I don't think oprofile will be our solution based on this. It's more about continuous monitoring than benchmarking.

Interesting though:

- We use sampling rather than full instrumentation to reduce overhead in observing/measuring.
- Most whole-of-system profiling needs Root Access.
- General events for profiling include clock cycles, L1/2 cache misses and branch mispredicts.

this paper provides a precedent for using profiling to compare hardware implementations for the same application.

↳ good justification for our tentative approach to PIUMA.

↳ I haven't seen yet what a standard "profile" for our graph algo's is. I think that this is a gap we could dive into! ☆

Future work: viz of Memory Access patterns.

Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers

Google-Wide Profiling (GWP), a continuous profiling infrastructure for data centers, provides performance insights for cloud applications. With negligible overhead, GWP provides stable, accurate profiles and a datacenter-scale tool for traditional performance analyses. Furthermore, GWP introduces novel applications of its profiles, such as application-platform affinity measurements and identification of platform-specific, microarchitectural peculiarities.

Gang Ren
Eric Tune
Tipp Moseley
Yixin Shi
Silvius Rus
Robert Hundt
Google

• • • • • • As cloud-based computing grows in pervasiveness and scale, understanding datacenter applications' performance and utilization characteristics is critically important, because even minor performance improvements translate into huge cost savings. Traditional performance analysis, which typically needs to isolate benchmarks, can be too complicated or even impossible with modern datacenter applications. It's easier and more representative to monitor datacenter applications running on live traffic. However, application owners won't tolerate latency degradations of more than a few percent, so these tools must be nonintrusive and have minimal overhead. As with all profiling tools, observer distortion must be minimized to enable meaningful analysis. (For additional information on related techniques, see the "Profiling: From single systems to data centers" sidebar.)

Sampling-based tools can bring overhead and distortion to acceptable levels, so they're uniquely qualified for performance monitoring in the data center. Traditionally, sampling tools run on a single machine, monitoring specific processes or the system as a whole. Profiling can begin on demand to analyze a performance problem, or it can run continuously.[1]

Sampling is a technique to collect observations on performance without significantly degrading it.

Google-Wide Profiling can be theoretically viewed as an extension of the Digital Continuous Profiling Infrastructure (DCPI)[1] to data centers. GWP is a continuous profiling infrastructure; it samples across machines in multiple data centers and collects various events—such as stack traces, hardware events, lock contention profiles, heap profiles, and kernel events—allowing cross-correlation with job scheduling data, application-specific data, and other information from the data centers.

GWP collects daily profiles from several thousand applications running on thousands of servers, and the compressed profile

0272-1732/10/$26.00 © 2010 IEEE

Published by the IEEE Computer Society

Profiling: From single systems to data centers

From gprof[1] to Intel VTune (http://software.intel.com/en-us/intel-vtune), profiling has long been standard in the development process. Most profiling tools focus on a single execution of a single program. As computing systems have evolved, understanding the bigger picture across multiple machines has become increasingly important.

Continuous, always-on profiling became more important with Morph and DCPI for Digital UNIX. Morph uses low overhead system-wide sampling (approximately 0.3 percent) and binary rewriting to continuously evolve programs to adapt to their host architectures.[2] DCPI gathers much more robust profiles, such as precise stall times and causes, and focuses on reporting information to users.[3]

OProfile, a DCPI-inspired tool, collects and reports data much in the same way for a plethora of architectures, though it offers less sophisticated analysis. GWP uses OProfile (http://oprofile.sourceforge.net) as a profile source. High-performance computing (HPC) shares many profiling challenges with cloud computing, because profilers must gather representative samples with minimal overhead over thousands of nodes.

Oprofile used for profiling.

Cloud computing has additional challenges—vastly disparate applications, workloads, and machine configurations. As a result, different profiling strategies exist for HPC and cloud computing. HPCToolkit uses lightweight sampling to collect call stacks from optimized programs and compares profiles to identify scaling issues as parallelism increases.[4] OpenSpeedShop (www.openspeedshop.org) provides similar functionality. Similarly, Upshot[5] and Jumpshot[6] can analyze traces (such as MPI calls) from parallel programs but aren't suitable for continuous profiling.

Other profiling options.

References

1. S.L. Graham, P.B. Kessler, and M.K. Mckusick, ''Gprof: A Call Graph Execution Profiler,'' *Proc. Symp. Compiler Construction* (CC 82), ACM Press, 1982, pp. 120-126.
2. X. Zhang et al., ''System Support for Automatic Profiling and Optimization,'' *Proc. 16th ACM Symp. Operating Systems Principles* (SOSP 97), ACM Press, 1997, pp. 15-26.
3. J.M. Anderson et al., ''Continuous Profiling: Where Have All the Cycles Gone?'' *Proc. 16th ACM Symp. Operating Systems Principles* (SOSP 97), ACM Press, 1997, pp. 357-390.
4. N.R. Tallent et al., ''Diagnosing Performance Bottlenecks in Emerging Petascale Applications,'' *Proc. Conf. High Performance Computing Networking, Storage and Analysis* (SC 09), ACM Press, 2009, no. 51.
5. V. Herrarte and E. Lusk, *Studying Parallel Program Behavior with Upshot,* tech. report, Argonne National Laboratory, 1991.
6. O. Zaki et al., ''Toward Scalable Performance Visualization with Jumpshot,'' *Int'l J. High Performance Computing Applications,* vol. 13, no. 3, 1999, pp. 277-288.

database grows by several Gbytes every day. Profiling at this scale presents significant challenges that don't exist for a single machine. Verifying that the profiles are correct is important and challenging because the workloads are dynamic. Managing profiling overhead becomes far more important as well, as any unnecessary profiling overhead can cost millions of dollars in additional resources. Finally, making the profile data universally accessible is an additional challenge. GWP is also a cloud application, with its own scalability and performance issues.

With this volume of data, we can answer typical performance questions about datacenter applications, including the following:

- What are the hottest processes, routines, or code regions?
- How does performance differ across software versions?
- Which locks are most contended?
- Which processes are memory hogs?
- Does a particular memory allocation scheme benefit a particular class of applications?
- What is the cycles per instruction (CPI) for applications across platforms?

Additionally, we can derive higher-level data to more complex but interesting questions, such as which compilers were used for applications in the fleet, whether there are more 32-bit or 64-bit applications running, and how much utilization is being lost by suboptimal job scheduling.

Infrastructure

Figure 1 provides an overview of the entire GWP system.

Collector

GWP samples in two dimensions. At any moment, profiling occurs only on a small subset of all machines in the fleet, and event-based sampling is used at the machine level. Sampling in only one dimension would be unsatisfactory; if event-based profiling were active on every machine all the time, at a normal event-sampling rate, we would be using too many resources across the fleet. Alternatively, if the event-sampling rate is too low, profiles become too sparse to drill down to the individual machine level. For each event type, we choose a sampling rate high enough to provide meaningful machine-level data while still minimizing the distortion caused by the profiling on critical applications. The system has been actively profiling nearly all machines at Google for several years with only rare complaints of system interference.

A central machine database manages all machines in the fleet and lists every machine's name and basic hardware characteristics. The GWP profile collector periodically gets a list of all machines from that database and selects a random sample of machines from that pool. The collector then remotely activates profiling on the selected machines and retrieves the results. It retrieves different types of sampled profiles sequentially or concurrently, depending on the machine and event type. For example, the collector might gather hardware performance counters for several seconds each, then move on to profiling for lock contention or memory allocation. It takes a few minutes to gather profiles for a specific machine.

For robustness, the GWP collector is a distributed service. It helps improve availability and reduce additional variation from the collector itself. To minimize distortion on the machines and the services running on them, the collector monitors error conditions and ceases profiling if the failure rate reaches a predefined threshold. Aside from the collector, we monitor all other GWP components to ensure an always-on service to users.

On the top of the two-dimensional sampling approach, we apply several techniques to further reduce the overhead. First, we measure the event-based profiling overhead on a set of benchmark applications and then conservatively set the maximum rates to ensure the overhead is always less than a few percent. Second, we don't collect whole call stacks for the machine-wide profiles to avoid the high overhead associated with unwinding (but we collect call stacks for most server profiles at lower sampling frequencies). Finally, we save the profile and metadata in their raw format and perform symbolization on a separate set of machines. As a result, the aggregated profiling overhead is negligible—less than 0.01 percent. At the same time, the derived profiles are still meaningful, as we show in the "Reliability analysis" section.

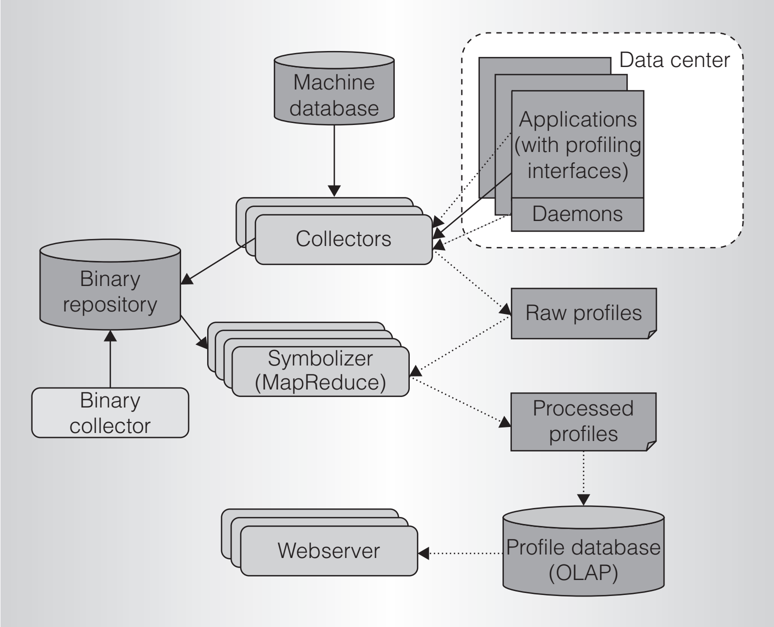


Figure 1. An overview of the Google-Wide Profiling (GWP) infrastructure. The whole system consists of collector, symbolizer, profile database, Web server, and other components.

Profiles and profiling interfaces

GWP collects two categories of profiles: whole-machine and per-process. Whole-machine profiles capture all activities happening on the machine, including user applications, the kernel, kernel modules, daemons, and other background jobs. The whole-machine profiles include hardware performance monitoring (HPM) event profiles, kernel event traces, and power measurements.

Users without root access cannot directly invoke most of the whole-machine profiling systems, so we deploy lightweight daemons on every machine to let remote users (such as GWP collectors) access those profiles. The daemons act as gate keepers to control access, enforce sampling rate limits, and collect system variables that must be synchronized with the profiles.

Need Root access for whole-of-machine profiling tasks

We use OProfile (http://oprofile.sourceforge.net) to collect HPM event profiles. OProfile is a system-wide profiler that uses HPM to generate event-based samples for all running binaries at low overhead. To hide the heterogeneity of events between architectures, we define some generic HPM events on top of the platform-specific events, using an approach similar to PAPI.[2] The most commonly used generic events are CPU cycles, retired instructions, L1 and L2 cache misses, and branch mispredictions. We also provide access to some architecture-specific events. Although the aggregated profiles for those events are biased to specific architectures, they provide useful information for machine-specific scenarios.

Generic events for profiling.
(1) CPU cycles
(2) L1/L2 Cache Misses
(3) Branch Mispredictions

In addition to whole-machine profiles, we collect various types of profiles from most applications running on a machine using the Google Performance Tools (http://code.google.com/p/google-perftools). Most applications include a common library that enables process-wide stacktrace-attributed profiling mechanisms for heap allocation, lock contention, wall time and CPU time, and other performance metrics. The common library includes a simple HTTP server linked with handlers for each type of profiler. A handler accepts requests from remote users, activates profiling (if it's not already active), and then sends the profile data back.

The GWP collector learns from the cluster-wide job management system what applications are running on a machine and on which port each can be contacted for remote profiling invocation. A machine lacking remote profiling support has some programs, which the per-process profiling doesn't capture. However, these are few; comparison with system-wide profiles shows that remote per-process profiling captures the vast majority of Google's programs. Most programs' users have found profiling useful or unobtrusive enough to leave them enabled.

Together with profiles, GWP collects other information about the target machine and applications. Some of the extra information is needed to postprocess the collected profiles, such as a unique identifier for each running binary that can be correlated across machines with unstripped versions for offline symbolization. The rest are mainly used to tag the profiles so that we later correlate the profiles with job, machine, or datacenter attributes.

Symbolization and binary storage

After collection, the Google File System (GFS) stores the profiles.[3] To provide meaningful information, the profiles must correlate to source code. However, to save network bandwidth and disk space, applications are usually deployed into data centers without any debug or symbolic information, which can make source correlation impossible. Furthermore, several applications, such as Java and QEMU, dynamically generate and execute code. The code is not available offline and can therefore no longer be symbolized. The symbolizer must also symbolize operating system kernels and kernel loadable modules.

Therefore, the symbolization process becomes surprisingly complicated, although it's usually trivial for single-machine profiling. Various strategies exist to obtain binaries with debug information. For example, we could try to recompile all sampled applications at specific source milestones. However, it's too resource-intensive and sometimes impossible for applications whose source isn't readily available. An alternative is to persistently store binaries that contain debug information before they're stripped.

Currently, GWP stores unstripped binaries in a global repository, which other services use to symbolize stack traces for automated failure reporting. Since the binaries are quite large and many unique binaries exist, symbolization for a single day of profiles would take weeks if run sequentially. To reduce the result latency, we distribute symbolization across a few hundred machines using MapReduce.[4]

Profile storage

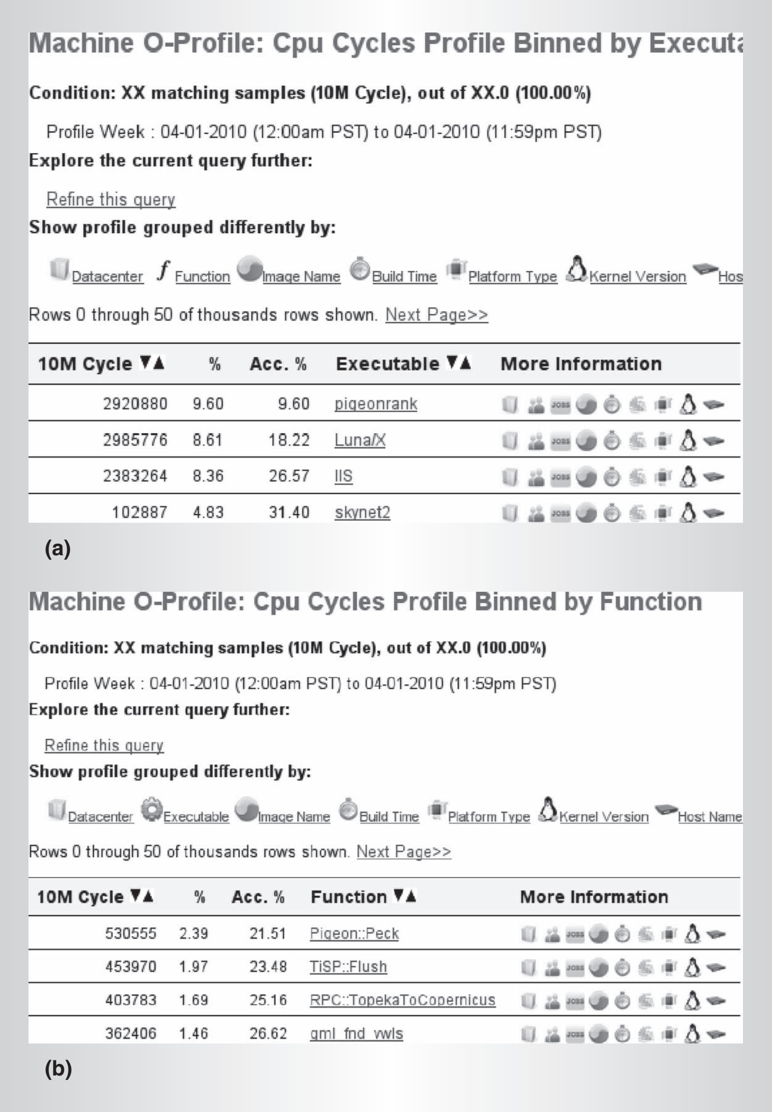
Over the past years, GWP has amassed several terabytes of historical performance data. GFS archives the entire performance logs and corresponding binaries. To make the data useful and accessible, we load the samples into a read-only dimensional database that is distributed across hundreds of machines. That service is accessible to all users for ad hoc queries and to systems for automated analyses.

The database supports a subset of SQL-like semantics. Although the dimensional database is well suited to perform queries that aggregate over the large data set, some individual queries can take tens of seconds to complete. Fortunately, most queries are seen frequently, so the profile server uses aggressive caching to hide the database latency.

User interfaces

For most users, GWP deploys a webserver to provide a user interface on top of the profile database. This makes it easy to access profile data and construct ad hoc queries for the traditional use of application profiles (with additional freedom to filter, group, and aggregate profiles differently).

Query view. Several visual interfaces retrieve information from the profile database, and all are navigable from a web browser. The primary interface (see Figure 2) displays the result entries, such as functions or executables, that match the desired query parameters. This page supplies links that let users refine the query to more specific data. For example, the user can restrict the query to only report samples for a specific executable collected within a desired time period. Additionally, the user can modify or refine any of the parameters to the current query to create a custom profile view. The GWP homepage has links to display the top results, Google-wide, for each performance metric.



Machine O-Profile: Cpu Cycles Profile Binned by Executa

Condition: XX matching samples (10M Cycle), out of XX.0 (100.00%)

Profile Week : 04-01-2010 (12:00am PST) to 04-01-2010 (11:59pm PST)

Explore the current query further:

Refine this query

Show profile grouped differently by:

Datacenter Function Image Name Build Time Platform Type Kernel Version Hos

Rows 0 through 50 of thousands rows shown. Next Page>>

10M Cycle ▼▲	%	Acc. %	Executable ▼▲	More Information
2920880	9.60	9.60	pigeonrank	
2985776	8.61	18.22	Luna/X	
2383264	8.36	26.57	IIS	
102887	4.83	31.40	skynet2	

(a)

Machine O-Profile: Cpu Cycles Profile Binned by Function

Condition: XX matching samples (10M Cycle), out of XX.0 (100.00%)

Profile Week : 04-01-2010 (12:00am PST) to 04-01-2010 (11:59pm PST)

Explore the current query further:

Refine this query

Show profile grouped differently by:

Datacenter Executable Image Name Build Time Platform Type Kernel Version Host Name

Rows 0 through 50 of thousands rows shown. Next Page>>

10M Cycle ▼▲	%	Acc. %	Function ▼▲	More Information
530555	2.39	21.51	Pigeon::Peck	
453970	1.97	23.48	TiSP::Flush	
403783	1.69	25.16	RPC::TopekaToCopernicus	
362406	1.46	26.62	gml_fnd_vwls	

(b)

Figure 2. An example query view: an application-level profile (a) and a function-level profile (b).

Call graph view. For most server profile samples, the profilers collect full call stacks with each sample. Call stacks are aggregated to produce complete dynamic call graphs for a given profile. Figure 3 shows an example call graph. Each node displays the function name and its percentage of samples, and the nodes are shaded based on this percentage. The call graph is also displayed through the web browser, via a Graphviz plug-in.

Source annotation. The query and call graph views are useful in directing users to specific functions of interest. From there, GWP provides a source annotation view that presents the original source file with a header describing overall profile information about the file and a histogram bar showing the relative hotness of each source file line. Because different versions of each file can exist in source repositories and branches, we retrieve a hash signature from the repository for each source file and aggregate samples on files with identical signatures.

Profile data API. In addition to the webserver, we offer a data-access API to read profiles directly from the database. It's more suitable for automated analyses that must process a large amount of profile

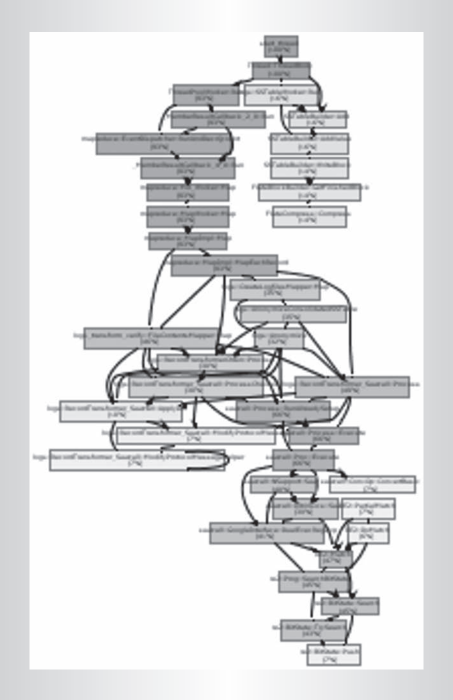


Figure 3. An example dynamic call graph. Function names are intentionally blurred.

data (such as reliability studies) offline. We store both raw profiles and symbolized profiles in ProtocolBuffer formats (http://code.google.com/apis/protocolbuffers). Advanced users can access and reprocess them using their preferred programming language.

Application-specific profiling

Although the default sampling rate is high enough to derive top-level profiles with high confidence, GWP might not collect enough samples for applications that consume relatively few cycles Google-wide. Increasing the overall sampling rate to cover those profiles is too expensive because they're usually sparse.

Therefore, we provide an extension to GWP for application-specific profiling on the cloud. The machine pool for application-specific profiling is usually much smaller than GWP, so we can achieve a high sampling rate on those machines for the specific application. Several application teams at Google use application-specific profiling to continuously monitor their applications running on the fleet.

Application-specific profiling is generic and can target any specific set of machines. For example, we can use it to profile a set of machines deployed with the newest kernel version. We can also limit the profiling duration to a small time period, such as the application's running time. It's useful for batch jobs running on data centers, such as MapReduce, because it facilitates collecting, aggregating, and exploring profiles collected from hundreds or thousands of their workers.

Reliability analysis

To conduct continuous profiling on datacenter machines serving real traffic, extremely low overhead is paramount, so we sample in both time and machine dimensions. Sampling introduces variation, so we must measure and understand how sampling affects the profiles' quality. But the nature of datacenter workloads makes this difficult; their behavior is continually changing. There's no direct way to measure the datacenter applications' profiles' representativeness. Instead, we use two indirect methods to evaluate their soundness.

First, we study the stability of aggregated profiles themselves using several different metrics. Second, we correlate profiles with the performance data from other sources to cross-validate both.

Stability of profiles

We use a single metric, entropy, to measure a given profile's variation. In short, entropy is a measure of the uncertainty associated with a random variable, which in this case is profile samples. The entropy H of a profile is defined as

$$H(W) = -\sum_{i=1}^{n} p(x_i)\log(p(x_i))$$

where n is the total number of entries in the profile and $p(x)$ is the fraction of profile samples on the entry x.[5]

In general, a high entropy implies a flat profile with many samples. A low entropy usually results from a small number of samples or most samples being concentrated to few entries. We're not concerned with entropy itself. Because entropy is like the signature in a profile, measuring the inherent

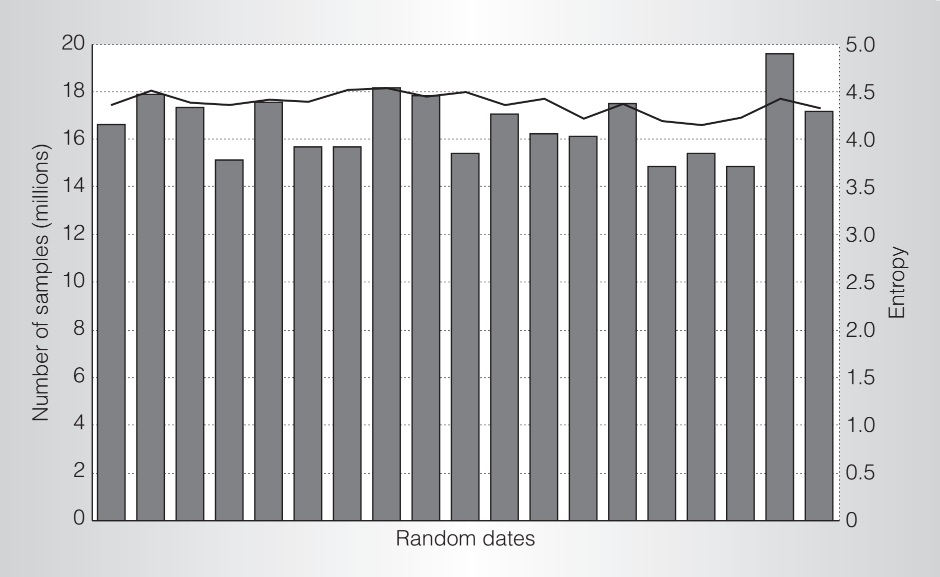


Figure 4. The number of samples and the entropy of daily application-level profiles. The primary *y*-axis (bars) is the total number of profile samples. The secondary *y*-axis (line) is the entropy of the daily application-level profile.

variation, it should be stable between representative profiles.

Entropy doesn't account for differences between entry names. For example, a function profile with *x* percent on *foo* and *y* percent on *bar* has the same entropy as a profile with *y* percent on *foo* and *x* percent on *bar*. So, when we need to identify the changes on the same entries between profiles, we calculate the Manhattan distance of two profiles by adding the absolute percentage differences between the top *k* entries, defined as

$$M(X, Y) = \sum_{i=1}^{k} \left| p_x(x_i) - p_y(x_i) \right|$$

where *X* and *Y* are two profiles, *k* is the number of top entries to count, and $p_y(x_i)$ is 0 when x_i is not in *Y*. Essentially, the Manhattan distance is a simplified version of relative entropy between two profiles.

Profiles' entropy. First, we compare the entropies of application-level profiles where samples are broken down on individual applications. Figure 2a shows an example of such an application-level profile.

Figure 4 shows daily application-level profiles' entropies for a series of dates, together with the total number of profile samples collected for each date. Unless specified, we used CPU cycles in the study, and our conclusions also apply to the other types. As the graph shows, the entropy of daily application-level profiles is stable between dates, and it usually falls into a small interval. The correlation between the number of samples and the profile's entropy is loose. Once the number of samples reaches some threshold, it doesn't necessarily lead to a lower entropy, partly because GWP sometimes samples more machines than necessary for daily application-level profiles. This is because users frequently must drill down to specific profiles with additional filters on certain tags, such as application names, which are a small fraction of all profiles collected.

We can conduct similar analysis on an application's function-level profile (for example, the application in Figure 2b). The result, shown in Figure 5a, is from an application whose workload is fairly stable when aggregating from many clients. Its entropy is actually more stable. It's interesting to analyze how entropy changes among machines for an application's function-level profiles. Unlike the aggregated profiles across machines, an application's per-machine profiles can vary greatly in terms

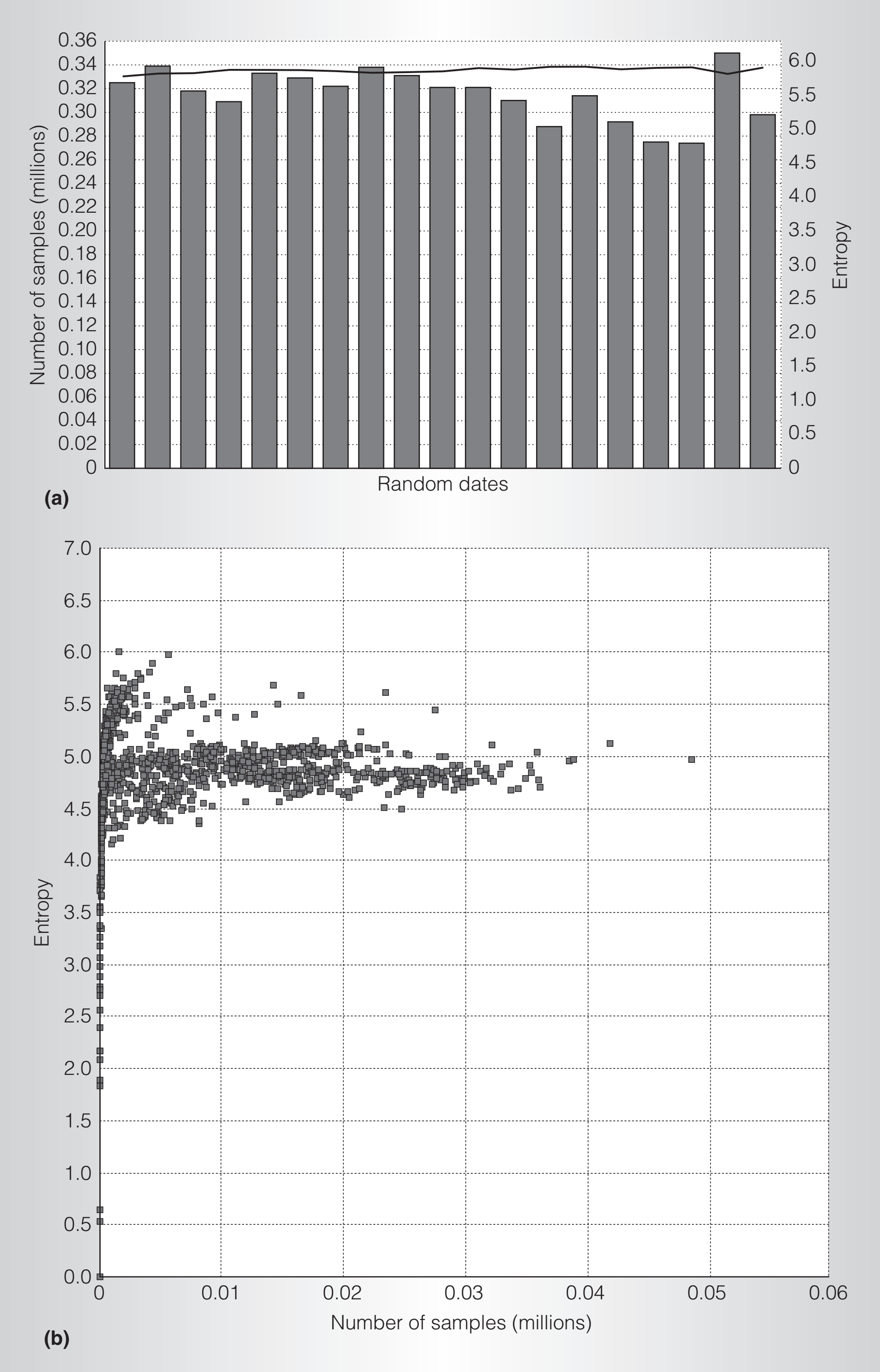


Figure 5. Function-level profiles. The number of samples and the entropy for a single application (a). The correlation between the number of samples and the entropy for all per-machine profiles (b).

of number of samples. Figure 5b plots the relationship between the number of samples per machine and the entropy of function-level profiles. As expected, when the total number of samples is small, the profile's entropy is also small (limited by the maximum possible uncertainty). But once the threshold is reached at the maximum number of samples, the entropy becomes stable. We can observe two clusters from the graph; some entropies are concentrated between 5.5 and 6, and the others fall between 4.5 and 5. The application's two behavioral states can explain the two clusters. We've seen various clustering patterns on different applications.

The Manhattan distance between profiles. We use the Manhattan distance to study the variation between profiles considering the changes on entry name, where smaller distance implies less variation. Figure 6a illustrates the Manhattan distance between the daily application-level profiles for a series of dates. The results from the Manhattan distances for both application-level and function-level profiles are similar to the results with entropy.

In Figure 6b, we plot the Manhattan distance for several profile types, leading to two observations:

- In general, memory and thread profiles have smaller distances, and their variations appear less correlated with the other profiles.
- Server CPU time profiles correlate with HPM profiles of cycles and instructions in terms of variations, which could imply that those variations resulted naturally from external reasons, such as workload changes.

To further understand the correlation between the Manhattan distance and the number of samples, we randomly pick a subset of machines from a specific machine set and then compute the Manhattan distance of the selected subset's profile against the whole set's profile. We could use a power function's trend line to capture the change in the Manhattan distance over the number of samples. The trend line roughly approximates a square root relationship between the distance and the number of samples,

$$M(X) = C/\sqrt{N(X)}$$

where $N(X)$ is the total number of samples in a profile and C is a constant that depends on the profile type.

Derived metrics. We can also indirectly evaluate the profiles' stability by computing some derived metrics from multiple profiles. For example, we can derive CPI from HPM profiles containing cycles and retired instructions. Figure 7 shows that the derived CPI is stable across dates. Not surprisingly, the daily aggregated profiles' CPI falls into a small interval between 1.7 and 1.8 for those days.

Comparing with other sources

Beyond measuring profiles' stability across dates, we also cross-validate the profiles with performance and utilization data from other Google sources. One example is the utilization data that the data center's monitoring system collects. Unlike GWP, the monitoring system collects data from all machines in the data center but at a coarser granularity, such as overall CPU utilization. Its CPU utilization data, in terms of core-seconds, matches the measurement from GWP's CPU cycles profile with the following formula:

$$CoreSeconds = Cycles * SamplingRatemachine * SamplingPeriod/CPUFrequencyaverage$$

Profile uses

In each profile, GWP records the samples of interesting events and a vector of associated information. GWP collects roughly a dozen events, such as CPU cycles, retired instructions, L1 and L2 cache misses, branch mispredictions, heap memory allocations, and lock contention time. The sample definition varies depending on the event type—it can be CPU cycles or cache misses, bytes allocated, or the sampled thread's locking time. Note that the sample must be numeric and capable of aggregation. The associated vector contains information such as application name, function name,

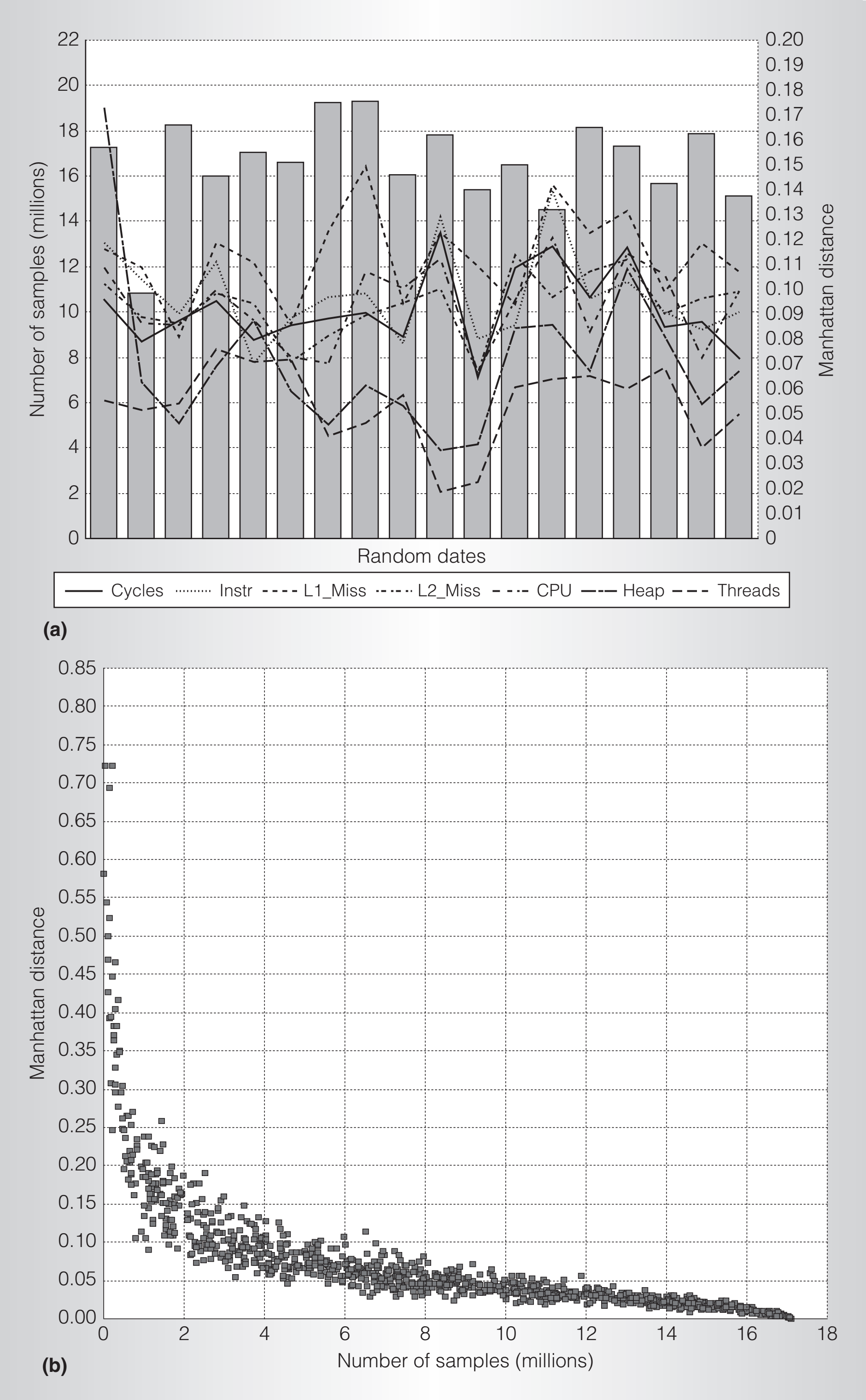


Figure 6. The Manhattan distance between daily application-level profiles for various profile types (a). The correlation between the number of samples and the Manhattan distance of profiles (b).

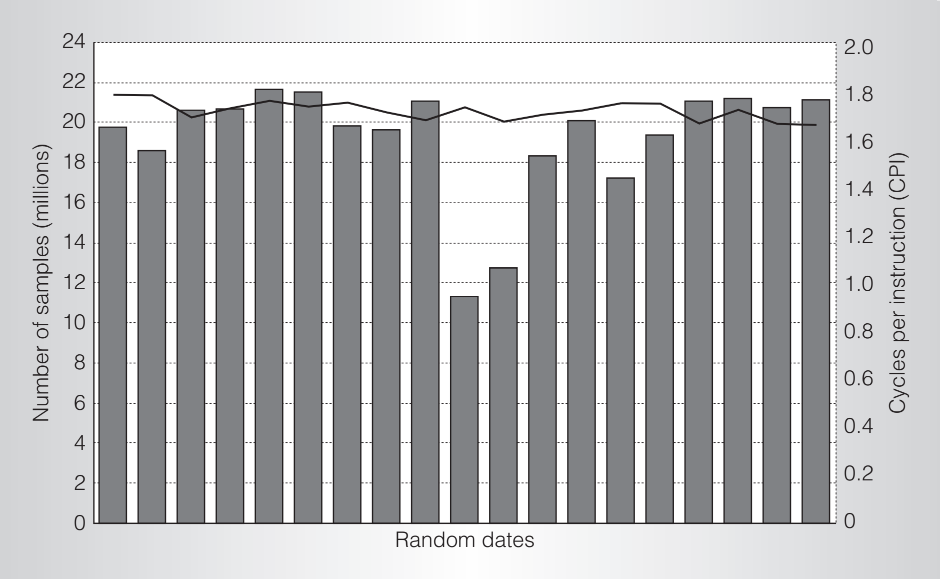


Figure 7. The correlation between the number of samples and derived cycles per instruction (CPI).

platform, compiler version, image name, data center, kernel information, build revision, and builder's name. Assuming that the vector contains *m* elements, we can represent a record GWP collected as a tuple <event, sample counter, *m*-dimension vector>.

When aggregating, GWP lets users choose *k* keys from the *m* dimensions and groups the samples by the keys. Basically, it filters the samples by imposing one or more restrictions on the rest of the dimensions (*m*—*k*) and then projects the samples into *k* key dimensions. GWP finally displays the sorted results to users, delivering answers to various performance queries with high confidence. Although not every query makes sense in practice, even a small subset of them are demonstrably informative in identifying performance issues and providing insights into computing resources in the cloud.

Cloud applications' performance

GWP profiles provide performance insights for cloud applications. Users can see how cloud applications are actually consuming machine resources and how the picture evolves over time. For example, Figure 2 shows cycles distributed over top executables and functions, which is useful for many aspects of designing, building, maintaining, and operating data centers. Infrastructure teams can see the big picture of how their software stacks are being used, aggregated across every application. This helps identify performance-critical components, create representative benchmark suites, and prioritize performance efforts.

At the same time, an application team can use GWP as the first stop for the application's profiles. As an always-on profiling service, GWP collects a representative sample of the application's running instances over time. Application developers often are surprised by application's profiles when browsing GWP results. For example, Google's speech-recognition team quickly optimized a previously unknown hot function that they couldn't have easily located without the aggregated GWP results. Application teams also use GWP profiles to design, evaluate, and calibrate their load tests.

Finding the hottest shared code. Shared code is remarkably abundant. Profiling each program independently might not identify hot shared code if it's not hot in any single application, but GWP can identify routines that don't account for a significant portion

Table 1. Platform affinity example.

Random assignment of instructions (CPI in brackets)		
	Platform 1	Platform 2
NumCrunch	100 (1)	100 (1)
MemBench	100 (1)	100 (2)
Total cycles	200	300
Optimal assignment of instructions		
	Platform 1	Platform 2
NumCrunch	0 (1)	200 (1)
MemBench	200 (1)	0 (2)
Total cycles	200	200

of any single application but consume the most cycles overall. For example, the GWP profiles revealed that the zlib library (www.zlib.net) accounted for nearly 5 percent of all CPU cycles consumed. That motivated an effort to optimize zlib routines and evaluate compression alternatives. In fact, some users have used GWP numbers to calculate the estimated savings for performance tuning efforts on shared functions. Not surprisingly, given the Google fleet's scale, a single-percent improvement on a core routine could potentially save significant money per year. Unsurprisingly, the new informal metric, "dollar amount per performance change," has become popular among Google engineers. We're considering providing a new metric, "dollar per source line," in annotated source views.

At the same time, some users have used GWP profiles as a source of coverage information to assess the feasibility of deprecation and to uncover users of library functions that are to be deprecated. Due to the profiles' dynamic nature, the users might miss less common clients, but the biggest, most important callers are easy to find.

Evaluating hardware features. The low-level information GWP provides about how CPU cycles (and other machine resources) are spent is also used for early evaluation of new hardware features that datacenter operators might want to introduce. One interesting example has been to evaluate whether it would be beneficial to use a special coprocessor to accelerate floating-point computation by looking at its percentage of all Google's computations. As another example, GWP profiles can identify the applications running on old hardware configurations and evaluate whether they should be retired for efficiency.

Precedent for using profile information to evaluate the likely consequence of using new hardware.
→ Analogy for eval of PIUMA
↳ this compare related profiles of what the graph apps do to memory? → possible GAP

Optimizing for application affinities

Some applications run better on a particular hardware platform due to sensitivity to architectural details, such as processor microarchitecture or cache size. It's generally very hard or impossible to predict which application will fare best on which platform. Instead, we measure an efficiency metric, CPI, for each application and platform combination. We can then improve job scheduling so that applications are scheduled on platforms where they do best, subject to availability. The example in Table 1 shows how the total number of cycles needed to run a fixed number of instructions on a fixed machine capacity drops from 500 to 400 using preferential scheduling. Specifically, although the application NumCrunch runs just as well on Platform1 as on Platform2, application MemBench does poorly on Platform2 because of the smaller L2 cache. Thus, the scheduler should give MemBench preference to Platform1.

Hard to predict performance across multi architectures

The overall optimization process has several steps. First, we derive cycle and instruction samples from GWP profiles. Then, we compute an improved assignment table by moving instructions away from application-platform combinations with the worst relative efficiency. We use cycle and instruction samples over a fixed period of time, aggregated per job and platform. We then compute and normalize CPI by clock rate. We can formulate finding the optimal assignment as a linear programming problem. The one unknown is $Load_{ij}$—the number of instruction samples of application j on platform i.

The constants are:

- CPI_{ij}, the measured CPI of application j on platform i.
- $TotalLoad_j$, the total measured number of instruction samples of application j.
- $Capacity_i$, the total capacity for platform i, measured as total number of cycle samples for platform i.

The equation is:

$$\text{Minimize} \sum_{i,j} CPI_{ij} * Load_{ij}$$

$$\text{where} \sum_{j} Load_{ij} = TotalLoad_j$$

$$\text{and} \sum_{j} CPI_{ij} * Load_{ij} \leq Capacity_i$$

We use a simulated annealing solver that approximates the optimal solution in seconds for workloads of around 100 jobs running on thousands of machines of four different platforms over one month. Although application developers already mapped major applications to their best platform through manual assignment, we've measured 10 to 15 percent potential improvement in most cases where many jobs run on multiple platforms. Similarly, users can use GWP data to identify how to colocate multiple applications on a single machine to achieve the best throughput.[6]

Datacenter performance monitoring

GWP users can also use GWP queries with computing-resource–related keys, such as data center, platform, compiler, or builder, for auditing purposes. For example, when rolling out a new compiler, users inspect the versions that running applications are actually compiled with. Users can easily measure such transitions from time-based profiles. Similarly, a user can measure how soon a new hardware platform becomes active or how quickly old ones are retired. This also applies to new versions of applications being rolled out. Grouping by data center, GWP displays how the applications, CPU cycles, and machine types are distributed in different locations.

Beyond providing profile snapshots, GWP can monitor the changes between chosen profiles from two queries. The two profiles should be similar in that they must have the identical keys and events, but different in one or more other dimensions. For applications, GWP usually focuses on monitoring function profiles. This is useful for two reasons:

- performance optimization normally starts at the function level, and
- function samples can reconstruct the entire call graph, showing users how the CPU cycles (and other events) are distributed in the program.

A dramatic change to hot functions in the daily profiles could trigger a finer-grained comparison, which eventually points blame to the source code revision number, compiler, or data center.

Feedback-directed optimization

Sampled profiles can also be used for feedback-directed compiler optimization (FDO), outlined in work by Chen et al.[7] GWP collects such sampled profiles and offers a mechanism to extract profiles for a particular binary in a format that the compiler understands. This profile will be higher quality than any profile derived from test inputs because it was derived from running on live data. Furthermore, as long as developers make no changes in critical program sections, we can use an aged profile to improve a freshly released binary's performance. Many Web companies have release cycles of two weeks or less, so this approach works well in practice.

Similar to load test calibration, users also use GWP for quality assurance for profiles generated from static benchmarks. Benchmarks can represent some codes well, but not others, and users use GWP to identify which codes are ill-suited for FDO compilation.

Finally, users use GWP to estimate the quality of HPM events and microarchitectural features, such as cache latencies of various incarnations of processors with the same instruction set architecture (ISA). For example, if the same application runs on two platforms, we can compare the HPM counters and identify the relevant differences.

Besides adding more types of performance events to collect, we're now exploring more directions for using GWP profiles. These include not only user-interface enhancement but also advanced data-mining techniques to detect interesting patterns in profiles. It's also interesting to mesh-up GWP profiles with performance data from other sources to address complex performance problems in datacenter applications. MICRO

References

1. J.M. Anderson et al., “Continuous Profiling: Where Have All the Cycles Gone?” *Proc. 16th ACM Symp. Operating Systems Principles* (SOSP 97), ACM Press, 1997, pp. 357-390.
2. J. Dongarra et al., “Using PAPI for Hardware Performance Monitoring on Linux Systems,” *Conf. Linux Clusters: The HPC Revolution,* 2001.
3. S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” *Proc. 19th ACM Symp. Operating Systems Principles* (SOSP 03), ACM Press, 2003, pp. 29-43.
4. J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Proc. 6th Conf. Symp. Operating System Design and Implementation* (OSDI 04), Usenix Assoc., 2004, pp. 137-150.
5. S. Savari and C. Young, “Comparing and Combining Profiles,” *Proc. Workshop on Feedback-Directed Optimization* (FDO 02), ACM Press, 2000, pp. 50-62.
6. J. Mars and R. Hundt, “Scenario Based Optimization: A Framework for Statically Enabling Online Optimizations,” *Proc. 2009 Int'l Symp. Code Generation and Optimization* (CGO 09), IEEE CS Press, 2009, pp. 169-179.
7. D. Chen, N. Vachharajani, and R. Hundt, “Taming Hardware Event Samples for FDO Compilation, *Proc. 8th Ann. IEEE/ACM Int'l Symp. Code Generation and Optimization* (CGO 10), ACM Press, 2010, pp. 42-52.

Gang Ren is a senior software engineer at Google, where he's working on datacenter application performance analysis and optimization. His research interests include application performance tuning in data centers and building tools for datacenter-scale performance analysis and optimization. Ren has a PhD in computer science from the University of Illinois at Urbana-Champaign.

Eric Tune is a senior software engineer at Google, where he's working on a system that allocates computational resources and provides isolation between jobs that share machines. Tune has a PhD in computer engineering from the University of California, San Diego.

Tipp Moseley is a software engineer at Google, where he's working on datacenter-scale performance analysis. His research interests include program analysis, profiling, and optimization. Moseley has a PhD in computer science from the University of Colorado, Boulder. He is a member of ACM.

Yixin Shi is a software engineer at Google, where he's working on performance analysis tools and large-volume imagery data processing. His research interests include architectural support for securing program execution, cache design for wide-issue processors, computer architecture simulators, and large-scale data processing. Shi has a PhD in computer engineering from the University of Illinois at Chicago. He is a member of ACM.

Silvius Rus is a senior software engineer at Google, where he's working on datacenter application performance optimization through compiler and library transformations based on memory allocation and reference analysis. Rus has a PhD in computer science from Texas A&M University. He is a member of ACM.

Robert Hundt is a tech lead at Google, where he's working on compiler optimization and datacenter performance. Hundt has a Diplom Univ. in computer science from the Technical University in Munich. He is a member of IEEE and SIGPLAN.

Direct question and comments about this article to Gang Ren, Google, 1600 Amphitheatre Pkwy. Mountain View, CA 94043; gangren@google.com.

Call for Papers | General Interest

IEEE Micro seeks general-interest submissions for publication in upcoming issues. These works should discuss the design, performance, or application of microcomputer and microprocessor systems. Of special interest are articles on performance evaluation and workload characterization. Summaries of work in progress and descriptions of recently completed works are most welcome, as are tutorials. *Micro* does not accept previously published material.

Check our author center (www.computer.org/mc/mi cro/author.htm) for word, figure, and reference limits. All submissions pass through peer review consistent with other professional-level technical publications, and editing for clarity, readability, and conciseness. Contact *IEEE Micro* at micro-ma@computer.org with any questions.

IEEE
micro

ADVERTISER INFORMATION • IEEE MICRO • JULY/AUGUST 2010

Advertising Personnel

Marion Delaney: IEEE Media, Advertising Dir.
Email: md.ieeemedia@ieee.org
Phone: +1 415 863 4717

Marian Anderson: Sr. Advertising Coordinator
Email: manderson@computer.org
Phone: +1 714 821 8380 | Fax: +1 714 821 4010

Sandy Brown: Sr. Business Development Mgr.
Phone: +1 714 821 8380 | Fax: +1 714 821 4010

Advertising Sales Representatives (Recruitment)

Mid Atlantic: Lisa Rinaldo | Email: lr.ieeemedia@ieee.org
Phone: +1 732 772 0160 | Fax: +1 732 772 0164

New England: John Restchack | Email: j.restchack@ieee.org
Phone: +1 212 419 7578 | Fax: +1 212 419 7589

Southeast: Thomas M. Flynn | Email: flynntom@mindspring.com
Phone: +1 770 645 2944 | Fax: +1 770 993 4423

Midwest/Southwest: Darcy Giovingo | Email: dg.ieeemedia@ieee.org
Phone: +1 847 498 4520 | Fax: +1 847 498 5911

Northwest/Southern CA: Tim Matteson | Email: tm.ieeemedia@ieee.org
Phone: +1 310 836 4064 | Fax: +1 310 836 4067

Japan: Tim Matteson | Email: tm.ieeemedia@ieee.org
Phone: +1 310 836 4064 | Fax: +1 310 836 4067

Europe: Heleen Vodegel | Email: impress@impressmedia.com
Phone: +44 1875 825700 | Fax: +44 1875 825701

Advertising Sales Representatives (Product)

US East: Dawn Becker | Email: db.ieeemedia@ieee.org
Phone: +1 732 772 0160 | Fax: +1 732 772 0164

US Central: Darcy Giovingo | Email: dg.ieeemedia@ieee.org
Phone: +1 847 498 4520 | Fax: +1 847 498 5911

US West: Lynne Stickrod | Email: ls.ieeemedia@ieee.org
Phone: +1 415 931 9782 | Fax: +1 415 931 9782

Europe: Sven Anacker | Email: sanacker@intermediapartners.de
Phone: +49 202 27169 11 | Fax: +49 202 27169 20