Rapid Identification of Architectural Bottlenecks via Precise Event Counting

John Demme Simha Sethumadhavan

Computer Architecture and Security Technologies Lab
Department of Computer Science
Columbia University
New York, NY, USA
{jdd,simha}@cs.columbia.edu

ABSTRACT

On-chip performance counters play a vital role in computer architecture research due to their ability to quickly provide insights into application behaviors that are time consuming to characterize with traditional methods. The usefulness of modern performance counters, however, is limited by inefficient techniques used today to access them. Current access techniques rely on imprecise sampling or heavyweight kernel interaction forcing users to choose between precision or speed and thus restricting the use of performance counter hardware.

In this paper, we describe new methods that enable precise, lightweight interfacing to on-chip performance counters. These low-overhead techniques allow precise reading of virtualized counters in low tens of nanoseconds, which is one to two orders of magnitude faster than current access techniques. Further, these tools provide several fresh insights on the behavior of modern parallel programs such as MySQL and Firefox, which were previously obscured (or impossible to obtain) by existing methods for characterization. Based on case studies with our new access methods, we discuss seven implications for computer architects in the cloud era and three methods for enhancing hardware counters further. Taken together, these observations have the potential to open up new avenues for architecture research.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques*; B.8.2 [**Hardware**]: Performance and Reliability—*Performance Analysis and Design Aids*

General Terms

Measurement, Performance

Keywords

Performance Evaluation, Hardware Performance Counters, Locking

ISCA'11, June 4–8, 2011, San Jose, California, USA.

1. THE NEED FOR RAPID, PRECISE EVENT COUNTING

These are exciting times for computer architecture research. Today there is significant demand to improve the performance and energy-efficiency of emerging, transformative applications that are being hammered out by the hundreds for new compute platforms and usage models. This booming growth of applications and the variety of programming languages used to create them is challenging our ability as architects to rapidly and rigorously characterize these applications. Consequently, developing optimizations for these applications is becoming harder.

On-chip performance counters offer a convenient alternative to guide computer architecture researchers through the challenging, evolving application landscape. Performance counters measure microarchitectural events at native execution speed and can be used to identify bottlenecks in any real-world application. These bottlenecks can then be captured in microbenchmarks and used for detailed microarchitectural exploration through simulation.

Recently, some hardware vendors have increased coverage, accuracy and documentation of performance counters making them more useful than before. For instance, as shown in Figure 1, about 400 events can be monitored on a modern Intel chip, representing a three-fold increase in a little over a decade. Despite these improvements, it is still difficult to realize the full potential of hardware counters because the costly methods used to access these counters perturb program execution or trade overhead for loss in precision. We redress this key issue in this paper with cheaper new access methods and illustrate how these methods enable observation of a range of new phenomena.

Popular tools used for accessing performance counters today such as PAPI [12], OProfile [13] or vTune [17] attempt to read performance counters via hardware interrupts or heavyweight kernel calls. An inherent downside of kernel calls is that they interrupt normal program execution and slow down the program thereby affecting the quantity being measured. To minimize these perturbations, most profilers resort to occasionally reading these counters and extrapolating full program statistics from the sampled measurements. While this extrapolation is necessarily imprecise, the error introduced by the process has been acceptable when profiling hotspots in serial programs.

Traditional sampling, however, has fundamental incompatibilities for parallel programs which have become com-

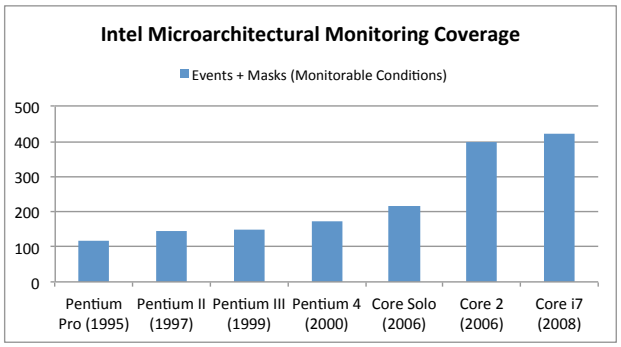


Figure 1: Number of countable conditions using Intel's performance monitoring framework through several generations.

monplace with the availability of multicores. Traditional sampling methods are likely to miss small critical sections because they do not constitute the hottest regions of the code. Amdahl's law, however, teaches us that optimizing critical sections is necessary to ensure scalability, even if the time spent in critical sections is relatively low [9]. Moreover, as we will discuss in Sec 3.1, irrespective of the size, it is not easy to correctly monitor critical sections. Performance characterization of parallel programs with performance counters calls for simple, lightweight access methods that can enable *precise* performance measurement for both hot and cold code regions.

In this paper, we describe novel lightweight techniques for accessing performance counters and report new application behaviors which are difficult to capture with existing access methods. Our precise access method, embodied in an x86-Linux tool called **LiMiT** (Lightweight Microarchitectural Toolkit), requires less than 12 *ns* per access and is over 90x faster than PAPI-C [12] and 23x faster than Linux's perf_event, tools that provides similar functionality.

Based on three case studies with **LiMiT** using unscaled, production workloads we put forth several recommendations for architecture researchers.

In our first case study, we measure synchronization regions in production applications (Apache, MySQL and Firefox) as well as the PARSEC benchmark suite. Our measurements show that Firefox and MySQL spend nearly a third of the execution time in synchronization which is 10x more than the synchronization time in PARSEC benchmarks. These results indicate that synchronization is used differently in production system applications than traditionally-studied scientific/numerical applications and architects must be aware of these differences. Performing similar measurements with PAPI-C show inflated synchronization times due to high measurement overheads, drastically changed cycle count ratios and increased instrumentation overheads from 42% to over 745%. Some workloads such as Firefox could not even run properly with PAPI-C because of the high overheads.

Our next case study examines the interaction of programs with the Linux kernel via popular library calls. This interaction has not received much attention because of the difficulty in running modern, unscaled web workloads on full-system simulators. Our investigation reveals that production applications spend a significant fraction of execution cycles in dynamically linked libraries and operating system calls. Further, we find that routines in these two segments show distinctly different microarchitectural performance characteristics than userspace behavior.

The third and final case study demonstrates **LiMiT**'s breadth of utility by conducting longitudinal studies of modern software evolution. By examining the evolution of locking behaviors over several versions of MySQL, we investigate if there has been a return on investment in parallelizing the software for multicores. This study illustrates how the utility of precise counting goes beyond traditional applications in architecture, compilers and OS, and that well-architected performance counting systems can have wide and deep impact on several computer science disciplines.

Finally, we suggest modest hardware modifications — based on our experiences with **LiMiT** — that can increase the precision and utility of performance counters even further. Specifically, we suggest (1) a destructive performance counter read instruction for lower overheads (2) 64-bit counters, and instructions that can read and write to the full 64 bits to avoid overflows and (3) integration of counter selection into the read instruction. The combination of these three features would allow single instruction counter readouts and resets.

The rest of this paper is laid out as follows: in Section 2 we describe the evolution of and existing performance counter systems. Section 3 describes access techniques used in **LiMiT** and compares **LiMiT** to existing approaches. The case studies are presented in Sections 4, 5 and 6. We present suggestions for enhancing performance counters with hardware support in Section 7 and closing thoughts are presented in Section 8.

2. PERFORMANCE COUNTERS REVIEW

Performance counter based studies have proved exceedingly valuable in the past, and many influential research studies have been based on performance counter measurements of production systems. Emer and Clark shaped quantitative computer architecture with their seminal work on characterization of the VAX system using hardware counters [8]. Anderson *et al.* described results from system wide profiling on Alpha machines [4]. Ailamaki *et al.* describe results of profiling DBMS applications [3]. Keeton *et al.* characterized OLTP workloads on the Pentium Pro Machine [10]. Like these papers, we use novel performance measurement methods to study contemporary applications.

Performance counters started appearing in commercial machines in the '90s. The performance counter access facilities in these machines were intentionally minimalist to reduce area overheads. For instance, initial designs of the Alpha 21064, one of the first machines to include performance counters, did not even have read/write access to the performance counters. To keep chip-area overhead tiny, the counters interrupted processor execution when a counter overflowed, allowing only basic sampling support based on interrupts [16]. As the usefulness of the counters became clear and transistors became cheaper, later Alpha chips and other vendors enhanced their performance counter infrastructure. By the late '90s, all of the major processor lines, including Pentium, PPC, UltraSparc, PA-RISC and MIPS processors included performance counters and simple access methods.

A common feature of many of the counter designs in early processors – and a source of major frustration to date – is that all of these counters were accessible only in the priv-

ileged mode, thus requiring a high overhead kernel call for access. This problem was mitigated to an extent in the MIPS R10000 [18] (1995), which included support for both user-/kernel-level access to the performance counters. Later x86 machines from Intel and AMD have included similar configurable support. However, the software used to access the counters (kernel and libraries) often do not enable user space counter reads by default, likely to allow them to mask the complexity of counter virtualization behind the kernel interface. A recent proposal from AMD [2] published in 2007, discusses lightweight, configurable user space access. The proposed scheme appears promising but hardware implementations are not yet available.

Hand in hand with the hardware improvements, many software tools have been developed over the years to obtain information from performance counters. These tools can either pull data from the performance counters on demand (precise methods) at predetermined points in the program or operate upon data pushed by the performance counter (imprecise methods) during externally-triggered sampling interrupts. Intel's vTune [17] and DCPI/ProfileMe [7] are some commercial examples of tools that support only imprecise access methods. An open source example is the Performance API (PAPI) which was created in 1999 to provide an standard interface to performance counters on different machines [12]. OProfile [13] is another Linux profiling tool that provides interrupt-based sampling support. With these tools, users can extrapolate measurements obtained from samples collected either at predetermined points in the program or during sampling interrupts triggered by user specified conditions *e.g.,* N cache misses. A general drawback to these sampling methods is that it introduces error inversely proportional to the sampling frequency. As a result, short or cold regions of interest are difficult to measure precisely.

Examples of tools that provide precise performance monitoring access methods for Linux are perfmon2 [14], perf_event [1] and Rabbit [15]. Perfmon2 is an older Linux kernel interface which provides both sampling support and precise counter reads, though the precise read support requires system calls. The newly introduced perf_event interface is intended to replace perfmon2 but still uses system calls (the read syscall, specifically) for precise access to performance counters. Rabbit is an older access method written to avoid system calls, but provides none of the virtualization features of **LiMiT**, perfmon2 or perf_event.

All these tools require that performance counters be read by the kernel, requiring heavyweight system calls to obtain precise measurements. Unlike the above tools, our access techniques provide both precise and low overhead measurements by allowing userspace counter access. We compare our measurements to PAPI-C and perf_event, showing that by enabling userspace access, **LiMiT** introduces less perturbation than PAPI, and decreased overheads enable accurate, precise profiling of long running or interactive production applications.

3. ENABLING LOW-OVERHEAD PERFORMANCE COUNTER ACCESS

The key to low overhead counter reads is to avoid kernel calls by allowing user applications to directly read the performance counters. In this section, we detail the methods used to implement our interface and compare the overheads of our performance counter access method to existing methods.

Enabling userspace access is a three step process:

§ **1:** Stock Linux kernels do not allow direct user space access to performance counters. As a simple first step, we set the configuration bit (an MSR in x86) to allow user access.

§ **2:** Performance counters cannot be directly configured to monitor events of interest (*e.g.,* instructions retired) from userspace. We add a system call to the Linux kernel to configure the counters. Since most applications are likely to set up these counters once or few times per program we do not take any special measures to optimize this step.

§ **3:** A more involved third step is to enable process isolation by virtualizing the operation of the performance counter hardware, allowing multiple programs to use one hardware instance of the performance counters. Without this support, programs would read events which occurred while other programs were executing, resulting in incorrect results and also opening up side-channels that can be used to infer information about program execution.

In theory, virtualization support should be as simple saving and restoring the performance counters during context swaps just like any other register. However, we need to deal with the possibility of performance counters overflowing. Intel's 48 bit counters can overflow every 26 hours, so overflows are likely for long running applications. Additionally, Intel chips prior to Sandy Bridge allowed only 32 bit writes to the counters so after only 1.4 seconds, the kernel can find itself unable to correctly restore the counter when a process is swapped back in.

We work around overflows by detecting overflow conditions and accumulating the overflowed values in user memory. When a process wants to read a performance counter it must get the current value via `rdpmc` then fetch and add the contents of the overflow value in memory. However, this set of instructions must be executed atomically; if an interrupt and overflow occurs during their processing (before the memory fetch but after the `rdpmc`) then the value read will be off by the previous value of the counter, as the kernel has zeroed the (already read) register and incremented the (as-yet-unread) overflow variable.

Two obvious solutions to ensure atomic execution, turning off interrupts or protecting the critical section with a lock, cannot work in this context. If we disable interrupts, the executing process would never be swapped out and could starve other applications. Further, allowing an user process to disable external interruption is dangerous. Locking is even more problematic. Our algorithm requires the kernel to update the user space memory location that keeps track of the performance counter values. To do this the kernel must obtain a lock when the process is being swapped back in. However, if the process holds the lock, then the kernel cannot continue and the process will never resume to release the lock. In this situation deadlock is guaranteed.

Linux kernel interfaces such as Perfmon2 and perf_event deal with this problem by placing all sensitive code in the kernel where techniques like disabling interrupts can operate normally. By doing so, however, they add significant overhead to counter reads in the form of system calls to access counters.

To solve this problem, we use an approach similar to Bershad *et al.* [5] (Figure 2). We speculatively assume that

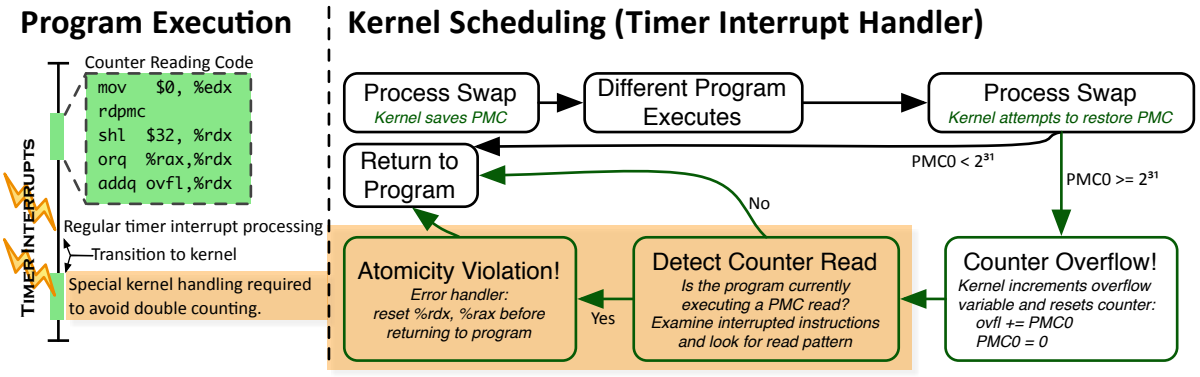


Figure 2: LHS figure shows LiMiT's five instruction counter read sequence (dotted box) embedded as part of regular program execution. As shown, program execution can be interrupted when the program is executing uninstrumented code or when executing user space code for reading counters. Interrupts received during counter read require special handling to avoid double counting bugs. RHS figure shows special modifications (highlighted boxes) that provide detection of interrupted counter reads and fixes for double counting bugs.

there will be no atomicity violation, but build detection and error handling into the kernel code for cases where such events happen. With this approach, there is no additional overhead added to counter reading code in user space and overhead is only incurred on relatively infrequent counter overflows. To detect whether or not an application is in the middle of a counter read during a counter overflow we simply check the pattern of instructions before the process was interrupted (pointed to by the process' instruction pointer). If a counter read is detected, the kernel zeros the process' registers (%rax and %rdx in the x86 example) to match the new (overflowed) contents of the performance counter. Once resumed, the program will behave as if the interrupt, context switch and overflow had occurred immediately prior to the read of the performance counter. The primary difference from the approach in Bershad *et al.* [5] is that they rewind execution to the beginning of the critical section instead of fixing up the correct counter values as we do.

3.1 Comparison to Sampling

Sampling typically is used in two ways: interrupt based or by polling. In interrupt based sampling, interrupts are triggered when a pre-determined event such as the number of committed instructions reaches a pre-determined count. These interrupts are received by the OS and passed on to the application. In polling based sampling, the counters are precisely read out once out of every N times a code region is executed to reduce overhead. While both approaches can have low overhead, there are a number of situations in which neither works well.

For example, Figure 3 contains a critical section from MySQL which accounts for 30% of MySQL's overall critical section time. Let us say that we are interested in measuring time spent in critical sections using interrupt based sampling. If K of the N samples were in critical section we would extrapolate that K/N of the total time was spent in critical sections. However, there are several complications with this approach. In the above example, a sampling interrupt routine which fires during the critical section, would have difficultly determining whether or not a lock is held because the locks are executed based on the `if` conditional preceding the lock.

An alternative to interrupt sampling is to use precise access methods intermittently. In this case, explicit performance counter reads would have be used every time a lock is acquired or released. To reduce overhead, performance counter reads could execute only once out of every N times the region is entered, and the total time could be extrapolated from this measurement. While this method is effective in reducing overall overhead, the overheads for each precise read remain high. As a result, large perturbation is introduced immediately before and after the region of interest when measurement is actually occurring. We would therefore expect measurements for small regions to be inflated. We observe this effect during our Case Study A in Figure 5b.

In many of these situations in which sampling or heavyweight precision present difficulties, *ad hoc* solutions are possible. However as our case studies demonstrate, a low-overhead, precise measurement like **LiMiT** is sometimes the right tool for the job.

```
40 if (info->s->concurrent_insert)
41   rw_rdlock(&info->s->
                key_root_lock[inx]);

42 changed= mi_test_if_changed(info);
43 if (!flag) {
44   switch(info->s->
           keyinfo[inx].key_alg) {
     /* 37 lines omitted */
82 }
84 if (info->s->concurrent_insert) {
85   if (!error) {
86     while (...) {
         /* 10 lines omitted */
97     }
98   }
99   rw_unlock(&info->s->
              key_root_lock[inx]);
100 }
```

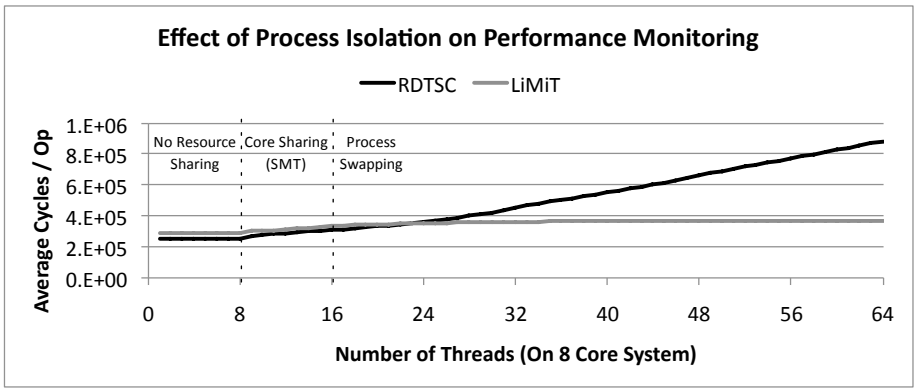
Conditional Locks

Figure 3: Code excerpt from MySQL 5.0.89, `mi_rnext.c`. The critical section shown here accounts for 30% of all the time spent in critical sections.

```
#define rdtsc(X)          \
asm volatile ("rdtsc;"    \
     "shl $32, %%rdx;"    \
     "orq %%rax, %%rdx;"  \
   : "=d"(X) :  : "%rax");

int main(void) {
  uint64_t b, e;
  rdtsc(b);
  for (uint64_t i=0;
       i<ITER; i++) {
    // ... some operation
  }
  rdtsc(e)
  printf("Time per op: %lf\n",
   ((double)e - b)/ITER);
}
```

(a) RDTSC Example



(b) RDTSC Isolation Effects

Figure 4: LHS: typical `rdtsc` usage example. RHS: Process isolation in LiMiT prevents other threads and processes from directly affecting event counts. RDTSC has no such ability.

3.2 Comparison to PAPI and perf_event

For years, PAPI has been the standard library to write cross platform performance monitoring tools. As a library, it relies on kernel interface support; traditionally it has used perfmon2 on Linux. In contrast, perf_event is the newest Linux kernel interface. It is touted to be faster and more featureful than perfmon2 and will thus eventually replace it. However, due to its relative youth, library support for perf_event remains poor, placing burden on the user but yield better speeds as there is no library overhead.

Any performance counter readout call (be it PAPI or **LiMiT**) will cost some number of cycles. To examine this overhead, we construct a short benchmark which samples a counter configured to count three events (cycles, branches and branch misses) 10^7 times each. With this high number of iterations, we can report the wall time for comparison of the overheads and compute the cost of each readout call. The results are presented in Table 1. On our Xeon 5550-based system, the average for **LiMiT**'s five instruction readout code is 37.14 cycles. Since **LiMiT** does not require a system call for each sample, it is substantially faster compared to PAPI-C (by 92x) and perf_event (by 23x).

In Section 4, we instrument MySQL to examine locking, unlocking and critical section timing (setup described in detail in the following section). Figure 5b shows that using **LiMiT** incurs a 42% cycle increase over uninstrumented execution. Further, when the same instrumentation is performed using PAPI, a 745% user space cycle overhead is introduced and a 97% is incurred with perf_event. Both PAPI's and perf_event's actual overheads, however, are much larger since over 90% of their overheads occur in kernel space (as shown in Table 1) but are not counted in figure 5b. As a result, we would expect both PAPI and perf_event instrumentation to perturb execution more than **LiMiT** making the results virtually unusable.

Time	PAPI-C	perf_event	**LiMiT**	Speedups	
User	1.26s	0.53s	0.34s	3.7x	1.56x
Kernel	30.10 s	7.30s	0s	∞	∞
Wall	31.44s	7.87s	0.34s	92x	23.1x

Table 1: Speedups of LiMiT, perf_event, and PAPI (10^7 reads of 3 counters) plus LiMiT's speedup over PAPI and perf_event respectively.

Overheads also directly affects usability. We attempted to instrument and measure modern cloud workloads such as Firefox, MySQL and Apache with both **LiMiT** and PAPI. Firefox was unresponsive to input with PAPI, while it operated with no discernible slowdown when instrumented with **LiMiT**. We also measured that Apache served 9,246 requests per second with **LiMiT** instrumentation and 9,276 requests per second without instrumentation. These minor changes in speed demonstrate **LiMiT**'s low overhead.

3.3 Comparison to RDTSC Measurements

Using `rdtsc`, the read time stamp counter instruction on x86 architectures, is *de rigeur* in userspace lightweight measurement. The time stamp counter is a free running counter present on all x86 machines. It simply counts bus cycles (uncore cycles for modern Intel processors) and most operating systems allow programs direct access to it. Since `rdtsc` is simple and lightweight, programmers will often use it to measure the time spent in short or long regions of code or to judge the effect of code changes on performance. **LiMiT**, however, offers capabilities that are superior to plain `rdtsc`: aside from a variety of counting events besides bus cycles, **LiMiT** provides process isolation which allows each process to shield its measurements from other processes' direct interference. While one could apply many of **LiMiT**'s techniques to `rdtsc`, this does not occur in practice so we compare against `rdtsc` without any such additions.

To examine the effect of process isolation, we construct a simple microbenchmark which executes non-memory operations across multiple threads on an 8 core system, allowing the operating system to schedule them onto cores. We then compute the average amount of time each operation takes using both `rdtsc` and **LiMiT**. We would expect the performance of each operation to degrade as resource sharing increases. There should be little or no performance degradation with 8 or fewer threads, mild degradation from 8 to 16 threads as SMT is utilized then a little more performance degradation above 16 threads as threads are swapped in and out. The data presented in Figure 4b confirm these expectations when using **LiMiT**. `rdtsc`, however, incorrectly reports massive, linearly increasing performance degradation above 16 threads.

4. CASE STUDY A: LOCKING IN WEB WORKLOADS

Usage patterns of computers have changed drastically over the past decade. Modern computer users live in the cloud. These users spend most of the their time in web browsers – either on a traditional desktop or mobile device – which moves computation to backend servers. As a result, there are two separate and extremely important workloads in the web model: the frontend, consisting of web browsers and Javascript engines, and the backend, consisting of HTTP servers, script interpreters and database engines. Further, the workloads of these applications have also changed. Often web pages rely far more on Javascript than ever before and database operations are no longer well modeled by traditional transactional benchmarks, often favoring scalability and speed over data security and transactional atomicity and durability.

We briefly characterize the synchronization behavior of several popular web technologies. Specifically, this study aims to answer the following questions: (1) Is synchronization a concern in web workloads and what are the locking usage patterns? (2) What future architecture directions can optimize web workloads? For comparison purposes, we also measure and analyze the PARSEC benchmark [6]. As a numerical workload, PARSEC is likely representative of traditional (scientific computing) notions of parallel programming and may be different from web technologies.

Necessity of LiMiT There are three features offered by **LiMiT** which enable this study: precise instrumentation, process isolation and low-overhead reads, not all of which are simultaneously offered by other technologies. Precision is necessary because we are capturing very short regions of executions – lock acquires/releases and critical sections – which are likely to be missed by sampling techniques. Process isolation (which is not offered by the traditional `rdtsc`) is required since we are operating in a multi-threaded environment with I/O, so processes are likely to be swapped in and out often. Finally, **LiMiT**'s low-overhead counter readout routine is required to prevent large perturbation from skewing results. To further examine **LiMiT**'s lowered overhead, we will compare results obtained with **LiMiT** to results obtained with PAPI.

Experimental Setup To gain insight into modern web workloads, we examine the following software and input sets:

Firefox A popular, open-source web browser, we ran Mozilla Firefox version 3.6.8. We visited and interacted with the top 15 most visited sites, as ranked by Alexa. Additionally, we used two web apps from Google, Gmail and Google Reader, two applications which rely heavily on AJAX, asynchronous Javascript and XML.

Apache The Apache HTTP server is, according to Netcraft, the most popular HTTP sever with 56% market share as of August 2010. We evaluated the latest stable version, 2.2.16, using the included "ab" (Apache Benchmark) tool to fetch a simple static page. A total of 250k requests were served with 256 requests being requested concurrently. Because we look only at static loads, the results will indicate a best-case scenario for Apache.

MySQL MySQL is the traditional database server of choice for websites. The most recent stable version is MySQL 5.1.50 Community Server, which we evaluated. To exercise it's functionality, we ran the "sql-bench" benchmarking scripts included with MySQL's source code.

PARSEC The PARSEC benchmark suite [6] is a set of parallel applications largely targeting RMS workloads. We executed seven of the multithreaded benchmarks: blackscholes, swaptions, fluidanimate, vips, x264, canneal and streamcluster.

We instrumented each of these applications using **LiMiT** to track their critical sections and locking behaviors. Specifically, we collected information on the number of cycles spent acquiring and releasing locks, and time spent with locks held.

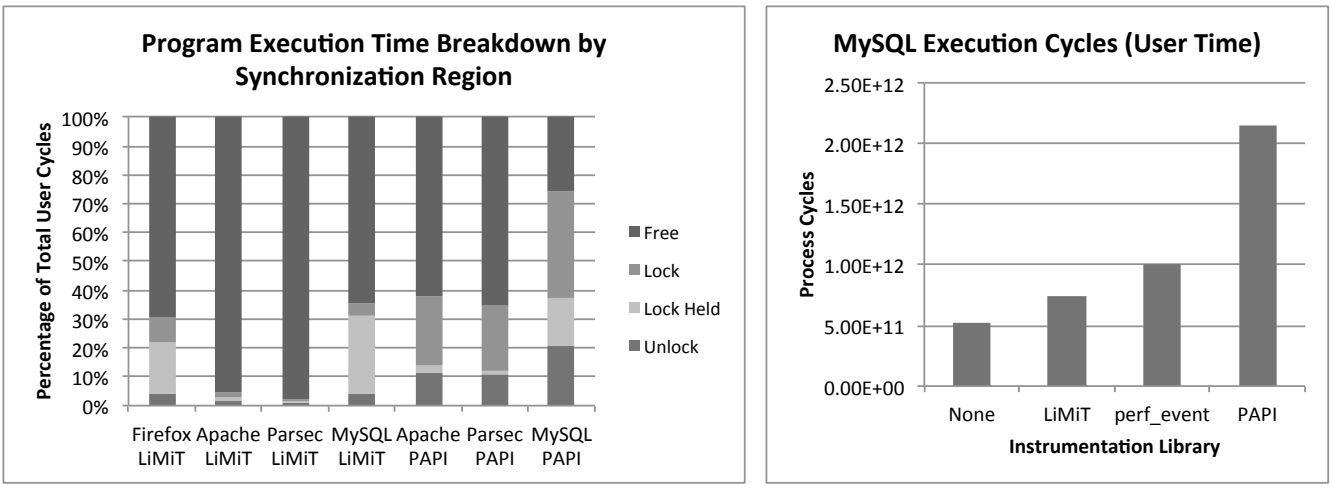
Results The charts in Figures 5 and 6 summarize the collected data. Figure 5 contains an overview of synchronization overheads and critical section times. Execution time is computed as the total number of cycles in all threads, lock and unlocking times as all time spent in `pthread_mutex_lock` and `pthread_mutex_unlock` in all threads. Lock held time, however, is defined as summation of the amount of time each thread has at least one lock held; if more than one lock is held, time is not double-counted.

These data show that this behavior varies a great deal between the applications. Figure 6 contain histograms of locking and unlocking overheads (latency of lock acquire and release) and times spent in critical sections. We break down this data by both dynamic locks (number of lock acquires during execution) and static locks (number of lock instances observed during execution), revealing insights about lock usage patterns. From this data, we make several observations:

Critical Section Times The histograms in Figure 6 indicate that the manner in which each application uses locks varies. PARSEC, for instance, holds locks for very short amounts of time in stark contrast to MySQL and Firefox. (See Table 2.) This is likely because many of PARSEC's applications parallelize nicely, *e.g.,* using data parallelism and static assignment. The other applications, however, are interactive and must respond to events as they occur. Since this makes static assignment impossible, threads must interact more often, requiring more synchronization.

Number of Locks The previous point is further supported by the number of locks shown in Table 2. Highly interactive applications like Firefox and MySQL require significantly higher number of locks. PARSEC is likely able to use only barrier-like constructs to synchronize computation.

Based on this data, we will attempt to answer the questions set forth. To answer our first question, about locking patterns in web workloads, we observe that synchronization is a mixed bag in web applications. Some workloads, like Apache, are likely to be very parallel and scale easily. MySQL does not fit into this category as it does not scale as easily. Additionally, Firefox has far more synchronization overheads then one would expect. Based on personal experience with Mozilla code, we suspect this is a result of difficulties in parallelizing legacy "spaghetti" code which is likely to have many side effects which must be isolated from other threads.



(a) Synchronization overheads

(b) MySQL cycles counts

Figure 5: Comparison of synchronization and critical section timing for various popular applications and the PARSEC benchmark suite along with execution times for MySQL. Results obtained with PAPI are inflated due to instrumentation overheads. We also see that PAPI instrumentation increases userspace cycle counts by more than 745% compared to LiMiT's 42% increase. We also note that Firefox (being an interactive program) could not execute with PAPI instrumentation.

	Firefox	Apache	PARSEC	MySQL
Average Lock Held Time	789	149	118	1076
Dynamic Locks per 10k Cycles	3.24	1.12	0.545	3.18
Static Locks per Thread per Application	57	1	17	13853

Table 2: Locking-related averages. We note that the vast majority of PARSEC's static locks are observed in one benchmark: fluidanimate. Without this benchmark, the number of static locks per thread per application drops to 0.575. These data indicate that scientific and web workloads have significant difference in synchronization behavior.

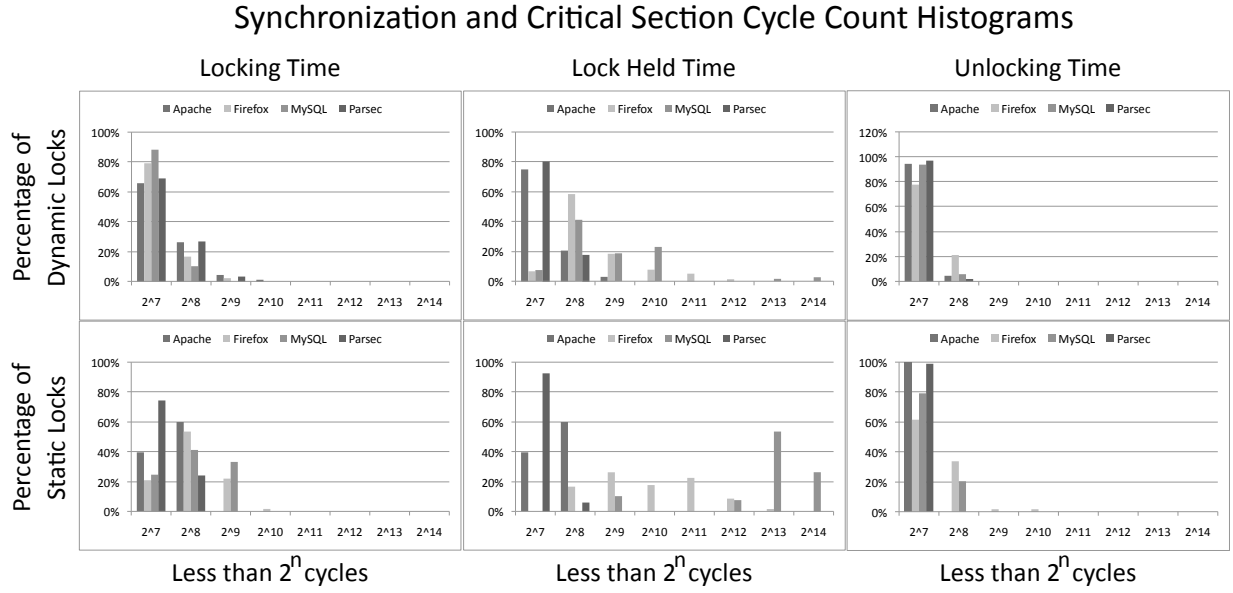


Figure 6: Histograms of synchronization overheads and critical section times for several applications. Times are broken down by dynamic locks (number of lock acquisitions) and average for each static lock (observed lock instance). We note that many critical section times are very short, comparable in cycle counts to lock acquisition times.

Implications for Architects (#1, #2, #3) Our second question — How are architects affected by these results and what future directions would best support the web? — bears further analysis. There are several interesting points:

 # 1: A new benchmark suite of web software may be necessary for new web-centric architecture research. SPEC has several versions of the "SPECweb" benchmark; future studies should include comparisons. However, many of the applications we have reviewed and other important cloud workloads are not part of SPECweb, including Firefox, Javascript, website supporting databases (non-transactional workloads), server caching and load balancing.

 # 2: Our data show locking overheads can be non-trivial compared to critical section times. Since locking/unlocking overheads can be 8% to 13% of overall cycles, speedups in this range may be possible with architectural/software techniques for streamlining lock acquisition. Further, we observe that the static lock distributions differ from the dynamic lock distributions, suggesting that one may be able to statically determine which locks are likely to be contended and which are likely to be held for many cycles.

 # 3: Critical section times for MySQL are relatively large. In particular, over half of the lock instances have average lock hold times around 8,000 cycles (although they are locked less often). These represent segments of code which will not scale well. These regions are prime targets for microarchitectural optimization. If they can be sped up, parallel performance and scalability of MySQL will improve.

5. CASE STUDY B: KERNEL/USERSPACE OVERHEADS IN RUNTIME LIBRARY

Our next case study is aimed at examining the interaction of programs with the Linux kernel via popular library calls and understanding their impact on program performance. A prior study has shown that kernel calls can negatively impact performance by polluting branch predictors [11]. Are there other on-chip structures that are affected by kernel calls? To what degree are modern applications affected by their kernel interaction? Is it possible to obtain fine-grained information about execution that can be tracked back to originating function calls? Our goal is to use **LiMiT** to study common library functions' behaviors in both userspace and kernel space.

Necessity of LiMiT There are two alternatives to using **LiMiT** for collecting this data.

First, simulation can be used to study the interaction of user and kernel code. Full system multiprocessor simulators can model the effect of system interaction and can shed light on effect of library calls but can be prohibitively slow without scaling workloads. Although **LiMiT** cannot achieve the accuracy and detail level of simulation, it can be used to rapidly gather precise information and coarsely locate problem regions.

The second option is sampling with external interrupts. This style of sampling provides an interrupt every N events at which point the sampling interrupt can analyze the application's execution state. In this study, however, we must determine which library functions use processor resources *and* the purpose of the function calls. For instance, we would like to know whether `memcpy` is manipulating program data or copying data for I/O. Obtaining this data in both user and kernel space is difficult for sampling-based methods as each sample interrupt must also run a stack trace (often from the kernel stack all the way back to and through the user stack) to identify the library entry point. To our knowledge, no existing sampling tool is able to track kernel function usage back to the calling userspace function. While theoretically possible for sampling, **LiMiT** makes this approach downright easy. With **LiMiT**, we read counters at the entry and exit points of functions in each category, so all events occurring between the function entry and exit, including all functions called from within the function, are counted towards that function. For example, if `pwrite` calls `memcpy` internally or the kernel executes some locking functions during a `read` system call, any microarchitectural events resulting from the `memcpy` or kernel locking will count towards `pwrite` or `read` rather than memory or locking categories.

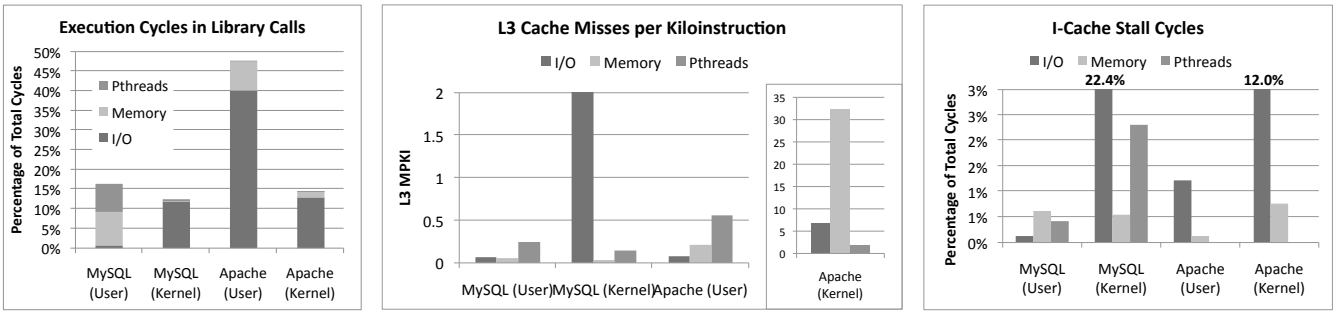
Experimental Setup To examine the effects of kernel code, we intercept and instrument functions in libc and pthreads. During calls to these libraries, we count cycles, L3 cache misses and instruction cache stalls in user space and kernel space separately. After collecting data, we aggregate the data from each function into three separate categories: I/O, memory and pthreads. I/O contains functions such as `read`, `write` and `printf` whereas memory has functions like `malloc` and `memset`. Pthreads contains all of the commonly-used synchronization functions. We look at two important systems applications, Apache and MySQL, using the workloads described in Section 4.

Results The results of this study are shown in Figures 7, 8 and 9. Figure 7 reveals potential inefficiencies. First, we observe that MySQL spends over 10% of its execution cycles in kernel I/O functions. Apache spends a comparable amount of time, but also spends a large amount of time in user I/O code. Overall, in fact, Apache spends the majority (about 61%) of its cycles in library code. Looking at cache information, Figure 7b shows that kernel I/O experiences far more cache misses per kiloinstruction than userspace code. The last chart, Figure 7c helps explains further, revealing extremely poor instruction cache utilization in kernel mode, especially in I/O functions.

Figures 8 and 9 show the CPI and last level cache misses for the worst performing functions in libc plus aggregates of userspace code, kernel code, library functions and normal program code. These data show that kernel code does not perform as well as userland code and that several functions perform very poorly, especially in terms of cache misses. In particular, the math function `floor` performs very poorly (due largely to cache misses) though it does not contain a kernel call. Fortunately, MySQL does not call it often (241 times compared with 4.4e8 times for memcpy). The infrequent calls and last level cache miss results suggest that that poor temporal locality and prefetching of mathematical constants or code in libm may be to blame for the poor performance.

Implications for Architects (#4,#5,#6)

The first important result from this data is that system applications have a lot of kernel interaction and their behavior in kernel regions is markedly different from userspace. As a result, userspace-only simulation misses potentially important information. Additionally, there are two key observa-



(a) Cycles in Library Functions (b) Last Level Cache Misses (c) ICache Stalls

Figure 7: Various user space and kernel space microarchitectural events occuring in categories of library functions. Comparing userspace to kernel, we see that kernel code behaves very differently than userspace code. Please note the different scale in (b) for Apache in kernel space.

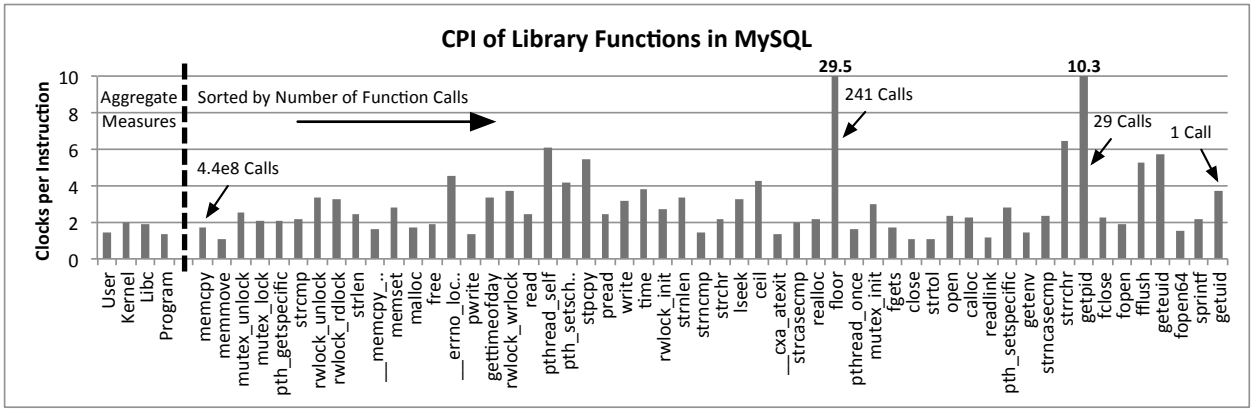


Figure 8: Cycles per instruction for various library functions executed by MySQL are listed here, sorted by number of calls. We see that in many cases, code in the dynamically linked library performs worse than typical program code. The same is true of kernel code to an even greater extent. Although performance is particularly poor for functions like `floor` and `getpid`, they are not called often and thus do not affect overall speed.

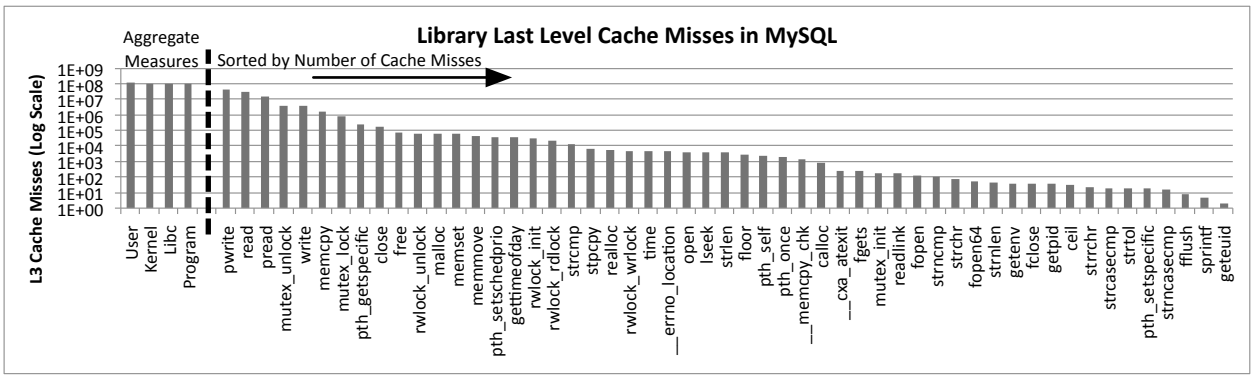
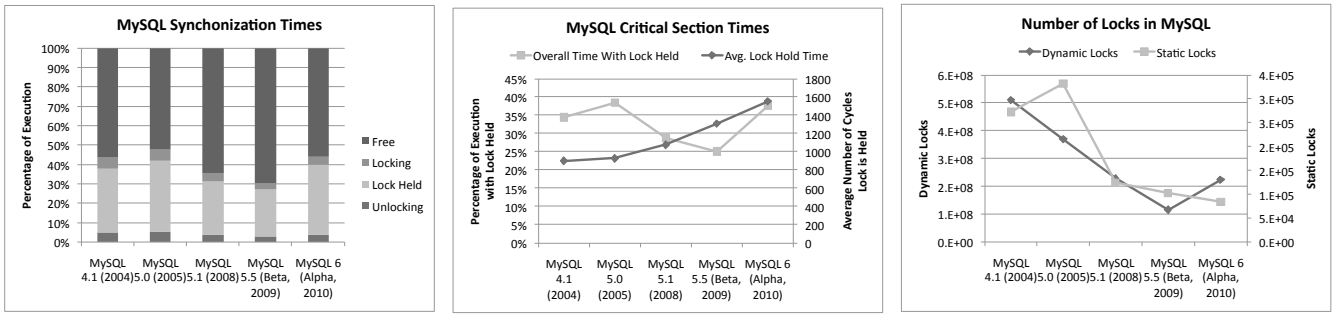


Figure 9: L3 cache misses in various dynamically linked library functions show that a handful of library functions account for a large portion of all the cache misses. Many of these functions result in kernel calls which suffer from abnormally high cache miss rates, as seen in Figure 7b. The MySQL benchmark executed for these data uses a database growing up to 45MB in size, relative to 8MB of CPU cache.



(a) Locking and Lock Held Times (b) Lock Held Times (c) Static and Dynamic Locks

Figure 10: A history of synchronization in MySQL. With the exception of MySQL 6 (a likely un-optimized alpha-quality version), time with locks held and time getting locks (contention and overhead) has decreased since version 4.1.

tions in the above data which indicate potential avenues for optimization:

 # 4: The Apache results show the importance of I/O optimization. Apache spends much time interacting with the kernel, incurring significant overheads. Hardware support to allow Apache (and similar programs) to circumvent the kernel to do its I/O could drastically decrease its latency and increase throughput.

 # 5: Poor instruction cache behavior in kernel mode may indicate that the processor is unable to prefetch kernel instructions before interrupts occur. It should be possible for a hardware prefetcher to determine the system call number and prefetch the necessary upcoming instruction code, avoiding I-Cache misses.

 # 6: Finally, this **LiMiT**-obtained data has identified several problem points in real applications with unscaled workloads. With **LiMiT**, a process that would have taken months using simulators took only 3 days. If microbenchmarks can be designed to capture these bottlenecks, they can be used in full system simulation. This style of combining **LiMiT**'s precise event counter approach with detailed simulation may be necessary for quantitative architecture research in the cloud era.

6. CASE STUDY C: LONGITUDINAL STUDY OF LOCKING BEHAVIOR IN MYSQL

Embarking on parallelization is often a risky investment with little guarantee of performance improvements due to the difficulties in writing multithreaded code. Many organizations that have legacy sequential codes are hesitant to invest in parallelization without quantitative models that can be used to predict return of investment on parallelization. **LiMiT** offers capabilities to build such a model.

In this case study, we use **LiMiT** to examine the benefits of adapting software to multicores over multiple versions spanning years. To examine software development progress, we examine several versions of MySQL, an extremely popular database management system. Gartner Group estimates that 50% of IT organizations had MySQL deployments in 2008, making MySQL a very common workload. As an open source product, we are also able to access its source code from many versions going back to 2004. Releases from 2004 on are beneficiaries of increased market penetration of multicore machines, increasing pressure on MySQL to use multithreading for performance.

Goals We will attempt to answer the following questions using behavioral information: (1) Has synchronization in MySQL changed through versions? (2) Has the amount of time in critical sections changed? We will use these questions to judge if MySQL developers have improved at multicore development since the widespread availability of multicore systems.

Necessity of LiMiT As in case study A, we are examining fine-grained program sections: lock acquires/releases and critical sections. To avoid perturbation, interference from multiple threads and error introduced by sampling, we require **LiMiT**'s low-overhead reads, process isolation and precision. Sampling is a poor option for the same reasons as given in case study A.

Experimental Setup To answer these questions, we intercept mysqld calls to the pthread library's locking routines to insert timing instrumentation. All versions of MySQL were compiled and executed on identical systems, so they all use the same, recent version of pthreads. As input, we run the "sql-bench" benchmark suite supplied with MySQL.

Results The results of this study are shown in Figure 10. They indicate that synchronization efficiency has increased since the 4.1 series, first introduced in 2004. Figure 10a examines overall times in synchronization and critical sections. Figure 10b rehashes the critical section results from the previous chart and overlays the average lock held time. Finally, Figure 10c examines the number of static and dynamic locks observed during execution. There are several interesting points to note:

Average Lock Held Times MySQL developers have decreased the total amount of time spent with locks held while simultaneously increasing the average amount of time each lock is held. This implies that the functionality of multiple critical sections has been combined. For low-contention critical sections, this increases overall efficiency by avoiding lock overheads.

Lock Granularity The number of static and dynamic locks have both decreased. This implies that – on av-

erage – lock granularity has increased. Although this could increase contention, it has not come at that cost, so this granularity shift has likely been carefully tuned.

Alpha Version MySQL 6, the alpha version, is an outlier with respect to recent versions. This is likely because it has not yet been optimized with respect to locking and new features have been implemented in overly conservative fashions.

To answer our initial questions, both synchronization overheads and critical section times have decreased over time. These performance improvements clearly show that developers have become more skilled, likely a result of multicore availability as parallel machines were not commonly available to hobbyist hackers before 2004.

Implication for Architects (#7): While this is primarily a software engineering/project management study – *a*nd to the best of our knowledge the first study to use precise performance counters for software engineering – there is a very important take away point here for computer architects: there is a potentially broader consumer base for on-chip performance counter data beyond computer architects, OS and compiler writers. Computer architects should take this into consideration when designing future hardware monitoring systems. Broadly, this means that monitors should be optimized not to capture just the common execution cases but also uncommon cases which are interest in domains such as software engineering and security.

7. HARDWARE ENHANCEMENTS FOR BETTER PRECISE PERFORMANCE COUNTING

Precise performance measurement does not appear to be an intended application for performance counter architectures today. Some modest modifications to existing performance monitoring hardware can reduce the complexity and overheads of precise counting with tools like **LiMiT**. The operations suggested below will reduce **LiMiT**'s read routine from five instructions down to one and reduce the overhead of frequent counter usage patterns. Such low overheads would encourage programs to self-monitor and adapt to changing conditions.

Enhancement #1: 64-bit Reads and Writes **LiMiT**'s overflow handling is necessitated by a lack of full 64-bit read and write support. With 31-bit counters, the counters can overflow every 0.72 seconds, but with 64-bit support they would require centuries to overflow. Until such simple support can be added **LiMiT** will have a vital role in low overhead precise performance measurement.

Enhancement #2: Destructive Reads When characterizing code segments, a difference in counts between two points in the program is often required. A destructive read instruction – one that zeros the counter after reading it – could eliminate the currently necessary subtraction in many cases when counters are used.

Enhancement #3: Combined Reads Currently, the x86 performance counter read instruction requires that the `%ecx` register contain the number of the counter to read. Were this integrated into the instruction as an immediate, another instruction would be eliminated.

A further proposal for hardware support is AMD's Lightweight Profiling [2]. Unfortunately, LWP is not available on existing processors, making **LiMiT** valuable for research today.

8. CONCLUSION

Our paper makes the following contributions: (1) We have described a lightweight, precise interface to performance counters on contemporary hardware. (2) We have conducted case studies to demonstrate the utility of precise monitoring to architects. Based on data collected with **LiMiT**, we offer new insights on program behavior which were not possible with existing tools. (3) Based on our experience with **LiMiT**, we suggest hardware support to decrease the cost of accesses to performance counters.

To continue having real world impact, architects must be engineers, designing machines to accelerate a wide variety of new applications and usage models. As scientists, architects also need to conduct rigorous, reproducible research studies. While this latter goal can be achieved with simulation technology available today, it has been challenging for simulators to keep pace with rapid changes in the software landscape. Tools such as **LiMiT** help architects keep pace with new software, potentially using the insights gained to develop fast, robust, representative microbenchmarks for simulation based studies.

As a demonstration of the usefulness of precise performance monitoring capabilities offered by **LiMiT**, we conducted three case studies on current web workloads. These studies lead us to the following conclusions:

- A new benchmark suite is recommended for research in computer architectures for the cloud era because traditional multithreaded benchmarks have different execution characteristics than multithreaded applications frequently used today.
- Web applications tend to have many very short critical sections which could be sped up with architectural support for lighter weight synchronization. Since the total overhead of lock acquisition and release is about 13% and 8% for Firefox and MySQL respectively, speedups in that range may be possible.
- Dynamically linked libraries and kernel code suffer from poor microarchitectural performance and also make up substantial portions of run time for system applications. Further research to enhance this performance could significantly accelerate web workloads.
- Performance counters have far wider applicability than just computer architecture (*e.g.,* software engineering) and architects designing performance counter systems should consider other applications.

These insights were made possible by precise, low-overhead performance monitoring capabilities provided by the **LiMiT** tool. These features allow monitoring of parallel programs more precisely than existing sampling based tools. In **LiMiT** we revisited and re-architected existing performance counter access methodologies (which had not been revised in the past decade). Specifically, we used novel kernel/user space cooperative techniques to allow user space readouts of performance counters. As a result, **LiMiT** is at least an order of magnitude faster than its existing state-of-the-art alternative, and reduces instrumented execution overheads significantly. In short, **LiMiT** can read virtualized counters in less than 12 nanoseconds, allowing precise measurements at finer granularities than have ever been studied.

Much of **LiMiT**'s implementation complexity and execution cost was due to suboptimal hardware support. **LiMiT** can be further optimized with minimal additional hardware support. Specifically, we suggest the following ISA changes for future architectures: (1) increasing the counter size to 64-bit and allowing full 64-bit reads and writes, (2) including a destructive read instruction and (3) integrating counter selection into the read instruction. These three simple modifications would drastically reduce complexity and allow single instruction readouts.

LiMiT is a significant step towards rapid, precise program characterization and is now available at http://castl.cs.columbia.edu/limit. We are planning integration with perf_event to provide **LiMiT**'s benefits to all Linux users.

9. ACKNOWLEDGMENTS

We thank Prof. Steve Blackburn, Prof. Mark Hill, Dr. Viji Srinivasan, Dr. Dick Sites, anonymous reviewers and members of the Computer Architecture and Security Technologies Lab (CASTL) at Columbia University for their feedback on this work. Research conducted at CASTL is funded by grants from DARPA, AFRL (FA8750-10-2-0253, FA9950-09-1-0389), the NSF CAREER program, gifts from Microsoft Research and Columbia University, and software donations from Synopsys and Wind River.

10. REFERENCES

[1] Linux kernel 2.6.32, `perf_event.h`.

[2] Amd64 technology lightweight profiling specification, revision 3.08, 2010.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. Dbmss on a modern processor: Where does time go? pages 266–277, 1999.

[4] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[5] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. *SIGPLAN Not.*, 27:223–233, September 1992.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[7] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[8] Joel S. Emer and Douglas W. Clark. A characterization of processor performance in the vax-11/780. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 301–310, New York, NY, USA, 1984. ACM.

[9] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. *SIGARCH Comput. Archit. News*, 38:362–370, June 2010.

[10] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance characterization of a quad pentium pro smp using oltp workloads. *SIGARCH Comput. Archit. News*, 26(3):15–26, 1998.

[11] Tao Li, Lizy Kurian John, Anand Sivasubramaniam, N. Vijaykrishnan, and Juan Rubio. Understanding and improving operating system effects in control flow prediction. *SIGPLAN Not.*, 37:68–80, October 2002.

[12] Shirley Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In Peter Sloot, Alfons Hoekstra, C. Tan, and Jack Dongarra, editors, *Computational Science – ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 904–912. Springer Berlin / Heidelberg, 2002.

[13] Oprofile. http://oprofile.sourceforge.net/.

[14] Perfmon2. http://perfmon2.sourceforge.net/.

[15] Rabbit, a performance counters library for intel/amd processors and linux. http://www.scl.ameslab.gov/Projects/Rabbit/.

[16] Dr. Richard Sites. Personal communications.

[17] Intel vtune. http://software.intel.com/en-us/intel-vtune/.

[18] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 16–16, 1996.