Increasing Utilization in Modern Warehouse-Scale Computers Using Bubble-Up

Article *in* IEEE Micro · May 2012

DOI: 10.1109/MM.2012.22

Jason Mars
University of North Texas
84 PUBLICATIONS **2,977** CITATIONS
SEE PROFILE

Lingjia Tang
University of Michigan
77 PUBLICATIONS **2,880** CITATIONS
SEE PROFILE

Kevin Skadron
University of Virginia
319 PUBLICATIONS **15,670** CITATIONS
SEE PROFILE

Mary Lou Soffa
University of Virginia
269 PUBLICATIONS **8,949** CITATIONS
SEE PROFILE

Increasing Utilization in Modern Warehouse-Scale Computers Using Bubble-Up

Precisely predicting performance degradation due to colocating multiple executing applications on a single machine is critical for improving utilization in modern warehouse-scale computers (WSCs). Bubble-Up is the first mechanism for such precise prediction. As opposed to over-provisioning machines, Bubble-Up enables the safe colocation of multiple workloads on a single machine for Web service applications that have quality of service constraints, thus greatly improving machine utilization in modern WSCs.

Jason Mars
Lingjia Tang
Kevin Skadron
Mary Lou Soffa
University of Virginia

Robert Hundt
Google

Warehouse-scale computers (WSCs) house large-scale Web applications and cloud services.[1] The cost of constructing and operating these datacenters ranges from tens to hundreds of millions of dollars. As more computing moves onto the cloud, it's exceedingly important to leverage WSCs' resources efficiently. However, the use of the computing resources in modern WSCs remains low, often not exceeding 20 percent.[2]

Each machine in WSCs houses numerous cores, often four to eight cores per socket, and two to four sockets per machine. Cores share a number of resources, and in light of the significant potential for parallelism on a single machine, this sharing can result in performance interference across cores. This interference negatively and unpredictably impacts the quality of service (QoS) of user-facing and latency-sensitive application threads. To avoid the potential for interference, colocation is disallowed for latency-sensitive applications, leaving cores idle, and resulting in an overprovisioning that negatively impacts the entire datacenter's usage.

This overprovisioning is often unnecessary, as colocations could result in significant performance interference. The heavy-handed solution of disallowing colocation results from an inability to precisely predict the performance impact for a given colocation. On the other hand, without prediction, profiling all possible colocations' performance interference beforehand to guide colocation decisions is prohibitively expensive. The profiling complexity for all pairwise colocations is $O(N^2)$, where N is the number of applications. With hundreds to thousands of applications running in a datacenter, and the frequent development and updating of these applications, a brute-force profiling approach is impractical.

This work aims to enable the precise prediction of the performance degradation that results from contention for shared resources in the memory subsystem. A precise prediction is one that provides an expected amount of performance lost when colocated. With this information, colocations that don't violate an application's QoS threshold can be allowed, resulting in improved utilization in the datacenter.

This is a challenging problem. The most relevant related work aims to classify applications on the basis of how aggressive they are for the shared memory resources and to identify colocations to reduce contention based on the classification. Furthermore, although some work has looked at the server consolidation problem for datacenters hosting virtual machines (VMs), these works can't be applied at the application level to WSCs that aren't hosting VMs or without novel hardware changes. Until now, prior work hasn't presented a solution to precisely predict the amount of performance degradation each application suffers due to colocation, which is essential for colocation decisions of latency-sensitive applications in WSCs. In this article, we present such a solution—the Bubble-Up methodology.

Bubble-Up in a nutshell

Bubble-Up's key insight is that predicting the performance interference of co-running applications can be decoupled into measuring the pressure an application generates on the memory subsystem and measuring how much an application suffers from different levels of pressure. The underlying hypothesis is that both the pressure and sensitivity can be quantified using a common pressure metric. Having such a metric reduces the complexity of colocation analysis. As opposed to the brute force approach of profiling and characterizing every possible colocation, Bubble-Up only requires characterizing each application once to produce precise pairwise interference predictions (such as $O(N)$).

Bubble-Up is a two-step characterization process. First, each application is tested against an expanding bubble to produce a sensitivity curve. The bubble is a carefully designed stress test for the memory subsystem that provides a dial for the amount of pressure applied to the entire memory subsystem. This bubble is run along with the host application being characterized. As this dial is increased automatically (expanding the bubble), the impact on the host application is recorded, producing a sensitivity curve for the host application, such as the one illustrated in Figure 1. The *y*-axis shows the application's normalized QoS performance (latency, throughput, and so on), and the *x*-axis shows the bubble pressure. In the second step, we identify a pressure score for the application using a bubble pressure score reporter. Applying these two Bubble-Up methodology steps to each application gives us a sensitivity curve and a pressure score for each application. Given two applications *A* and *B*, we can then predict the performance impact of *A* when colocated with *B* by using *A*'s sensitivity curve to look up *A*'s relative performance at *B*'s pressure score. In the example in Figure 1, *B* has a pressure score of 2 and, as we can see from *A*'s sensitivity curve, *A*'s predicted QoS with that colocation is 90 percent.

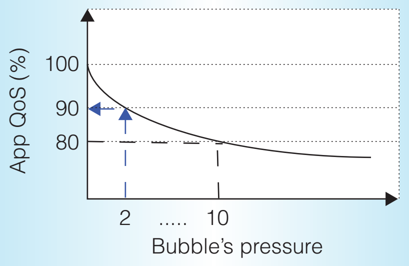

Figure 1. Example sensitivity curve for an application *A*. Assuming an application *B*'s pressure score is 2, we can predict that *A* will be performing at 90 percent of full performance.

Modern WSCs

Before taking a closer look at Bubble-Up, we must discuss how large-scale Web services are run in modern WSCs, and how we conceptualize QoS and application colocation in production WSCs.

Datacenter task placement

In modern WSCs, each Web service comprises one to hundreds of application tasks,

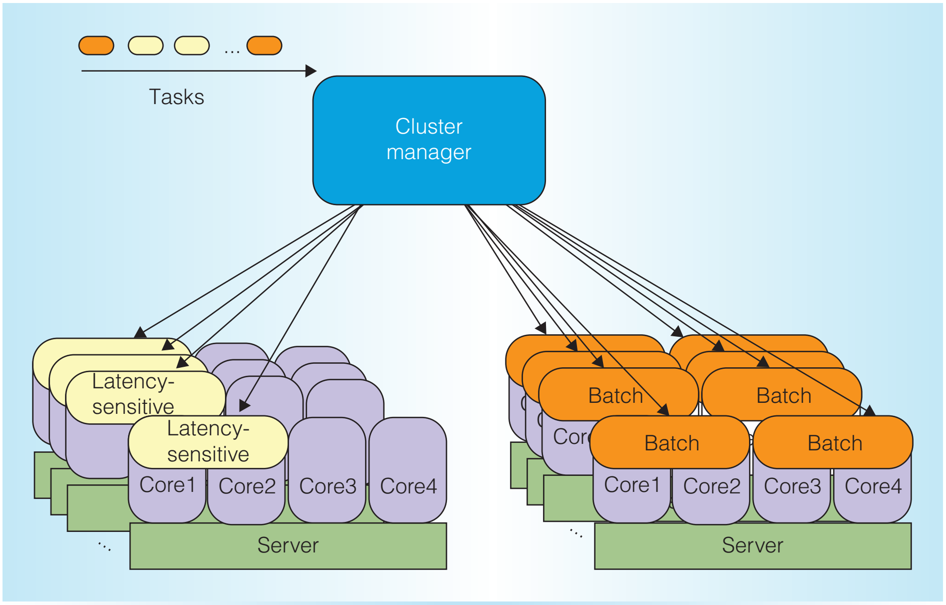

Figure 2. Task placement in a cluster. The cluster manager doesn't colocate latency-sensitive applications with others to protect their quality of service (QoS) from performance interference, causing low machine utilization.

and each task runs on a single machine. A task comprises an application binary, associated data, and a configuration file that specifies the machine-level resources required. These resources include the number of cores and amount of memory and disk space that are to be allocated to the task. A task's configuration file could also include special rules for the cluster manager, such as whether to disallow colocations with other tasks.

A cluster-level manager responsible for a number of servers conducts task placement. On the basis of the resource requirement, the cluster manager uses an algorithm similar to bin packing to place each task in a cluster of machines.[3] After a task is assigned a machine, a machine-level manager (in the form of a daemon running in user mode) uses resource containers to allocate and manage the resources belonging to the task.[4] For the remainder of this article, we use the term *application* to represent the program binary for a given component of a Web service, and *application task* to represent this binary coupled with its execution configuration file.

Figure 2 shows a simplified illustration of the task-placement process. The amount of required cores and memory resources specified in each task's configuration is carefully tuned by that task's developers to achieve the QoS requirements. Latency-sensitive tasks that disallow colocation inadvertently occupy more server resources, leading to unnecessary overprovisioning and lower machine utilization.

Application QoS

As multicores become widely adopted in datacenters, the cluster manager consolidates multiple disparate tasks on a single server to improve machine utilization. However, various application tasks in a datacenter often have different QoS priorities. User-facing applications for Web search, maps, e-mail, and other Internet services are latency sensitive, and have high QoS priorities. Applications such as file backup, offline image processing, and video compression are batch applications that often have no QoS constraints. For these, latency isn't as important. We define QoS of a latency-sensitive application in terms of the relevant performance metric specified in its internal service-level agreements (SLAs). For example, the QoS of Google's Web search is measured using query latency and queries per second,

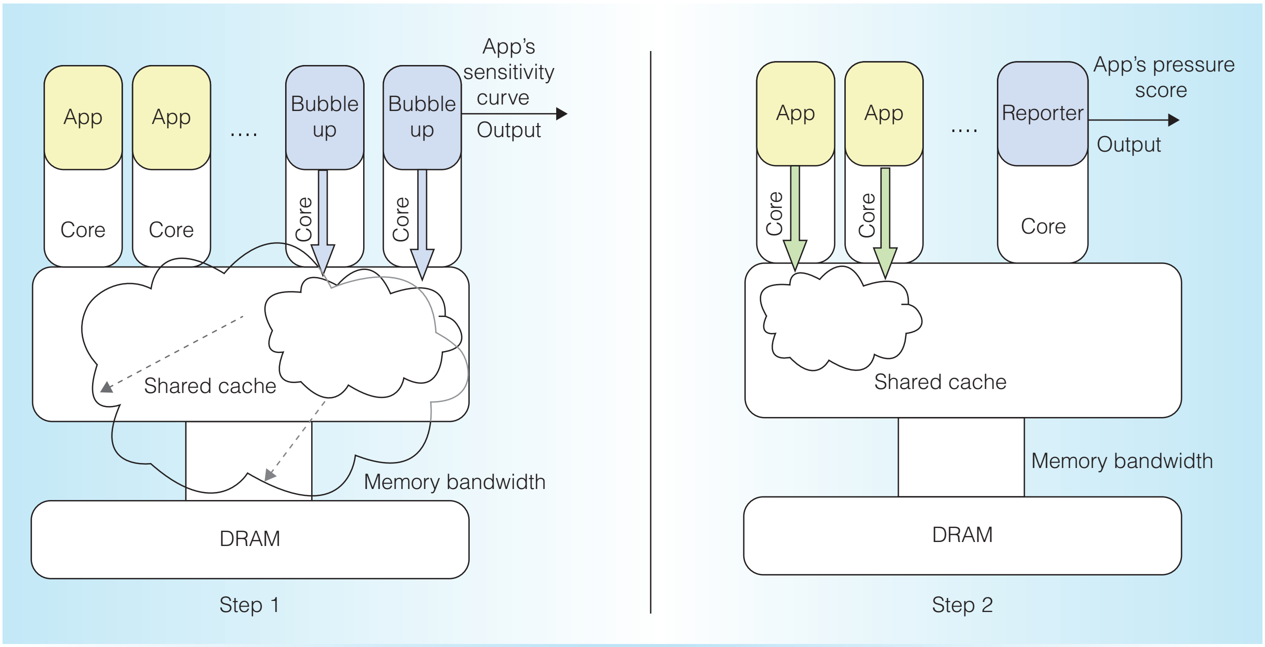

Figure 3. Bubble-Up methodology. In step 1, we characterize the host application task's sensitivity to pressure in the memory subsystem using a bubble. In step 2, we characterize the host application's contentiousness in terms of the pressure it causes on a reporter.

in contrast to Bing's,[5,6] which uses the quality of search results provided.

There's a tradeoff between the QoS performance of latency-sensitive applications and the machine utilization in WSCs. When equipped with a precise prediction of the performance degradation due to colocation, we can allow a small amount of QoS degradation from colocation to improve the machine utilization. As long as a colocation pair is predicted to cause only a small amount of QoS degradation within a specified threshold, the cluster manager can allow the colocation. We specify the tolerable amount of QoS degradation in a QoS policy. For example, a 95 percent QoS policy indicates that we're willing to sacrifice 5 percent of the QoS performance to improve machine usage. To enforce these QoS policies, we need precise prediction for QoS degradation due to colocation.

Bubble-Up methodology

The Bubble-Up methodology enables precise prediction for QoS degradation due to colocation. Figure 3 illustrates the methodology's two steps.

Step 1: Characterizing sensitivity

Bubble-Up's first step is to characterize each application task's sensitivity to pressure in the memory subsystem. As Figure 3 shows, in this step, we use a carefully designed stress test we call "the bubble" to iteratively increase the pressure applied to the memory subsystem (for example, "bubble up in the subsystem"). As we incrementally increase this pressure dial, we produce a sensitivity curve that shows how each application's QoS degrades as pressure increases. As we've previously described, we configure each task to use a prescribed number of cores for the cluster-level bin-packing algorithm used to assign tasks to machines in the WSC. This number is commonly less than the number of cores available on a socket. The bubble runs on the remaining cores.

It's important to understand that there's no correct design for the bubble. Each design needs only to approximate varying levels of pressure, and there could be many good designs. In this article, we present one such design, and show that our single bubble design is effective across myriad application workloads and architectures. Keep in mind that we made assumptions about the architectures for which this type of bubble design is applicable. Most importantly, we assume the microarchitecture uses shared last-level caches, memory controllers, and bandwidth to memory. Modern commodity processors encompass this type of design.

The art of bubble design. Although there might be many ways to design a bubble, arriving at a good design that isn't prone to error and imprecision requires a set of key requirements and guidelines that apply generally to bubble design.

- *Monotonic curves.* As the bubble's pressure increases (turning the pressure dial up), the amount of performance interference should also increase monotonically. Assuming the host application task is sensitive to cross-core interference, higher amounts of pressure should result in worse performance.
- *Wide dial range.* The pressure dial should have a range that captures the contentiousness of all the application tasks of interest. It should start from essentially no pressure, and incrementally increase pressure to a point close to the maximum possible pressure, or at least worse than the most contentious application task in the set.
- *Broad impact.* The bubble should be designed to apply pressure to the memory subsystem as a whole, not stressing a single component. However, remember that error is introduced as the difference in component pressure relative to the host task's sensitivity. This error is generally minimized if first-order effects are prioritized.

For more detail on bubble design guidelines, see our paper for the 44th Annual IEEE/ACM International Symposium on Microarchitecture.[7]

Designing the bubble. Our bubble's design principle is to use a working set size as our measure of pressure. For a given working set size, we perform memory operations in software to exercise that working set as aggressively as possible.

Our bubble is a multithreaded kernel that generates memory traffic using both loads and stores with a mixture of random and streaming accesses. The number of threads spawned is based on the configuration file of the task being characterized. The pressure dial we use is the working set size on which our kernel works. For example, a pressure size of 1 means our bubble will continuously smash a 1-Mbyte chunk of memory. As we increase the pressure, we increase our kernel's working set size. This increases the amount of data being pumped through the memory subsystem, as computation isn't the bottleneck. We provide more details on our design elsewhere.[7]

Step 2: Bubble scoring

Bubble-Up's second step characterizes each application task's contentiousness in terms of its pressure on the memory subsystem. We call this measure of contentiousness a bubble score. As Figure 3 shows, to identify an application's bubble score, we use a reporter that observes how its own performance is affected by the application to generate a score for the application. The reporter is a carefully designed single threaded workload that's sensitive to contention. As with the bubble, the reporter is only designed once and then can be used with myriad applications and architectures.

Designing the reporter. We use the reporter's own sensitivity to performance interference as a basis for reporting the pressure a host application generates. The impact the reporter felt is translated in terms of the host application's predicted bubble score. The only guideline to designing a good reporter is to have a broad sensitivity—for example, it should be sensitive to the memory subsystem holistically.

Like the bubble, there's also no correct design for the reporter. However, unlike the bubble, there is more flexibility in designing the reporter. This flexibility comes from the fact that the reporter is trained, and the sensitivity curve serves as a rubric for score reporting, no matter the shape. To implement the reporter, we use a mixture of random accesses and streaming accesses similar to those used for the bubble itself, without the high instruction-level parallelism. The working set of the reporter used in this work is about 20 Mbytes; thus, it uses the last-level cache, memory bandwidth, and prefetcher on the machines used in our evaluation.

Training the reporter. Before we can use the reporter, we must first train it using the bubble on the architecture for which it will be reporting. This training involves running the reporter against the bubble on the architecture of interest, and collecting the reporter's sensitivity curve. This needs to be done only once for each architecture. The reporter can then use its own sensitivity curve to translate a performance degradation it suffers to the corresponding bubble score. The curve is essentially used in reverse. Instead of using scores to predict QoS, we use the reporter's QoS to ascertain the colocated application's score.

Large-scale WSC workloads and their sensitivity curves

Although the cloud houses much of the world's computation, little is known about the application workloads that live in this computing domain. The characteristics of the tasks that compose a large-scale Web service vary significantly. In addition to data-retrieval tasks, there are compute-intensive tasks for analyzing, organizing, scoring, and preparing information for applications such as search, maps, and ad serving.

Google's diverse workloads

Table 1 presents several key application tasks housed in Google's production WSCs. These application tasks comprise a majority of the CPU cycles in arguably the world's largest datacenter infrastructure. In addition to each application task's name, Table 1 shows the description, priority class, and key optimization metric for each workload. Each application task corresponds to an actual binary run in the datacenter. Application tasks that are user facing, both directly and indirectly, are classified as latency sensitive, because the response time is paramount. Throughput-oriented tasks that aren't user facing are classified as batch. Note that some tasks can be used in both roles and are denoted as "Both" in the table. The "Metric" column shows each application's key metric. In the context of this work, we define each task's QoS as its performance along this metric.

We highlighted `search-render` in Table 1. This task is responsible for assembling the final view of the search process for the user, which includes assembling scored search results (including Web, image, and video), relevant ads from the `ads-servlet`, and so on. This task is highly latency sensitive and presents a compelling case that we use throughout this work to illustrate Bubble-Up's necessity and value.

Sensitivity curves of WSC workloads

Figure 4 presents the sensitivity curves generated by Bubble-Up for nine of the more sensitive Google benchmarks. Here, we examine our Bubble-Up design through analyzing the resulting sensitivity curves, and further improve our understanding of how pressure in the shared resources affects the QoS of Google's applications. To generate sensitivity curves, we adjust the pressure Bubble-Up generates and measure an application's QoS under each given pressure. Our evaluation uses a six-core Nehalem-based Xeon platform. The performance metric we used to describe each Google application's QoS is the internal metric, as presented in Table 1. We configured each application task to use three cores on the six-core Xeon. As we described earlier, during the characterization phase, the bubble runs on the remaining cores. Figures 4a to 4i present each Google application's sensitivity curve. For each figure, the *x*-axis shows the pressure on the shared memory system generated using Bubble-Up's bubble. The *y*-axis shows each application's QoS performance, normalized by its performance when it's running alone on the platform.

Figure 4 illustrates that our bubble design does indeed have the three properties we presented earlier. We observe monotonic curves. In general, each application's QoS decreases as the pressure increases.

Table 1. Google's large warehouse-scale computer (WSC) applications.

Google workload	Description	Type	Metric
`bigtable`	A distributed storage system for managing petabytes of structured data	Latency-sensitive	User time (secs)
`ads-servlet`	Ads sever responsible for selecting and placing targeted ads on syndication partners' sites	Latency-sensitive	CPU latency (ms)
`maps-detect-face`	Face detection for street view automatic face blurring	Batch	User time (secs)
`maps-detect-lp`	Optical character recognition (OCR) and text extraction from street view	Batch	User time (secs)
`maps-stitch`	Image stitching for street view	Batch	User time (secs)
`search-render`	Web search front-end server, collect results from many back ends and assembles HTML for user	Latency-sensitive	User time (secs)
`search-scoring`	Web search scoring and retrieval (traditional)	Latency-sensitive	Queries per sec
`nlp-mt-train`	Language translation	Latency-sensitive	User time (secs)
`openssl`	Secure Sockets Layer performance stress test.	Latency-sensitive	User time (secs)
`protobuf`	Protocol buffer, a mechanism for describing extensible communication protocols and on-disk structures. One of Google's most commonly used programming abstractions.	Latency-sensitive	Aggregated
`docs-analyzer`	Unsupervised Bayesian clustering tool to take keywords or text documents and explain them with meaningful clusters	Both	Throughput
`docs-keywords`	Unsupervised Bayesian clustering tool to take keywords or text documents and explain them with meaningful clusters	Both	Throughput
`rpc-bench`	Google rpc (remote procedure call) benchmark	Both	Throughput
`saw-countw`	Sawzall scripting language interpreter benchmark	Both	User time (secs)
`goog-retrieval`	Web indexing	Batch	Milliseconds per query
`youtube-x264yt`	x264yt video encoding.	Batch	User time (secs)
`zippy-test`	A lightweight compression engine designed for speed over space	Both	User time (secs)

This confirms our hypothesis that we can create an aggregate pressure dial in software that negatively and monotonically impacts an application's QoS. We also observe wide dial range. The sensitivity curves generally flatten after the Bubble-Up pressure goes beyond 20 Mbytes. At this point, the pressure on the shared cache and memory bandwidth saturates, and further increase of the pressure wouldn't have much more impact on an application's QoS. Finally, we observe Broad Impact. The monotonic trend beyond 12 Mbytes shows that we're not only saturating the cache, but also the bandwidth to memory. The pressure generated by the bubble stresses the caches, bandwidth, and prefetchers because of our bubble's streaming behavior.

Bubble-Up prediction accuracy

To evaluate Bubble-Up's accuracy when predicting applications' QoS degradation due to performance interference, we use colocations of Google applications with the SmashBench suite of contentious kernels (described in detail elsewhere[7]), as well as pairwise colocations between Google applications.

Colocating Google with SmashBench

When evaluating the Bubble-Up methodology's accuracy, it's critical to perform this evaluation across a wide spectrum of access patterns and working set sizes. As such, we first evaluate Bubble-Up's effectiveness in predicting the impact of our SmashBench workloads on Google's applications. In this

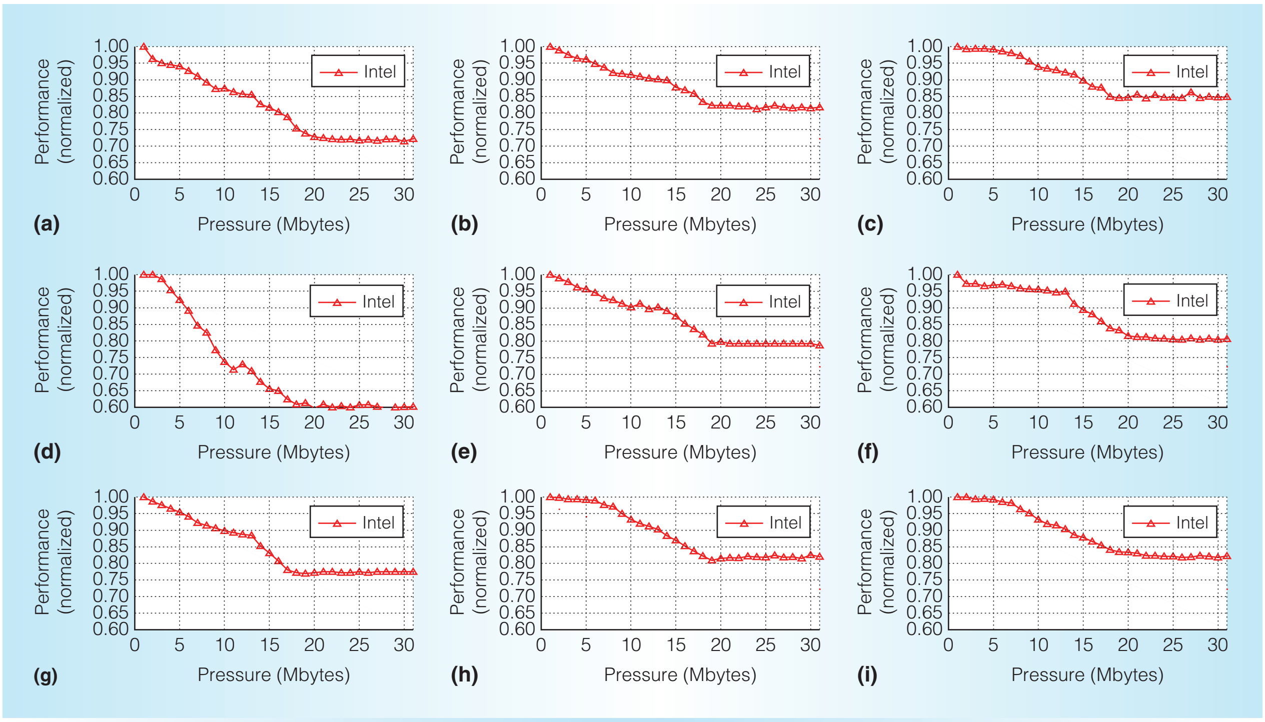

Figure 4. Bubble-Up sensitivity curves for nine highly sensitive Google workloads: `bigtable` (a), `ads-servlet` (b), `maps-detect-face` (c), `search-render` (d), `search-scoring` (e), `protobuf` (f), `docs-analyzer` (g), `saw-countw` (h), and `youtube-x264yt` (i). The *x*-axis shows the pressure on the shared memory system generated using Bubble-Up's bubble. The *y*-axis shows each application's QoS performance, normalized by its performance when it is running alone on the platform.

experiment, we apply step one of our methodology to nine memory-intensive Google applications, and we apply step two to our 15 SmashBench workloads. Step one needs to be applied only to applications whose QoS must be enforced, and step two only needs to be applied to applications that could threaten an application's QoS.

Figures 5a through 5i present the results for each of the nine Google applications. For each figure, the *x*-axis shows each of the 15 SmashBench benchmarks. The *y*-axis shows the Google application's QoS degradation. For each benchmark on the *x*-axis, the first bar shows the Google application's predicted degradation when colocated with the benchmark; the second bar shows its measured degradation. The closer the two bars are, the more accurate the Bubble-Up prediction is. Each figure's caption also documents the average prediction error for each Google application, calculated using the absolute difference between the prediction and the measured value. In general, Bubble-Up's prediction error is quite small. For the nine Google applications, the prediction error is 2.2 percent or less. SmashBench exhibits a wide range of memory-access patterns, stress points, and working set sizes. The fact that a single Bubble-Up design can accurately predict the QoS degradation caused by SmashBench demonstrates the Bubble-Up methodology's generality.

Pairwise Google colocation

Figure 6 summarizes Bubble-Up's prediction accuracy for pairwise colocations with nine of the most sensitive Google applications with the complete set of Google applications (shown in the *x*-axis). Each bar shows the error (delta) between the performance degradation predicted by Bubble-Up and the actual measured performance degradation in the colocation. Errors in the negative direction imply that the actual QoS

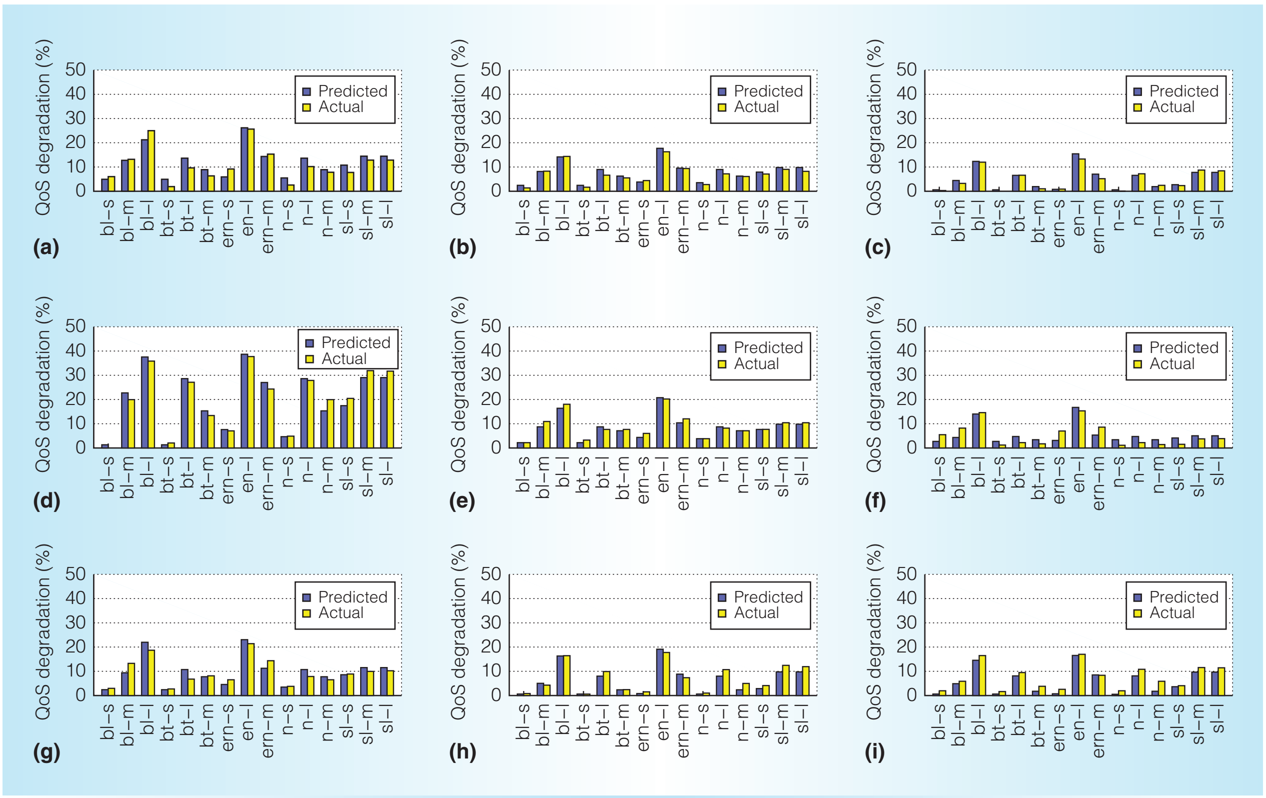

Figure 5. Bubble-Up's accuracy in predicting QoS impact for nine highly sensitive Google workloads (in the following, the average prediction errors are listed in parentheses as percentages): `bigtable` (2.2) (a); `ads-servlet` (0.8) (b); `maps-detect-face` (0.7) (c); `search-render` (1.8) (d); search-scoring (0.8) (e); protobuf (2.2) (f); `docs-analyzer` (1.7) (g); `saw-countw` (1.2) (h); and `youtube-x264yt` (1.5) (i).

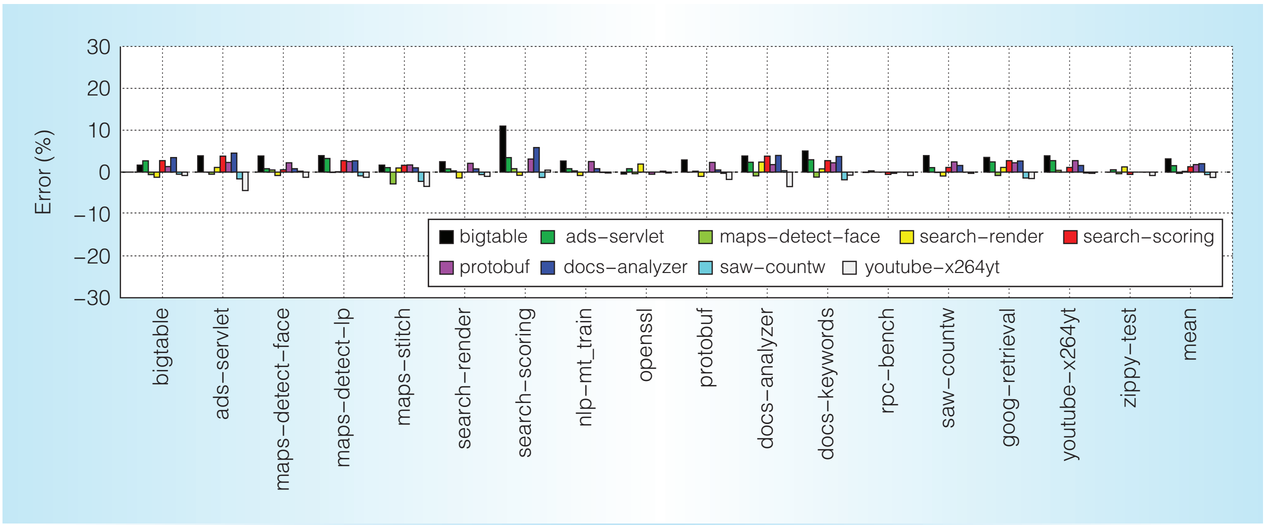

Figure 6. Bubble-Up's prediction accuracy for pairwise colocations of Google applications. Bars show the error (delta) between the predicted performance degradation and the actual measured performance degradation.

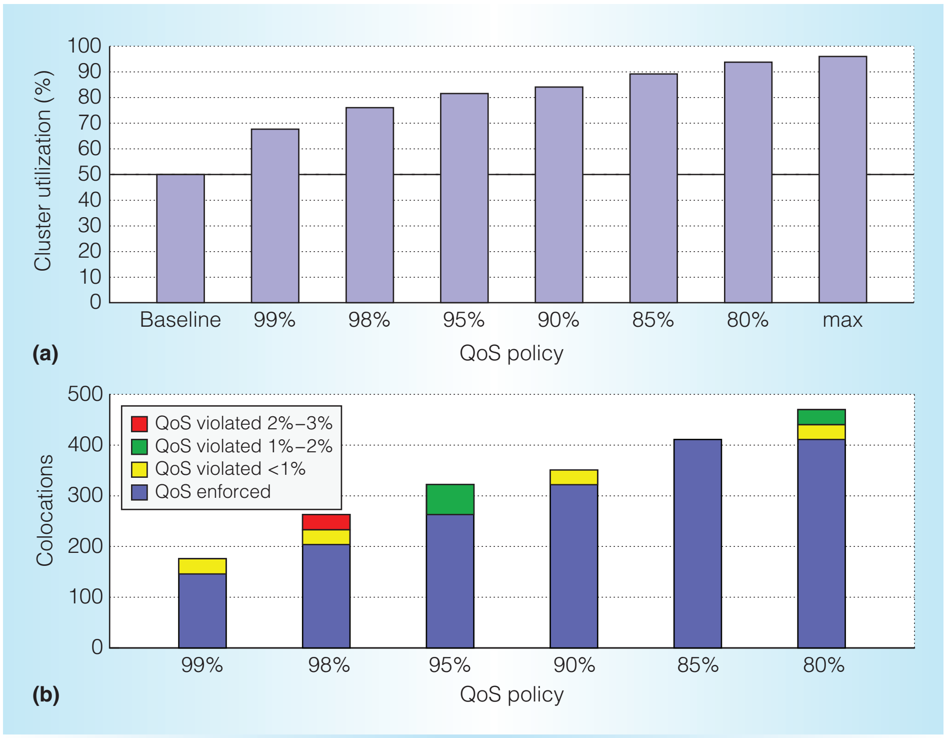

Figure 7. Impact of using Bubble-Up to allow safe colocations in WSCs. Improvement in cluster utilization when allowing Bubble-Up colocations with `search-render` under each QoS policy (a); number of Bubble-Up colocations under each QoS policy (b).

degradation is worse (more) than predicted; errors in the positive direction imply that the actual QoS degradation is better (less) than predicted. Only errors in the negative direction can result in a violation of a QoS policy. As the figure shows, Bubble-Up's prediction error is fairly small across all Google pairwise colocations.

Improving utilization with Bubble-Up

As Bubble-Up proves to be effective at precisely predicting the performance impact of colocations, the question arises of how we improve utilization throughout a WSC with Bubble-Up. To improve machine usage, we let latency-sensitive applications have a small amount of QoS degradation. Each application's QoS policy specifies the tolerable degradation threshold. Using Bubble-Up, we can predict the QoS degradation and allow colocation of latency-sensitive applications with other applications when the predicted QoS degradation is within the specified threshold. To evaluate Bubble-Up's effectiveness, we constructed a scenario using Google's commercial workloads, large traces of real queries from production, and production machines.

For this scenario, we use a cluster composed of 500 machines. In this experiment, we focused on `search-render` as our main latency-sensitive application whose QoS degradation must remain small. This cluster has 500 instances of `search-render`, each placed on a single machine. There are 500 other Google applications, evenly distributed across the 15 application types in Table 1. Every application uses three cores. Our evaluation baseline is the currently deployed cluster management that disallows colocation of `search-render` with any other applications. In this experiment, we investigated the potential colocation and utilization gained using Bubble-Up predictions under varying QoS policies.

Figure 7a presents the cluster's usage results after employing Bubble-Up prediction under various QoS policies. The baseline is the utilization of the cluster when colocation is disallowed and each instance of `search-render` occupies three of the six cores on a single machine, and thus at 50 percent cluster utilization. The maximum utilization is achieved by allowing all colocations—placing each of the 500 other Google applications to co-run with a `search-render` on every machine, regardless of `search-render`'s QoS degradation. The maximum utilization isn't 100 percent because we define a machine's utilization as the aggregate performance of all applications running on the machine, normalized by their solo performance. For example, applications *A* and *B* could be colocated, occupying all six cores on a machine. Because of cross-core interference, their performance is only 90 percent of that when running alone, occupying three cores on a machine. The resulting machine utilization when colocated in this case would be 90 percent instead of 100 percent.

As Figure 7a demonstrates, Bubble-Up prediction greatly improves machine utilization. Even under 99 percent of QoS policy (when the tolerable QoS degradation is only 1 percent), the usage improves from 50 percent to close to 70 percent. Allowing a more relaxed QoS policy improves the usage even more. Under the 80 percent QoS policy, the utilization improvement is close to 80 percent, showing a large potential benefit when adopting Bubble-Up in WSCs.

Figure 7b presents the total number of colocations the cluster manager allowed based on Bubble-Up's prediction under each QoS policy. Similar to utilization, the number of colocations increases as the allowed QoS degradation increases. The baseline colocation is 0. With 99 percent QoS policy, the colocation is close to 200. With 80 percent QoS policy, the allowed colocations increase to 400. However, because of Bubble-Up's prediction error, there could be colocations that violate the policy's specified QoS threshold. Stack bars present both the number of colocations that satisfy the QoS policy and the number of violations. The violations are broken down into three categories: violations that cause less than 1 percent of extra degradation beyond the QoS policy, 1 to 2 percent of extra degradation, and 2 to 3 percent of additional degradation. For example, as the figure shows, under 99 percent QoS policy, around 10 percent of the colocations violate the policy. However, all of these violations only cause less than 1 percent extra QoS degradation beyond the policy, meaning their QoS is within a 98 percent QoS policy. We describe many more results elsewhere, spanning three disparate architectures.[7]

Bubble-Up is not only a technique that solves an important and pressing problem, but also a philosophy that inspires a new way of thinking about establishing new metrics for architecting and optimizing modern WSCs. When developing real-world solutions to be deployed on commodity processors, black-box profiling methodologies are often preferable to white-box analytical approaches. This observation results from the fact that the real-world impact of contention and performance interference is in large part determined by the set of memory subsystem components and optimizations (prefetchers, queuing protocols, replacement policies, and victim caches) present on the architecture of interest. Modeling all of these factors as they exist on real architectures often proves intractable because the complexity is high, and many of these components' designs are trade secrets.

In addition to solving an important and pressing problem, Bubble-Up is an enabling technology. We can leverage Bubble-Up dynamically to perform short probes of already running services to analyze colocation decisions in flight. This type of approach opens up new possibilities in enabling adaptive methods in managing QoS and maximizing utilization at finer granularities. Bubble-Up presents the core principles to enable such developments.

The challenge of understanding how applications interact and interfere with each other goes beyond the domain of WSCs. Although our Bubble-Up methodology is particularly useful for increasing utilization in modern WSCs, it can also be applied anywhere a known set of workloads are continuously executed on multicore and many-core systems—and, with the insights and principles gained from this static approach, a dynamic approach is right around the corner. MICRO

References

1. U. Hölzle and L.A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines,* Morgan and Claypool, 2009.
2. L.A. Barroso and U. Hölzle, ''The Case for Energy-Proportional Computing,'' *Computer,* vol. 40, Dec. 2007, pp. 33-37.
3. A. Mishra et al., ''Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters,'' *ACM SIGMETRICS Performance Evaluation Rev.,* vol. 37, no. 4, 2010, pp. 34-41.
4. G. Banga, P. Druschel, and J.C. Mogul, ''Resource Containers: A New Facility for Resource Management in Server Systems,'' *Proc. 3rd Symp. Operating Systems, Design, and Implementation* (OSDI 99), Usenix Assoc., 1999, pp. 45-58.
5. V.J. Reddi et al., ''Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency,'' *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA 10), ACM, 2010, pp. 314-325.
6. C. Kozyrakis et al., ''Server Engineering Insights for Large-Scale Online Services,'' *IEEE Micro,* vol. 30, no. 4, 2010, pp. 8-19.
7. J. Mars et al., ''Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Colocations,'' *Proc. 44th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM, 2011, pp. 248-259.

Jason Mars is a PhD candidate in the Computer Science Department at the University of Virginia. His research interests include adaptive systems in software and hardware, warehouse-scale computer architecture, and software/hardware codesigned architectures. Mars has an MS in computer science from the University of Virginia. He is a member of IEEE and the ACM.

Lingjia Tang is a PhD candidate in the Computer Science Department at the University of Virginia. Her research interests include compilers, runtime systems, and computer architecture. Tang has an MS in computer science from the University of Virginia.

Kevin Skadron is a professor in the Computer Science Department at the University of Virginia. His research interests focus on parallel and accelerator architectures in the presence of physical constraints such as temperature and power. Skadron has a PhD in computer science from Princeton University. He is a senior member of IEEE and the IEEE Computer Society and a distinguished member of the ACM.

Mary Lou Soffa is the Owen R. Cheatham Professor and department chair of the Computer Science Department at the University of Virginia. Her research interests include optimizing compilers, virtual execution environments, software testing, program analysis, instruction-level parallelism, and multicore architectures. Soffa has a PhD in computer science from the University of Pittsburgh. She is an ACM Fellow.

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

Direct questions and comments about this article to Jason Mars, 2000 Jefferson Park Ave., Apt. 12, Charlottesville, VA, 22903; mars.ninja@gmail.com.