event definition may not be fully understood by the designer, resulting in events that are too broken to be useful. On the P6, the architect specified to count memory references that miss the DTLB, but the designer interpreted it as count the times the DTLB is referenced, with no match. This is problematic because canceled, conditional μops for string instructions all miss the DTLB, so DTLB miss counts can be unpredictably too high.

If there is still any doubt, consider this: Figure 3a shows that 100% of people want better documentation. A plausible theory is that vague documentation is an attempt to conceal details about a chip's microarchitecture, but discussions with internal engineers reveal that their resources are no better. It is wonderful that there are events to count nearly every aspect of the processor (and off-core systems as well), but the length of event descriptions is not commensurate with their complexity. Similar events need differentiation. Obscure events need better descriptions. Broken events need errata, and the event descriptions should refer to the errata if it is in a separate location. Subtle feature limitations, like PEBS not always being precise, must be disclosed.

Improved documentation of the HPM features is not enough; for users to leverage HPM, they must know more detail about how the hardware actually works. Researchers have invested their time to reverse engineer cache attributes and policies for Core i7 that should simply be in the manual [44]. Hardware optimizations like the loop stream detector need to be well-documented if vendors would like anyone other than their own employees to take advantage of them.

Interpretation. CPI stacks are nice for identifying the biggest performance bottlenecks, but they are difficult to construct for OoO machines. The information that is available is usually in the form of ratios like last-level cache misses per cycle or branch mispredicts per instruction. For non-experts, these ratios have very little qualitative meaning; how does one know when something is "bad"? Of course, latencies can be masked, so there is not a true measure of an event's cost, but an event's documentation should provide some guidance about what ranges can be considered mildly, moderately, or highly problematic. For HPC codes, peak floating point performance is the goal, but for most other codes there is no notion of peak.

For any metric, defining a qualitative value is a challenge in itself. One approach is Intel's Platform Modeling Tool (PMT) [16] with machine learning. PMT uses many application runs to sample a large set of performance counters and uses the data to identify which events are most correlated with instruction throughput. Another approach would be to create a synthetic benchmark to stress each event and define that as a high water mark for the event. Alternately, high water marks from standard benchmarks could be used as well.

4 Opportunities and Alternatives

4.1 A Pragmatic Proposition

To quote Douglas Adams, via Robert Fowler, "their fundamental design flaws are completely hidden by their superficial design flaws" [45]. Let us return to the fundamentals.

The previous section details our grievances with many different aspects of hardware performance monitoring. Here, we propose a few tenable changes that would alleviate much of the burden in performance tuning.

One issue, trust, lies at the heart of the HPM debate, and it relates directly to each of the issues above. Features frequently have vague documentation, unforeseen bias, or are outright broken, so users are often measuring something different from what they expect. For adoption of HPM to increase broadly beyond vendors' own tools, only the reliable features should be exposed and documented along with any caveats that may exist.

Remember the Average User. A significant amount of recent research is devoted solely to collecting unbiased edge, path, and call stack profiles [46, 31, 37, 47], and one such paper even won the best paper award at PLDI 2009 [2]. This ostensibly simple feature should have long ago become a commodity, programmable and accessible from user space.

Cache, TLB, and branch misses are historically, and will continue to be, the largest performance sinks. Each of these can of course be broken down into many subcomponents, and that is the problem – it is sometimes hard for the user to decide which of the hundreds of events should be used to measure a simple thing. But it should be easy. This problem spans hardware, software, users, and documentation, so each of these groups should be involved in a solution.

In addition to precision, IBS provides average latency and flags for events like cache/TLB/branch misses. For tuning stages 1 and 2, this is incredibly powerful. Some may argue that long latency instructions are not necessarily performance bottlenecks because OoO hides latencies, but we have never met a cache miss we liked – especially a frequently occurring one.

Forget the Expert. In being able to count a plethora of events, expert users are already well-served, but experts, too, are hindered by the same challenges as average users.

Free the Software. Processor vendors have their own internal tools and drivers for controlling HPM, and they underlie tools like VTune and CodeAnalyst [48]. They should be decoupled (or the underlying tools should be documented), and they should be free and open source. **We contend that open sourcing tools and drivers will add more value to the underlying hardware than selling closed-source tools.**

Many HPM tools are easy to run, but they require the user to specify which events will be counted. Instead, there should be a default mode that requires no parameters and collects an overview of where time is spent *and* what critical architectural events occur. Drivers like Perfmon should be finally pushed into OS kernels because installing it is a day-long chore or more. Once installed, root access should not be required to use the tools.

4.2 Feature Requests

We would be thrilled to see LWP and ProfileMe-like features offered on Intel x86 hardware. With increasingly complex hardware, instruction-based sampling could use a renovation as well. Extensions approach that allow programmable fields and filtering for instruction samples would be a killer features.

The PAPI model is very popular amongst programmers, not just for its abstraction, but because it allows caliper-style measurement of regions of code. PAPI has drawbacks, though. For small code regions, there is a relatively high overhead for system calls to read counters. Also, simply adding PAPI function calls to the code can alter register allocation, stack alignment, heap alignment, and instruction alignment, so there is potential for serious measurement perturbation. Instead of hardware, it would require compiler support to be easy to use, but we would like to see a feature that uses patching or breakpoints to perform PAPI accounting.

5 An Adventurous Approach

Consumers want hardware to collect and manage complex information, and they all want something different. Hardware designers want to count what is easily countable and have software synthesize conclusions. Clearly, the complexity must be handled somewhere. In over a decade, we have not seen any real advancement in the capabilities of HPM. At the same time, we have seen great advances in software instrumentation tools [49–53].

We propose to abandon HPM entirely in favor of a purely software profiling model. There is already some movement in this direction [54, 55]. Few people probably remember this, but DCPI/ProfileMe also used an interpretation mode to do value profiling to better understand dependences and complex interactions. Prior to DCPI [14] and Morph [56], it was done entirely this way, and we believe the shift to multicore processing makes it more feasible now than before. Techniques like bursty tracing [57, 58] and shadow profiling [59] enable long execution traces to be collected with overheads around 1%, which is comparable to current continuous profiling infrastructures.

Collecting execution traces is cheap and easy, and the real challenge lies in analyzing the traces. To the extreme, this involves a full pipeline simulation of the underlying microprocessor. For most common performance issues, simpler models of hardware will suffice.

One could even imagine a scenario where users upload execution traces to hardware vendors and the vendors' internal simulation tools are used to generate a detailed report breaking down stalls and interactions for a variety of ISA-compatible microarchitectures. In this arrangement, both parties win. The users can better tune their code, even for architectures they do not have access to. The vendor does not have to reveal the inner-workings of its design and effectively crowdsources collection of a previously unimaginable wealth of workloads to design future architectures. The workload distribution would even be biased toward those who care most about performance. If the system were to become popular, users could be given a certain number of free runs and charged for more intensive use. We anticipate customers from Wall St., datacenters, and HPC domains would pay for this type of service.

This type of thing actually happens regularly in practice. A developer with a particularly troublesome performance anomaly will send a trace (or the whole application) to an vendor's engineer for analysis. No data would be necessary with the traces, though some clients may desire very protective agreements for the instructions as well.

More realistically, we do not expect companies to publish these details, but fortunately, most performance issues are microarchitecture-independent. Hoste et al. show that a selection of architectural metrics like instruction mix, data stride patterns, and register conflicts can be used to predict relative processor performance with a 0.89 correlation coefficient [60]. In fact, much microarchitecture-specific tuning is done automatically or blindly anyway [24, 9], and the biggest issue is often data organization to benefit the memory hierarchy [11]. The variety of hardware underlying large-scale computing increases the appeal of generalized profiling.

Pure software profiling may be limited in modeling unpublished microarchitectural details, so it will have its own sources of bias. But to the user, complexity is greatly reduced. Any metrics of interest can be measured and correlated simultaneously, and precision is guaranteed. Program introspection enables heap [61] and locality analysis as well. Much more powerful analyses, like idealized execution [62–64] can also be performed. With a little cooperation from the hardware designers, even things like prefetching and subtle anomalies like those described in Section 3 can easily be modeled.

A less radical approach might employ synergy between hardware and software. HPM can be used to report issues that are too complex to model in software, precisely attribute where time is

spent, and identify troublesome spots where tracing should be done. Hardware could even offer logging features to expedite tracing, especially for multithreaded programs.

6 Conclusion

HPM can provide invaluable insight into application performance, but it is unreachable, overly complex, or easily misinterpreted by many users. Volume hardware purchasers tend to also be performance experts, but even they have tremendous difficulty measuring seemingly simple information. This paper highlights the key problems with HPM from an application performance tuning perspective and proposes potential improvements. The fundamental problem with HPM today is measuring simple things is just as difficult as measuring complex things. The complexity of data collection should be commensurate with the complexity of data collected.

Acknowledgment

Many of the examples and positions in this paper were documented with the help of Zia Ansari (Intel), Lee Baugh (Intel), Mickaël Ivascot (U. de Versailles), Todd Mytkowicz (U. of Colorado), Naveen Neelakantam (U. of Illinois, Intel), Sonny Rao (IBM), Alex Shye (Northwestern, AMD), Manish Vachharajani (U. of Colorado, LineRate Systems), Stéphane Zuckerman (U. de Versailles). David Levinthal (Intel) was instrumental in keeping us honest.

References

1. L. Liu and S. Rus, "Perflint: A context sensitive performance advisor for c++ programs," in *Code Generation and Optimization, IEEE/ACM International Symposium on*. IEEE Computer Society, 2009.
2. N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan, "Binary analysis for measurement and attribution of program performance," in *PLDI*, 2009.
3. T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *Proceedings of the 2007 International Conference on Computing Frontiers*, May 2007.
4. G. D. Price, J. Giacomoni, and M. Vachharajani, "Visualizing potential parallelism in sequential programs," in *PACT*, 2008.
5. C. B. Zilles, "Benchmark health considered harmful," *SIGARCH Computer Architecture News*, 2001.
6. T. Moseley, D. Grunwald, and R. V. Peri, "Optiscope: Performance accountability for optimizing compilers," in *CGO '09: Proceedings of the International Symposium on Code Generation and Optimization*. Seattle, WA, USA: IEEE Computer Society, 2009.
7. T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*, 2009.
8. T. Moseley, A. Shye, V. J. Reddi, M. Iyer, D. Fay, D. Hodgdon, J. L. Kihm, A. Settle, D. Grunwald, and D. A. Connors, "Dynamic run-time architecture techniques for enabling continuous optimization," in *Proceedings of the 2005 International Conference on Computing Frontiers*, May 2005.
9. D. Knights, T. Mytkowicz, P. F. Sweeney, M. C. Mozer, and A. Diwan, "Blind optimization for exploiting hardware features," in *Conference on Compiler Construction*, 2009.
10. Z. Pan and R. Eigenmann, "Fast, automatic, procedure-level performance tuning," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM Press, 2006, pp. 173–181.
11. C. R. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, 1998.
12. J. Callister, "Confessions of a performance monitor hardware designer," in *Workshop on Hardware Performance Monitor Design and Functionality colocated with HPCA*, 2005.
13. "Amd lightweight profiling specification." [Online]. Available: http://developer.amd.com/cpu/LWP/Pages/default.aspx
14. J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: where have all the cycles gone?" in *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1997, pp. 1–14.
15. "OProfile. http://oprofile.sourceforge.net." [Online]. Available: http://oprofile.sourceforge.net

16. “Intel platform modeling tool with machine learning.” [Online]. Available: http://software.intel.com/en-us/articles/intel-platform-modeling-with-machine-learning/
17. *Intel64 and IA-32 Architectures Software Developer’s Manual - Volume 3B*, Intel Corporation.
18. 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 *International Symposium on Microarchitecture*, 1997, pp. 292–302. [Online]. Available: citeseer.ist.psu.edu/dean97profileme.html
19. P. Drongowski, “Instruction-based sampling: A new performance analysis technique for amd family 10h processors,” 2007.
20. Intel Corporation, “Intel Itanium 2 processor reference manual: For software development and optimization,” May 2004.
21. “Workshop on hardware performance monitor design and functionality colocated with hpca,” 2005. [Online]. Available: http://lacsi.rice.edu/workshops/hpca11
22. “v2 of comments on performance counters for linux (pcl),” 2009. [Online]. Available: http://lkml.org/lkml/2009/6/16/432
23. H. C. Hunter and R. Nair, “Refining performance monitor design,” in *Proceedings of the 2004 Workshop on Complexity Effective Design (WCED)*, 2004.
24. J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. O’Boyle, G. Fursin, and O. Temam, “Automatic performance model construction for the fast software exploration of new hardware designs,” in *International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2006)*, October 2006.
25. B. Sprunt, “Performance monitoring hardware will always be a low priority, second class feature in processor designs until,” in *Workshop on Hardware Performance Monitor Design and Functionality colocated with HPCA*, 2005.
26. T. Moseley, J. L. Kihm, D. A. Connors, and D. Grunwald, “Methods for modeling resource contention on simultaneous multithreading processors,” in *Proceedings of the 2005 International Conference on Computer Design (ICCD)*, October 2005.
27. E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham, “Using model trees for computer architecture performance analysis of software applications,” in *ISPASS*, 2007.
28. X. Dai, A. Zhai, W.-C. Hsu, and P.-C. Yew, “A general compiler framework for speculative optimizations using data speculative code motion,” in *CGO ’05: Proceedings of the international symposium on Code generation and optimization*, 2005.
29. M. M. Canturk Isci, Gilberto Contreras, “Hardware performance counters for detailed runtime power and thermal estimations: Experiences and proposals,” in *Workshop on Hardware Performance Monitor Design and Functionality colocated with HPCA*, 2005.
30. T. Moseley, “Adaptive thread scheduling for simultaneous multithreading processors,” Boulder, CO, March 2006.
31. A. Shye, M. Iyer, T. Moseley, D. Hodgdon, D. Fay, V. J. Reddi, and D. A. Connors, “Analyis of path profiling information generated with performance monitoring hardware,” in *INTERACT ’05: Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT’05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 34–43.
32. A. Shye, B. Özisikyilmaz, A. Mallik, G. Memik, P. A. Dinda, R. P. Dick, and A. N. Choudhary, “Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction,” in *ISCA*, 2008.
33. M. M. Tikir, B. R. Buck, and J. K. Hollingsworth, “What we need to be able to count to tune programs,” in *Workshop on Hardware Performance Monitor Design and Functionality colocated with HPCA*, 2005.
34. I. Tuduce and T. Gross, “Efficient collection of information on the locality of accesses,” in *Workshop on Hardware Performance Monitor Design and Functionality colocated with HPCA*, 2005.
35. B. Brantley, “The NUMA challenge,” in *Workshop on Hardware Performance Monitor Design and Functionality colocated with HPCA*, 2005.
36. A. Rishi and J. A. Masamitsu, “Us patent no. 5953530. method and apparatus for run-time memory access checking and memory leak detection.”
37. T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox, “Hardware-based profiling: an effective technique for profile-driven optimization,” *Int. J. Parallel Programming*, 1996.
38. B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, “Interaction cost and shotgun profiling,” *ACM Trans. Architecture Code Optimization*, 2004.
39. C. B. Zilles and G. S. Sohi, “A programmable co-processor for profiling,” in *HPCA*, 2001.
40. V. M. Weaver and S. A. McKee, “Can hardware performance counters be trusted?” in *IISWC*, 2008.
41. P. Mucci, N. Smeds, and P. Ekman, “Performance monitoring with papi using the performance application programming interface,” *Dr. Dobb’s*, 2005. [Online]. Available: http://www.ddj.com/development-tools/184406109

42. P. Mucci, “Towards a flexible and realistic hardware performance monitor infrastructure,” in *Workshop on Hardware Performance Monitor Design and Functionality colocated with HPCA*, 2005.
43. B. Sprunt, “Managing the complexity of performance monitoring hardware: The brink andabyss approach,” *Int. J. High Perform. Comput. Appl.*, 2006.
44. R. S. Daniel Molka, Daniel Hackenberg and M. S. Mller, “Memory performance and cache coherency effects on an intel nehalem multiprocessor system.”
45. R. Fowler, “Performance hardware if i ran the world,” in *Workshop on Hardware Performance Monitor Design and Functionality colocated with HPCA*, 2005.
46. R. Levin, I. Newman, and G. Haber, “Complementing missing and inaccurate profiling using a minimum cost circulation algorithm,” in *HiPEAC*, 2008.
47. D. C. Todd Mytkowicz and A. Diwan, “Inferred call path profiling,” in *OOPSLA*, 2009.
48. “AMD CodeAnalyst.” [Online]. Available: http://developer.amd.com/cpu/CodeAnalyst/Pages/default.aspx
49. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2005, pp. 190–200.
50. N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, California, USA, 2007.
51. “Dyninst: An application program interface (api) for runtime code generation.” [Online]. Available: http://www.dyninst.org
52. D. L. Bruening, “Efficient, transparent, and comprehensive runtime code manipulation,” Ph.D. dissertation, Cambridge, MA, USA, 2004.
53. P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Computer*, 2002.
54. “Valgrind’s tools suite.” [Online]. Available: http://valgrind.org/info/tools.html
55. K. Hoste and L. Eeckhout, “Microarchitecture-independent workload characterization,” *IEEE Micro*, vol. 27, no. 3, pp. 63–72, 5 2007.
56. X. Zhang, Z. Wang, N. C. Gloy, J. B. Chen, and M. D. Smith, “System support for automated profiling and optimization,” in *SOSP*, 1997.
57. M. Hirzel and T. Chilimbi, “Bursty tracing: A framework for low-overhead temporal profiling,” in *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4).*, 2001. [Online]. Available: citeseer.ist.psu.edu/hirzel01bursty.html
58. M. Arnold and B. G. Ryder, “A framework for reducing the cost of instrumented code,” in *SIGPLAN Conference on Programming Language Design and Implementation*, 2001, pp. 168–179. [Online]. Available: citeseer.ist.psu.edu/arnold01framework.html
59. T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. V. Peri, “Shadow profiling: Hiding instrumentation costs with parallelism,” in *CGO ’07: Proceedings of the International Symposium on Code Generation and Optimization*. San Jose, CA, USA: IEEE Computer Society, 2007.
60. K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. D. Bosschere, “Performance prediction based on inherent program similarity,” in *PACT*, 2006.
61. R. Shaham, E. K. Kolodner, and M. Sagiv, “Heap profiling for space-efficient java,” in *PLDI ’01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001.
62. L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby, “Exploring application performance: a new tool for a static/dynamic approach,” in *Los Alamos Computer Science Institute Symp.*, Santa Fe, NM, Oct. 2005.
63. M. Iyer, C. Ashok, J. Stone, N. Vachharajani, D. A. Connors, and M. Vachharajani, “Finding parallelism for future epic machines,” in *Proceedings of the Fourth Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology (EPIC)*, 2005.
64. G. Fursin, M. O’Boyle, O. Temam, and G. Watts, “Fast and accurate method for determining a lower bound on execution time,” *Concurrency: Practice and Experience*, vol. 16, no. 2-3, pp. 271–292, 2004.