# Improving Resource Efficiency at Scale with Heracles

DAVID LO, Google Inc., Stanford University
LIQUN CHENG, RAMA GOVINDARAJU, and PARTHASARATHY RANGANATHAN, Google Inc.
CHRISTOS KOZYRAKIS, Stanford University

6

User-facing, latency-sensitive services, such as websearch, underutilize their computing resources during daily periods of low traffic. Reusing those resources for other tasks is rarely done in production services since the contention for shared resources can cause latency spikes that violate the service-level objectives of latency-sensitive tasks. The resulting under-utilization hurts both the affordability and energy efficiency of large-scale datacenters. With the slowdown in technology scaling caused by the sunsetting of Moore's law, it becomes important to address this opportunity.

We present Heracles, a feedback-based controller that enables the safe colocation of best-effort tasks alongside a latency-critical service. Heracles dynamically manages multiple hardware and software isolation mechanisms, such as CPU, memory, and network isolation, to ensure that the latency-sensitive job meets latency targets while maximizing the resources given to best-effort tasks. We evaluate Heracles using production latency-critical and batch workloads from Google and demonstrate average server utilizations of 90% without latency violations across all the load and colocation scenarios that we evaluated.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Scheduling*

Additional Key Words and Phrases: Datacenter, warehouse-scale computer, resource efficiency, interference, scheduling, QoS, latency-critical applications, performance isolation

**ACM Reference Format:**
David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2016. Improving resource efficiency at scale with Heracles. ACM Trans. Comput. Syst. 34, 2, Article 6 (May 2016), 33 pages.
DOI: http://dx.doi.org/10.1145/2882783

---

This work was supported by a Google research grant, the Stanford Experimental Datacenter Lab, and NSF Grant No. CNS-1422088. David Lo was supported by a Google PhD Fellowship.
Prior publication: David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (ISCA'15).

In this work, we extend the previous work with several new analyses and an extended mathematical treatment of the gradient descent process used in the controller. First, we demonstrate the pitfalls of applying a throughput metric (Instructions Per Cycle) to controlling the latency of latency-critical workloads and show that application-level latency is needed. Second, we evaluate two designs for last-level cache partitioning. One of the designs is a hard partition between latency-critical and best-effort workloads (exclusive), and the other is a soft partition where the best effort workloads are jailed to a portion of the cache while the latency-critical workload can access all of the cache (shared). Finally, we describe the gradient descent algorithm and how it is used in the memory-cache-CPU controller to search for the best partitioning of CPUs and the last-level cache between latency-critical and best-effort tasks.
Authors' addresses: D. Lo, L. Cheng, R. Govindaraju, and P. Ranganathan, Google Inc, 1600 Amphitheatre Parkway, Mountain View, CA 94043; emails: {davidlo, liquncheng, govindaraju, parthas}@google.com; C. Kozyrakis, Electrical Engineering Department, Stanford University, 353 Serra Mall, Stanford, CA 94305; email: kozyraki@stanford.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
2016 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 0734-2071/2016/05-ART6 $15.00
DOI: http://dx.doi.org/10.1145/2882783

## 1. INTRODUCTION

Public and private cloud frameworks allow us to host an increasing number of workloads in large-scale datacenters with tens of thousands of servers. The business models for cloud services emphasize reduced infrastructure costs. Of the total cost of ownership (TCO) for modern energy-efficient datacenters, servers are the largest fraction (50%–70%) [Barroso et al. 2013]. Maximizing server utilization is therefore important for continued scaling.

Until recently, scaling from Moore's law provided higher compute per dollar with every server generation, allowing datacenters to scale without raising the cost. However, with the sunset of Moore's law on the horizon due to several imminent challenges in technology scaling [Esmaeilzadeh et al. 2011; Hardavellas et al. 2011], alternate approaches are needed to continue scaling datacenter capability. Some efforts seek to reduce server costs through balanced designs or cost-effective components [Janapa Reddi et al. 2010; Malladi et al. 2012; Lim et al. 2012]. An orthogonal approach is to improve the return on investment and utility of datacenters by raising server utilization. Low utilization negatively impacts both operational and capital components of cost-efficiency. Energy proportionality can reduce operational expenses at low utilization [Barroso and Hölzle 2007; Lo et al. 2014]. But, to amortize the much larger capital expenses, an increased emphasis on the *effective use of server resources* is warranted.

Several studies have established that the average server utilization in most datacenters is low, ranging between 10% and 50% [McKinsey & Company 2008; Vasan et al. 2010; Reiss et al. 2012; Barroso et al. 2013; Delimitrou and Kozyrakis 2014; Carvalho et al. 2014]. A primary reason for the low utilization is the popularity of latency-critical (LC) services such as social media, search engines, software-as-a-service, online maps, webmail, machine translation, online shopping, and advertising. These user-facing services are typically scaled across thousands of servers and access distributed state stored in memory or Flash across these servers. While their load varies significantly due to diurnal patterns and unpredictable spikes in user accesses, it is difficult to consolidate load on a subset of highly utilized servers because the application state does not fit in a small number of servers and moving state is expensive. The cost of such underutilization can be significant. For instance, Google websearch servers often have an average idleness of 30% over a 24-hour period [Lo et al. 2014]. For a hypothetical cluster of 10,000 servers, this idleness translates to a wasted capacity of 3,000 servers.

A promising way to improve efficiency is to launch best-effort batch (BE) tasks on the same servers and exploit any resources underutilized by LC workloads [Marshall et al. 2011; Mars et al. 2011; Delimitrou and Kozyrakis 2013]. Batch analytics frameworks can generate numerous BE tasks and derive significant value even if these tasks are occasionally deferred or restarted [Delimitrou and Kozyrakis 2014; Boutin et al. 2014; Carvalho et al. 2014; Curino et al. 2014]. The main challenge of this approach is interference between colocated workloads on shared resources such as caches, memory, input/output (I/O) channels, and network links. LC tasks operate with strict service level objectives (SLOs) on tail latency, and even small amounts of interference can cause significant SLO violations [Mars et al. 2011; Meisner et al. 2011; Leverich and Kozyrakis 2014]. Hence, some of the past work on workload colocation focused only on throughput workloads [Nathuji et al. 2010; Cook et al. 2013]. More recent systems predict or detect when an LC task suffers significant interference from the colocated tasks and *avoid or terminate* the colocation [Vasić et al. 2012; Novakovic et al. 2013; Delimitrou and Kozyrakis 2014; Mars et al. 2011, 2012; Zhang et al. 2014]. These systems protect LC workloads but reduce the opportunities for higher utilization through colocation.

Recently introduced hardware features for cache isolation and fine-grained power control allow us to improve colocation. This work aims to enable aggressive colocation of LC workloads and BE jobs by automatically coordinating multiple hardware and software isolation mechanisms in modern servers. We focus on two hardware mechanisms, shared cache partitioning and fine-grained power/frequency settings, and two software mechanisms, core/thread scheduling and network traffic control. Our goal is to eliminate SLO violations at all levels of load for the LC job while maximizing the throughput for BE tasks.

There are several challenges towards this goal. First, we must carefully share each individual resource; conservative allocation will minimize the throughput for BE tasks, while optimistic allocation will lead to SLO violations for the LC tasks. Second, the performance of both types of tasks depends on multiple resources, which leads to a large allocation space that must be explored in real time as load changes. Finally, there are non-obvious interactions between isolated and non-isolated resources in modern servers. For instance, increasing the cache allocation for an LC task to avoid evictions of hot data may create memory bandwidth interference due to the increased misses for BE tasks.

We present *Heracles*[1] [Lo et al. 2015], a real-time, dynamic controller that manages four hardware and software isolation mechanisms in a coordinated fashion to maintain the SLO for an LC job. Compared to existing systems [Zhang et al. 2013; Mars et al. 2011; Delimitrou and Kozyrakis 2014] that *prevent* colocation of interfering workloads, Heracles *enables* an LC task to be colocated with any BE job. It guarantees that the LC workload receives just enough of each shared resource to meet its SLO, thereby maximizing the utility from the BE task. Using online monitoring and some offline profiling information for LC jobs, Heracles identifies when shared resources become saturated and are likely to cause SLO violations and configures the appropriate isolation mechanism to proactively prevent that from happening.

The specific contributions of this work are the following. First, we characterize the impact of interference on shared resources for a set of production, latency-critical workloads at Google, including websearch, an online machine learning clustering algorithm, and an in-memory key-value store. We show that the impact of interference is non-uniform and workload dependent, thus precluding the possibility of static resource partitioning within a server. Next, we design Heracles and show that (a) using application-level latency in the control algorithm is critical for guaranteeing the quality of service for latency-critical applications; (b) coordinated management of multiple isolation mechanisms is key to achieving high utilization without SLO violations; (c) carefully separating interference into independent subproblems is effective at reducing the complexity of the dynamic control problem; and (d) a local, real-time controller that monitors latency in each server is sufficient. We evaluate Heracles on production Google servers by using it to colocate production LC and BE tasks . We show that Heracles achieves an effective machine utilization of 90% averaged across all colocation combinations and loads for the LC tasks while meeting the latency SLOs. Heracles also improves throughput/TCO by 15% to 300%, depending on the initial average utilization of the datacenter. Finally, we establish the need for hardware mechanisms to monitor and isolate Dynamic Random-Access Memory (DRAM) bandwidth, which can improve Heracles' accuracy and eliminate the need for offline information.

To the best of our knowledge, this is the first study to make coordinated use of new and existing isolation mechanisms in a real-time controller to demonstrate significant improvements in efficiency for production systems running LC services.

[1]The mythical hero that killed the multi-headed monster, Lernaean Hydra.

## 2. SHARED RESOURCE INTERFERENCE

When two or more workloads execute concurrently on a server, they compete for shared resources. This section reviews the major sources of interference, the available isolation mechanisms, and the motivation for dynamic management.

The primary shared resource in the server are the **cores** in the one or more CPU sockets. We cannot simply *statically* partition cores between the LC and BE tasks using mechanisms such as `cgroups cpuset` [Menage 2007]. When user-facing services such as search face a load spike, they need all available cores to meet throughput demands without latency SLO violations. Similarly, we cannot simply assign high priority to LC tasks and rely on Operating System (OS) level scheduling of cores between tasks. Common scheduling algorithms such as Linux's completely fair scheduler (CFS) have vulnerabilities that lead to frequent SLO violations when LC tasks are colocated with BE tasks [Leverich and Kozyrakis 2014]. Real-time scheduling algorithms (e.g., `SCHED_FIFO`) are not work preserving and lead to lower utilization. The availability of HyperThreads in Intel cores leads to further complications, as a HyperThread executing a BE task can interfere with an LC HyperThread on instruction bandwidth, shared L1/L2 caches, and TLBs.

Numerous studies have shown that uncontrolled interference on the shared *last-level cache (LLC)* can be detrimental for colocated tasks [Sanchez and Kozyrakis 2011; Mars et al. 2012; Delimitrou and Kozyrakis 2014; Govindan et al. 2011; Leverich and Kozyrakis 2014]. To address this issue, Intel has recently introduced LLC cache partitioning in server chips. This functionality is called *Cache Allocation Technology (CAT)*, and it enables way partitioning of a highly associative LLC into several subsets of smaller associativity [Intel 2014]. Cores assigned to one subset can only allocate cache lines in their subset on refills but are allowed to hit in any part of the LLC. It is already well understood that, even when the colocation is between throughput tasks, it is best to dynamically manage cache partitioning using either hardware [Iyer et al. 2007; Qureshi and Patt 2006; Cook et al. 2013] or software [Nathuji et al. 2010; Lin et al. 2008] techniques. In the presence of user-facing workloads, dynamic management is more critical as interference translates to large latency spikes [Leverich and Kozyrakis 2014]. It is also more challenging as the cache footprint of user-facing workloads changes with load [Kasture and Sanchez 2014].

Most important LC services operate on large datasets that do not fit in on-chip caches. Hence, they put pressure on DRAM bandwidth at high loads and are sensitive to *DRAM bandwidth* interference. Despite significant research on memory bandwidth isolation [Iyer et al. 2007; Muralidhara et al. 2011; Jeong et al. 2012; Nesbit et al. 2006], there are no hardware isolation mechanisms in commercially available chips. In multi-socket servers, one can isolate workloads across Non-Uniform Memory Access (NUMA) channels [Blagodurov et al. 2011; Tang et al. 2011], but this approach constrains DRAM capacity allocation and address interleaving. The lack of hardware support for memory bandwidth isolation complicates and constrains the efficiency of any system that dynamically manages workload colocation.

Datacenter workloads are scale-out applications that generate *network traffic*. Many datacenters use rich topologies with sufficient bisection bandwidth to avoid routing congestion in the fabric [Issariyakul and Hossain 2010; Al-Fares et al. 2008]. There are also several networking protocols that prioritize short messages for LC tasks over large messages for BE tasks [Alizadeh et al. 2010; Wilson et al. 2011]. Within a server, interference can occur both in the incoming and outgoing direction of the network link. If a BE task causes incast interference, then we can throttle its core allocation until networking flow-control mechanisms trigger [Podlesny and Williamson 2012]. In the outgoing direction, we can use traffic control mechanisms in operating systems

like Linux to provide bandwidth guarantees to LC tasks and to prioritize their messages ahead of those from BE tasks [Brown 2006]. Traffic control must be managed dynamically as bandwidth requirements vary with load. Static priorities can cause underutilization and starvation [Pattara-Aukom et al. 2002]. Similar traffic control can be applied to solid-state storage devices [Seong et al. 2010].

*Power* is an additional source of interference between colocated tasks. All modern multi-core chips have some form of dynamic overclocking, such as Turbo Boost in Intel chips and Turbo Core in AMD chips. These techniques opportunistically raise the operating frequency of the processor chip higher than the nominal frequency in the presence of power headroom. Thus, the clock frequency for the cores used by an LC task depends not just on its own load but also on the intensity of any BE task running on the same socket. In other words, the performance of LC tasks can suffer from unexpected drops in frequency due to colocated tasks. This interference can be mitigated with per-core dynamic voltage frequency scaling, as cores running BE tasks can have their frequency decreased to ensure that the LC jobs maintain a guaranteed frequency. A static policy would run all BE jobs at minimum frequency, thus ensuring that the LC tasks are not power limited. However, this approach severely penalizes the vast majority of BE tasks. Most BE jobs do not have the profile of a power virus[2] and LC tasks only need the additional frequency boost during periods of high load. Thus, a dynamic solution that adjusts the allocation of power between cores is needed to ensure that LC cores run at a guaranteed minimum frequency while maximizing the frequency of cores for BE tasks.

A major challenge with colocation is *cross-resource interactions*. A BE task can cause interference in all the shared resources discussed. Similarly, many LC tasks are sensitive to interference on multiple resources. Therefore, it is not sufficient to manage one source of interference: All potential sources need to be monitored and carefully isolated if need be. In addition, interference sources interact with each other. For example, LLC contention causes both types of tasks to require more DRAM bandwidth, also creating a DRAM bandwidth bottleneck. Similarly, a task that notices network congestion may attempt to use compression, causing core and power contention. In theory, the number of possible interactions scales with the square of the number of interference sources, making this a very difficult problem.

There are additional sources of interference that we do not examine in this work, such as storage devices (disks or Solid State Drives (SSDs)) and other I/O devices. The workloads that we analyze in Section 3 are so latency critical that they do not rely on these slow devices that inherently have high and unpredictable tail latency characteristics. Nevertheless, we discuss in related work other solutions that have been proposed to handle performance isolation for storage and I/O.

## 3. INTERFERENCE CHARACTERIZATION & ANALYSIS

This section characterizes the impact of interference on shared resources for latency-critical services.

### 3.1. Latency-Critical Workloads

We examine three Google production latency-critical workloads and their sensitivity to interference in various shared resources. *websearch* is the query serving portion of a production web search service. It is a scale-out workload that provides high throughput with a strict latency SLO by using a large fan-out to thousands of leaf nodes that

[2]A computation that maximizes activity and power consumption of a core.

process each query on their shard of the search index. The SLO for leaf nodes is in the tens of milliseconds for the 99%-ile latency. Load for websearch is generated using an anonymized trace of real user queries.

The websearch workload has high memory footprint as it serves shards of the search index stored in DRAM. It also has moderate DRAM bandwidth requirements (40% of available bandwidth at 100% load), as most index accesses miss in the LLC. However, there is a small but significant working set of instructions and data in the hot path. Also, websearch is fairly compute intensive, as it needs to score and sort search hits. However, it does not consume a significant amount of network bandwidth. For this study, we reserve a small fraction of DRAM on search servers to enable colocation of BE workloads with websearch.

*ml_cluster* is a standalone service that performs real-time text clustering using machine-learning techniques. Several Google services use ml_cluster to assign a cluster to a snippet of text. The ml_cluster workload performs this task by locating the closest clusters for the text in a model that was previously learned offline. This model is kept in main memory for performance reasons. The SLO for ml_cluster is a 95%-ile latency guarantee of tens of milliseconds. The ml_cluster workload is exercised using an anonymized trace of requests captured from production services.

Compared to websearch, ml_cluster is more memory bandwidth intensive (with 60% DRAM bandwidth usage at peak) but slightly less compute intensive (lower CPU power usage overall). It has low network bandwidth requirements. An interesting property of ml_cluster is that each request has a very small cache footprint, but, in the presence of many outstanding requests, the combined cache footprint of all the queries translates to a large amount of cache pressure that spills over to DRAM. This behavior is reflected in our analysis as a super-linear growth in DRAM bandwidth use for ml_cluster versus load.

*memkeyval* is an in-memory key-value store, similar to *memcached* [Nishtala et al. 2013]. The memkeyval workload is used as a caching service in the backends of several Google web services. Other large-scale web services, such as Facebook and Twitter, use memcached extensively. The memkeyval workload has significantly less processing per request compared to websearch, leading to extremely high throughput in the order of hundreds of thousands of requests per second at peak. Since each request is processed quickly, the SLO latency is very low, in the few hundreds of microseconds for the 99%-ile latency. Load generation for memkeyval uses an anonymized trace of requests captured from production services.

At peak load, memkeyval is network bandwidth limited. Despite the small amount of network protocol processing done per request, the high request rate makes memkeyval compute bound. In contrast, DRAM bandwidth requirements are low (20% DRAM bandwidth utilization at max load), as requests simply retrieve values from DRAM and put the response on the wire. The memkeyval workload has both a static working set in the LLC for instructions, as well as a per-request data working set.

### 3.2. Characterization Methodology

To understand their sensitivity to interference on shared resources, we ran each of the three LC workloads with a synthetic benchmark that stresses each resource in isolation. While these are single node experiments, there can still be significant network traffic as the load is generated remotely. We repeated the characterization at various load points for the LC jobs and recorded the impact of the colocation on tail latency. We used production Google servers with dual-socket Intel Xeons based on the Haswell architecture. Each CPU has a high core count, with a nominal frequency of 2.3GHz and 2.5MB of LLC per core. The chips have hardware support for way partitioning of the LLC.

We performed the following characterization experiments:

—**Cores:** As we discussed in Section 2, we cannot share a logical core (a single HyperThread) between an LC and a BE task because OS scheduling can introduce latency spikes in the order of tens of milliseconds [Leverich and Kozyrakis 2014]. Hence, we focus on the potential of using separate HyperThreads that run pinned on the same physical core. We characterize the impact of a colocated HyperThread that implements a tight spinloop on the LC task. This experiment captures a *lower bound* of HyperThread interference. A more compute or memory intensive microbenchmark would antagonize the LC HyperThread for more core resources (e.g., execution units) and space in the private caches (L1 and L2). Hence, if this experiment shows high impact on tail latency, we can conclude that core sharing through HyperThreads is not a practical option.
—**LLC:** The interference impact of LLC antagonists is measured by pinning the LC workload to enough cores to satisfy its SLO at the specific load and pinning a cache antagonist that streams through a large data array on the remaining cores of the socket. We use several array sizes that take up a quarter, half, and almost all of the LLC and denote these configurations as LLC small, medium, and big respectively.
—**DRAM bandwidth:** The impact of DRAM bandwidth interference is characterized in a similar fashion to LLC interference, using a significantly larger array for streaming. We use *numactl* to ensure that the DRAM antagonist and the LC task are placed on the same socket(s) and that all memory channels are stressed.
—**Network traffic:** We use *iperf*, an open source TCP streaming benchmark [iperf 2011], to saturate the network transmit (outgoing) bandwidth. All cores except for one are given to the LC workload. Since the LC workloads we consider serve request from multiple clients connecting to the service they provide, we generate interference in the form of many low-bandwidth "mice" flows. Network interference can also be generated using a few "elephant" flows. However, such flows can be effectively throttled by TCP congestion control [Briscoe 2007], while the many "mice" flows of the LC workload will not be impacted.
—**Power:** To characterize the latency impact of a power antagonist, the same division of cores is used as in the cases of generating LLC and DRAM interference. Instead of running a memory access antagonist, a CPU power virus is used. The power virus is designed such that it stresses all the components of the core, leading to high power draw and lower CPU core frequencies.
—**OS Isolation:** For completeness, we evaluate the overall impact of running a BE task along with an LC workload using only the isolation mechanisms available in the OS. Namely, we execute the two workloads in separate Linux containers and set the BE workload to be low priority. The scheduling policy is enforced by CFS using the `shares` parameter, where the BE task receives very few `shares` compared to the LC workload. No other isolation mechanisms are used in this case. The BE task is the Google *brain* workload [Le et al. 2012; Rosenberg 2013], which we will describe further in Section 5.1.

### 3.3. Interference Analysis

Figure 1 presents the impact of the interference microbenchmarks on the tail latency of the three LC workloads. Each row in the table shows tail latency at a certain load for the LC workload when colocated with the corresponding microbenchmark. The interference impact is acceptable if and only if the tail latency is less than 100% of the target SLO. We color-code red/yellow all cases where SLO latency is violated.

By observing the rows for brain, we immediately notice that current OS isolation mechanisms are inadequate for colocating LC tasks with BE tasks. Even at low loads,

**websearch**

| | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% |
|---|---|---|---|---|---|---|---|---|---|
| LLC (small) | 103% | 96% | 102% | 96% | 104% | 100% | 100% | 103% | 103% |
| LLC (med) | 106% | 99% | 111% | 103% | 116% | 108% | 110% | 125% | 111% |
| LLC (big) | >300% | >300% | >300% | >300% | >300% | >300% | >300% | 264% | 123% |
| DRAM | >300% | >300% | >300% | >300% | >300% | >300% | >300% | 270% | 122% |
| HyperThread | 109% | 106% | 113% | 114% | 105% | 117% | 119% | 136% | >300% |
| CPU power | 124% | 107% | 115% | 108% | 114% | 105% | 101% | 100% | 99% |
| Network | 35% | 36% | 36% | 37% | 38% | 41% | 48% | 55% | 64% |
| brain | 165% | 173% | 168% | 230% | >300% | >300% | >300% | >300% | >300% |

**ml_cluster**

| | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% |
|---|---|---|---|---|---|---|---|---|---|
| LLC (small) | 88% | 84% | 110% | 93% | 216% | 106% | 105% | 206% | 202% |
| LLC (med) | 88% | 91% | 115% | 104% | >300% | 212% | 220% | 212% | 205% |
| LLC (big) | >300% | >300% | >300% | >300% | >300% | >300% | >300% | 250% | 214% |
| DRAM | >300% | >300% | >300% | >300% | >300% | >300% | >300% | 287% | 223% |
| HyperThread | 109% | 111% | 100% | 107% | 112% | 114% | 119% | 130% | 262% |
| CPU power | 101% | 89% | 86% | 90% | 92% | 90% | 89% | 92% | 97% |
| Network | 56% | 60% | 58% | 58% | 59% | 59% | 63% | 67% | 89% |
| brain | 149% | 189% | 202% | 217% | 239% | >300% | >300% | >300% | >300% |

**memkeyval**

| | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% |
|---|---|---|---|---|---|---|---|---|---|
| LLC (small) | 88% | 91% | 101% | 91% | 101% | 138% | 140% | 150% | 78% |
| LLC (med) | 148% | 107% | 119% | 108% | 138% | 230% | 181% | 162% | 100% |
| LLC (big) | >300% | >300% | >300% | >300% | >300% | >300% | 280% | 222% | 79% |
| DRAM | >300% | >300% | >300% | >300% | >300% | >300% | >300% | 234% | 103% |
| HyperThread | 31% | 32% | 32% | 35% | 43% | 51% | 62% | 119% | 153% |
| CPU power | 277% | 294% | >300% | >300% | 224% | 252% | 193% | 167% | 82% |
| Network | 28% | 29% | 27% | >300% | >300% | >300% | >300% | >300% | >300% |
| brain | 232% | >300% | >300% | >300% | >300% | >300% | >300% | >300% | >300% |

Each entry is color-coded as follows: is $\geq$120%, is between 100% and 120%, and is $\leq$100%.

Fig. 1. Impact of interference on shared resources on websearch, ml_cluster, and memkeyval. Each row is an antagonist and each column is a load point for the workload. The values are latencies, normalized to the SLO latency. For the full table of data, refer to the Appendix.

the BE task creates sufficient pressure on shared resources to lead to SLO violations for all three workloads. A large contributor to this behavior is that the OS allows both workloads to run on the same core and even the same HyperThread, further compounding the interference. Tail latency eventually goes above 300% of SLO latency. Proposed interference-aware cluster managers, such as Paragon [Delimitrou and Kozyrakis 2013] and Bubble-Up [Mars et al. 2011], would disallow these colocations. To enable aggressive task colocation, not only do we need to disallow different workloads on the same core or HyperThread, we also need to use stronger isolation mechanisms.

The sensitivity of LC tasks to interference on individual shared resources varies. For instance, memkeyval is quite sensitive to network interference, while websearch and ml_cluster are not affected at all. The websearch workload is uniformly insensitive to small and medium amounts of LLC interference, while the same cannot be said for memkeyval or ml_cluster. Furthermore, the impact of interference changes depending on the load: ml_cluster can tolerate medium amounts of LLC interference at loads

<50% but is heavily impacted at higher loads. These observations motivate the need for dynamic management of isolation mechanisms in order to adapt to differences across varying loads and different workloads. Any static policy would be either too conservative (missing opportunities for colocation) or overly optimistic (leading to SLO violations).

We now discuss each LC workload separately, in order to understand their particular resource requirements.

**websearch:** This workload has a small footprint and LLC (small) and LLC (med) interference do not impact its tail latency. Nevertheless, the impact is significant with LLC (big) interference. The degradation is caused by two factors. First, the inclusive nature of the LLC in this particular chip means that high LLC interference leads to misses in the working set of instructions. Second, contention for the LLC causes significant DRAM pressure as well. The websearch workload is particularly sensitive to interference caused by DRAM bandwidth saturation. As the load of websearch increases, the impact of LLC and DRAM interference decreases. At higher loads, websearch uses more cores while the interference generator is given fewer cores. Thus, websearch can defend its share of resources better.

The websearch workload is moderately impacted by HyperThread interference until high loads. This indicates that the core has sufficient instruction issue bandwidth for both the spinloop and websearch until around 80% load. Since the spinloop only accesses registers, it does not cause interference in the L1 or L2 caches. However, since the HyperThread antagonist has the smallest possible effect, more intensive antagonists will cause far larger performance problems. Thus, HyperThread interference in practice should be avoided. Power interference has a significant impact on websearch at lower utilization, as more cores are executing the power virus. As expected, the network antagonist does not impact websearch, due to websearch's low bandwidth needs.

**ml_cluster:** ml_cluster is sensitive to LLC interference of smaller size, due to the small but significant per-request working set. This manifests itself as a large jump in latency at 75% load for LLC (small) and 50% load for LLC (medium). With larger LLC interference, ml_cluster experiences major latency degradation. The ml_cluster workload is also sensitive to DRAM bandwidth interference, primarily at lower loads (see explanation for websearch). The ml_cluster workload is moderately resistant to HyperThread interference until high loads, suggesting that it only reaches high instruction issue rates at high loads. Power interference has a lesser impact on ml_cluster since it is less compute intensive than websearch. Finally, ml_cluster is not impacted at all by network interference.

**memkeyval:** Due to its significantly stricter latency SLO, memkeyval is sensitive to all types of interference. At high load, memkeyval becomes sensitive even to small LLC interference as the small per-request working sets add up. When faced with medium LLC interference, there are two latency peaks. The first peak at low load is caused by the antagonist removing instructions from the cache. When memkeyval obtains enough cores at high load, it avoids these evictions. The second peak is at higher loads, when the antagonist interferes with the per-request working set. At high levels of LLC interference, memkeyval is unable to meet its SLO. Even though memkeyval has low DRAM bandwidth requirements, it is strongly affected by a DRAM streaming antagonist. Ironically, the few memory requests from memkeyval are overwhelmed by the DRAM antagonist.

The memkeyval workload is not sensitive to the HyperThread antagonist except at high loads. In contrast, it is very sensitive to the power antagonist, as it is compute-bound. The memkeyval workload does consume a large amount of network bandwidth, and thus is highly susceptible to competing network flows. Even at small loads, it is

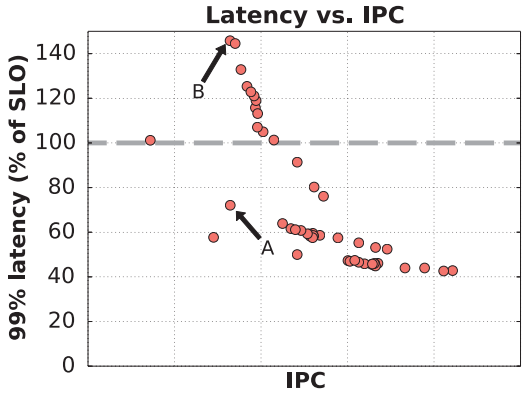



Fig. 2. Latency vs. IPC for an example LC job.

completely overrun by the many small "mice" flows of the antagonist and is unable to meet its SLO.

## 4. HERACLES DESIGN

We have established the need for isolation mechanisms beyond OS-level scheduling and for a dynamic controller that manages resource sharing between LC and BE tasks. Heracles is a dynamic, feedback-based controller that manages in real-time four hardware and software mechanisms in order to isolate colocated workloads. Heracles implements an *iso-latency* policy [Lo et al. 2014], namely that it can increase resource efficiency as long as the SLO is being met. This policy allows for increasing server utilization through tolerating some interference caused by colocation, as long as the the difference between the SLO latency target for the LC workload and the actual latency observed (latency slack) is positive. In its current version, Heracles manages one LC workload with many BE tasks. Since BE tasks are abundant, this approach is sufficient to raise utilization in many datacenters. We leave colocation of multiple LC workloads to future work.

### 4.1. Latency as the Primary Control Input

Heracles uses application-level latency from the LC workload to guide how it tunes various hardware/software isolation mechanisms to achieve iso-latency performance. This approach stands in contrast to several previously proposed systems for colocating workloads [Iyer et al. 2007; Yang et al. 2013; Mars et al. 2011; Zhang et al. 2013], which use throughput-based metrics such as Instructions Per Cycle (IPC) to guide their decision making. For workloads with performance measured by throughput, such as batch jobs, using IPC as a metric is a great way to directly measure how well an application is running. However, because IPC is a measure of throughput, it cannot capture how well an user-facing latency-critical application is meeting its target latency SLO.

To demonstrate this point, we measured 99%-ile latency and IPC for a production Google serving job. During the execution of the job, it was subjected to various sources of interference that caused both the IPC and the latency to diverge from when the serving job was running alone. We sampled both the IPC and the latency at various times and plot one versus the other as a scatter plot in Figure 2. We see that there exists a general trend that higher IPCs are less likely to lead to quality of service violations; however, given just an IPC value, there is no way of identifying whether or not the latency SLO is being violated. For example, consider the points denoted as *A* and *B* in Figure 2. Given the IPC at *A*, it is unclear if the latency is then *A* or *B*. The

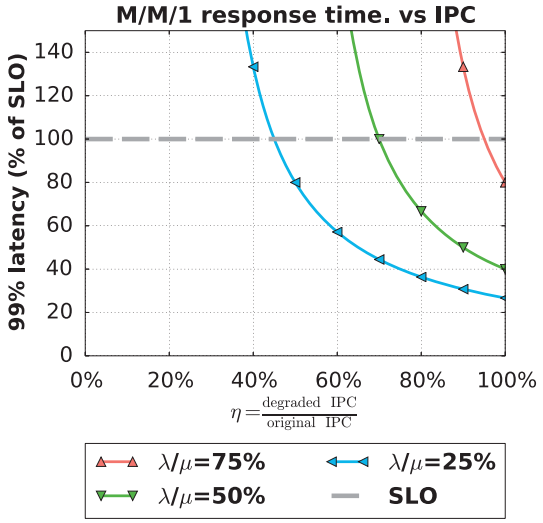



Fig. 3. Latency vs. IPC under different loads for example M/M/1 queue.

ambiguity can lead to wildly different conclusions, as $A$ is well under the latency SLO while $B$ is far above it.

Fundamentally, IPC cannot capture the latency behavior of an application. This fact can be explained using queueing theory. Consider an M/M/1 model, which models a system that has a single worker servicing a single queue where both the inter-arrival times and the service times are described by exponential distributions. While actual LC workloads are described by queueing systems more complicated than M/M/1 (e.g., multiple workers and queues, non-exponential inter-arrival and service times, etc.), the M/M/1 model is still a useful example for reasoning about why IPC is not a good predictor of latency.

In the M/M/1 example, we denote the peak service rate as $\mu$ and the average query arrival rate as $\lambda$. The 99%-ile tail latency can be analytically derived as Equation (1) [Gross 2008]:

$$r_{99} = \frac{1}{\mu - \lambda} \ln\left(\frac{100}{100 - 99}\right). \tag{1}$$

The effect that IPC has on tail latency is to reduce the peak service rate ($\mu$) of the system. For example, if IPC is degraded by 20% due to interference, then the peak amount of load the system can handle will also be lowered by 20%. Thus, to find the tail latency of an M/M/1 queue with an IPC degradation, we modify Equation (1) to get Equation (3):

$$\eta = \frac{\text{degraded IPC}}{\text{original IPC}}, \tag{2}$$

$$r_{99} = \frac{1}{\eta\mu - \lambda} \ln(100). \tag{3}$$

From here, the source of the ambiguity for latency and IPC is clear: latency is a two-dimensional function of $\eta$ and $\lambda$, which cannot be captured by a single variable. This can be seen in Figure 3, which shows the latency vs. IPC for an example M/M/1 queue at different loads. For any given IPC degradation, the latency could be one of infinitely many possibilities.

Thus, systems that only use IPC as the control metric must necessarily be conservative and keep the IPC of the LC task as high as possible. However, a conservative approach will not be able to exploit all possible colocation opportunities. For example, when the example M/M/1 system is at 25% load, it can afford to have its IPC degraded by up to 50% before a latency SLO violation occurs. Therefore, Heracles uses application-level latency as its primary feedback input, allowing Heracles to safely accommodate some IPC degradation while colocating workloads. This allows Heracles to realize additional colocations that were previously not allowed under an IPC-only control scheme, thereby increasing the utilization that Heracles can achieve under an iso-latency policy. In addition, by using latency that is directly measured from the LC task, Heracles sidesteps any accuracy issues that can arise from attempting to model the queueing behavior of the LC job.

### 4.2. Isolation Mechanisms

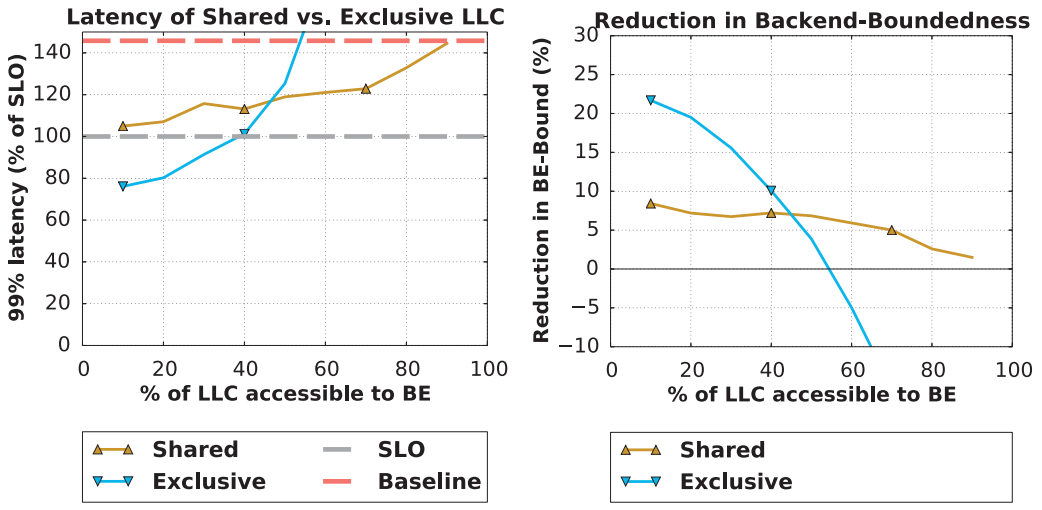
Heracles manages four mechanisms to mitigate interference. For *core isolation*, Heracles uses Linux's `cpuset cgroups` to pin the LC workload to one set of cores and BE tasks to another set (software mechanism) [Menage 2007]. This mechanism is necessary, since in Section 3 we showed that core sharing is detrimental to latency SLO. Moreover, the number of cores per server is increasing, making core segregation finer-grained. The allocation of cores to tasks is done dynamically. The speed of core (re-)allocation is limited by how fast Linux can migrate tasks to other cores, typically in the tens of milliseconds.

For *LLC isolation*, Heracles uses the Cache Allocation Technology (CAT) available in recent Intel chips (hardware mechanism) [Intel 2014]. CAT implements way-partitioning of the shared LLC. In a highly associative LLC, this allows us to define partitions at the granularity of a few percentages of the total LLC capacity. CAT can be configured to create non-overlapping or overlapping partitions, and in Section 4.3 we explore the differences between those two partitioning schemes. Ultimately, Heracles uses non-overlapping partitions, with one partition for the LC workload and a second partition for all BE tasks. Partition sizes can be adjusted dynamically by programming model specific registers (MSRs), with changes taking effect in a few milliseconds.

There are no commercially available DRAM bandwidth isolation mechanisms. We enforce *DRAM bandwidth limits* in the following manner: We implement a software monitor that periodically tracks the total bandwidth usage through performance counters and estimates the bandwidth used by the LC and BE jobs. If the LC workload does not receive sufficient bandwidth, then Heracles scales down the number of cores that BE jobs use. We discuss the limitations of this coarse-grained approach in Section 4.4.

For *power isolation*, Heracles uses hardware features, namely CPU frequency monitoring, Running Average Power Limit (RAPL), and per-core dynamic voltage frequency scaling (DVFS) [Intel 2014; Kim et al. 2008]. RAPL is used to monitor CPU power at the per-socket level, while per-core DVFS is used to redistribute power among cores. Per-core DVFS setting changes go into effect within a few milliseconds. The frequency steps are in 100MHz and span the entire operating frequency range of the processor, including Turbo Boost frequencies.

For *network traffic isolation*, Heracles uses Linux traffic control (software mechanism). Specifically we use the `qdisc` [Brown 2006] scheduler with hierarchical token bucket queueing discipline (HTB) to enforce bandwidth limits for outgoing traffic from the BE tasks. The bandwidth limits are set by limiting the maximum traffic burst rate for the BE jobs (`ceil` parameter in HTB parlance). The LC job does not have any limits set on it. HTB can be updated very frequently, with the new bandwidth limits taking effect in less than hundreds of milliseconds. Managing ingress network




(a) Latency (b) Backend-Boundedness

Fig. 4. Comparison of shared and exclusive LLC partitioning. Baseline is without any cache partitioning.

interference has been examined in numerous previous works and is outside the scope of this work [Jeyakumar et al. 2013].

### 4.3. LLC Partitioning Exploration

CAT allows LLC partitions to be either non-overlapping or overlapping. In the non-overlapping case, the last-level cache is hard partitioned into two segments: one partition that can only be used by the LC workload, and the other partition can only be used by the BE jobs (*exclusive*). In the overlapping case, the LLC is soft partitioned into a segment that only the LC job can access, and the rest of the cache can be accessed by either the LC or the BE jobs (*shared*). Shared partitioning guarantees the LC job a minimum amount of cache while allowing it to compete with the BE job for the rest of the LLC. Conversely, exclusive partitioning trades the opportunity to go beyond the guaranteed partition for more predictability, as the LLC behavior for the LC task is agnostic of the BE jobs. Intuitively, shared partitioning should have the potential to perform as well as, if not better than, exclusive partitioning for the same dedicated LC partition size. However, as we will see shortly, exclusive partitioning of the LLC turns out to be better than shared partitioning.

We ran an experiment to quantify the performance differences between the shared and exclusive LLC partitioning schemes. This experiment was conducted by isolating (to the best of our ability) LLC interference between a single LC workload and a BE job, using the same techniques as in Section 3. We varied the amount of cache accessible to the BE job for both partitioning schemes while keeping the QPS on the LC job constant. We then measured the resulting 99%-ile latency of the LC workload as well as the efficacy of using LLC partitioning. The latter metric is computed as the reduction in the number of CPU cycles that are stalled on the *backend* as computed using *TopDown* [Yasin 2014]. We compare both of these metrics to the baseline case of when both workloads were colocated without any cache partitioning, where both LC and BE jobs compete for the entire LLC.

Figure 4 shows the differences between the two different schemes for partitioning the LLC. First, we immediately observe that for this colocation scenario, the best LC latency in both cases occurs when almost all of the cache is dedicated to the LC

job, when the BE task can only access at most 10% of the LLC.[3] However, exclusive partitioning is able to achieve a much lower 99%-ile latency in this scenario compared to shared partitioning. This observation contradicts the intuitive expectation that shared partitioning should do no worse than exclusive partitioning. The reason behind this result is that in the shared case, both the LC and BE jobs are heavily contending in the shared region of the LLC, reducing the performance of the LC workload. This can be seen in Figure 4(b), where the reduction in backend stall cycles is far greater for the exclusive scheme than for the shared one. This demonstrates that the performance degradation due to contention in the shared region outweighs the benefits of allowing the LC workload to access the entire cache.

While exclusive partitioning allows for greater performance, it comes with a very significant downside: If misconfigured, exclusive partitioning can severely degrade performance. In the above experiment, when the LC job is not given enough cache, namely when BE is given more than 50% of the LLC, exclusive partitioning results in worse latency than shared partitioning. In the worst case, exclusive partitioning significantly degrades performance, causing latencies that exceed the latency of the baseline case of when no cache partitioning was used. Severe latency degradation from exclusive partitioning occurs when the LC partition does not fit the LC working set size, causing high numbers of misses to DRAM. Shared partitioning does not suffer from this effect, as in the worst case it has the same behavior as if no cache partitioning was used. Thus, shared partitioning trades performance gains for a *safer margin of error*.

In Heracles, we use exclusive LLC partitioning due to its notably better isolation guarantees. Indeed, in Figure 4(a), only exclusive partitioning can satisfy the latency SLO, while shared partitioning comes close but does not quite succeed. Nevertheless, from the above experiment it is clear that exclusive partitioning requires monitoring with a feedback loop to ensure that the LC partition is sufficiently large. We apply this lesson to the design of the sub-controller responsible for adjusting the LLC partitions in Section 4.5.

### 4.4. Design Approach

Each hardware or software isolation mechanism allows reasonably precise control of an individual resource. Given that, the controller must dynamically solve the high-dimensional problem of finding the right settings for all these mechanisms at any load for the LC workload and any set of BE tasks. Heracles solves this as an *optimization problem*, where the *objective* is to maximize utilization with the *constraint* that the SLO must be met.

Heracles reduces the optimization complexity by decoupling interference sources. The key insight that enables this reduction is that *interference is problematic only when a shared resource becomes saturated*, that is, its utilization is so high that latency problems occur. This insight is derived by the analysis in Section 3: The antagonists do not cause significant SLO violations until an inflection point, at which point the tail latency degrades extremely rapidly. Hence, if Heracles can prevent any shared resource from saturating, then *it can decompose the high-dimensional optimization problem into many smaller and independent problems of one or two dimensions each*. Then each sub-problem can be solved using sound optimization methods, such as gradient descent.

Since Heracles must ensure that the target SLO is met for the LC workload, it continuously monitors latency and latency slack and uses both as key inputs in its decisions. When the latency slack is large, Heracles treats this as a signal that it

[3]We note that the observation of better performance when dedicating almost all of the LLC to the LC task should not be interpreted as a general result, namely that for this particular set of workloads neither the LC nor the BE tasks are memory intensive.

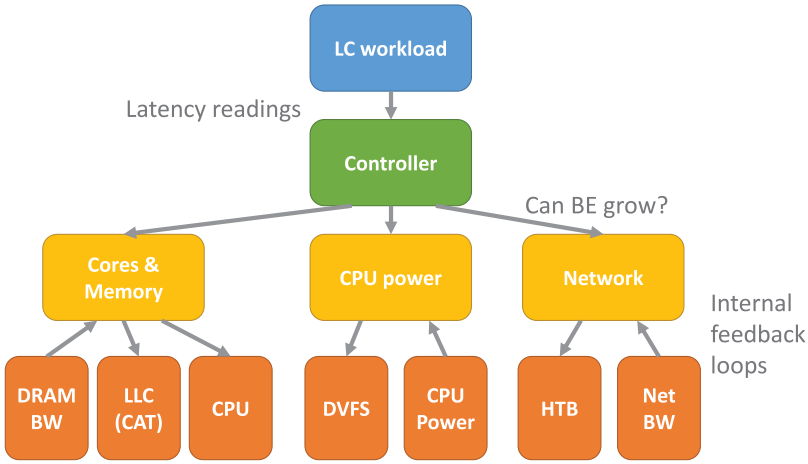



Fig. 5. The system diagram of Heracles.

is safe to be more aggressive with colocation; conversely, when the slack is small, it should back off to avoid an SLO violation. Heracles also monitors the load (queries per second), and, during periods of high load, it disables colocation due to a high risk of SLO violations. Previous work has shown that indirect performance metrics, such as CPU utilization, are insufficient to guarantee that the SLO is met [Lo et al. 2014].

Ideally, Heracles should require no offline information other than SLO targets. Unfortunately, one shortcoming of current hardware makes this difficult. The Intel chips we used do not provide accurate mechanisms for measuring (or limiting) DRAM bandwidth usage at a per-core granularity. To understand how Heracles' decisions affect the DRAM bandwidth usage of latency-sensitive and BE tasks and to manage bandwidth saturation, we require some offline information. Specifically, Heracles uses an offline model that describes the DRAM bandwidth used by the latency-sensitive workloads at various loads, core, and LLC allocations. We verified that this model needs to be regenerated only when there are significant changes in the workload structure and that small deviations are fine. There is no need for any offline profiling of the BE tasks, which can vary widely compared to the better managed and understood LC workloads. There is also no need for offline analysis of interactions between latency-sensitive and best-effort tasks. Once we have hardware support for per-core DRAM bandwidth accounting [Iyer et al. 2007], we can eliminate this offline model.

### 4.5. Heracles Controller

Heracles runs as a separate instance on each server, managing the local interactions between the LC and BE jobs. As shown in Figure 5, it is organized as three subcontrollers (cores and memory, power, network traffic) coordinated by a top-level controller. The subcontrollers operate fairly independently of each other and ensure that their respective shared resources are not saturated.

**Top-level controller:** The pseudo-code for the controller is shown in Algorithm 1. The controller polls the tail latency and load of the LC workload every 15s. Waiting for this amount of time allows for a sufficient number of queries to be serviced in order to calculate statistically meaningful tail latencies. The polling interval is dependent on the lowest QPS expected to be served and the percentile tail latency to be measured. A good rule of thumb is to ensure that there are at least 100 queries sampled, namely that $Interval \times QPS \times (1 - \frac{Percentile}{100}) > 100$. If the load for the LC workload exceeds 85% of

**ALGORITHM 1:** High-level Controller

```
1  while True :
2  |  latency = PollLCAppLatency()
3  |  load = PollLCAppLoad()
4  |  slack = (target - latency)/target
5  |  if slack < 0 :
6  |  |  DisableBE()
7  |  |  EnterCooldown()
8  |  elif load > 0.85 :
9  |  |  DisableBE()
10 |  elif load < 0.80 :
11 |  |  EnableBE()
12 |  elif slack < 0.10 :
13 |  |  DisallowBEGrowth()
14 |  |  if slack < 0.05 :
15 |  |  |  be_cores.Remove(be_cores.Size() - 2)
16 |  sleep(15)
```

its peak on the server, then the controller disables the execution of BE workloads. This empirical safeguard avoids the difficulties of latency management on highly utilized systems for minor gains in utilization. For hysteresis purposes, BE execution is enabled when the load drops below 80%. BE execution is also disabled when the latency slack, the difference between the SLO target and the current measured tail latency, is negative. A negative latency slack typically occurs when there is a sharp spike in load for the latency-sensitive workload. We give all resources to the latency-critical workload for a while (e.g., 5 minutes) before attempting colocation again. The constants used in the controller were determined through empirical tuning. The threshold for peak load (85%) can be set higher for a controller that is more aggressive at raising utilization or lower for a more conservative scheme.

When these two safeguards are not active, the controller uses slack to guide the subcontrollers in providing resources to BE tasks. If slack is less than 10%, then the subcontrollers are instructed to disallow growth for BE tasks in order to maintain a safety margin. If slack drops below 5%, then the subcontroller for cores is instructed to switch cores from BE tasks to the LC workload. Reassigning CPUs to the LC task improves the latency of the LC workload and reduces the ability of the BE job to cause interference on any resources. If slack is above 10%, then the subcontrollers are instructed to allow BE tasks to acquire a larger share of system resources. Each subcontroller makes allocation decisions independently, provided of course that its resources are not saturated. The latency slack margins are also user tunables, where a higher latency slack leads to a more conservative controller, while too low of a margin (<5% for when BE tasks are no longer allowed to grow) can lead to control instability due to inherent variations in application behavior. The 10% threshold was set as a compromise to achieve both high utilization and control stability.

**Core and memory subcontroller:** Heracles uses a single subcontroller for core and cache allocation due to the strong coupling among core count, LLC needs, and memory bandwidth needs. If there was a direct way to isolate memory bandwidth, then we would use independent controllers. The pseudo-code for this subcontroller is shown in Algorithm 2. Its output is the allocation of cores and LLC to the LC and BE jobs (two dimensions).

The first constraint for the subcontroller is to avoid memory bandwidth saturation. The DRAM controllers provide registers that track bandwidth usage, making it easy to detect when they reach 90% of peak streaming DRAM bandwidth. In this case, the

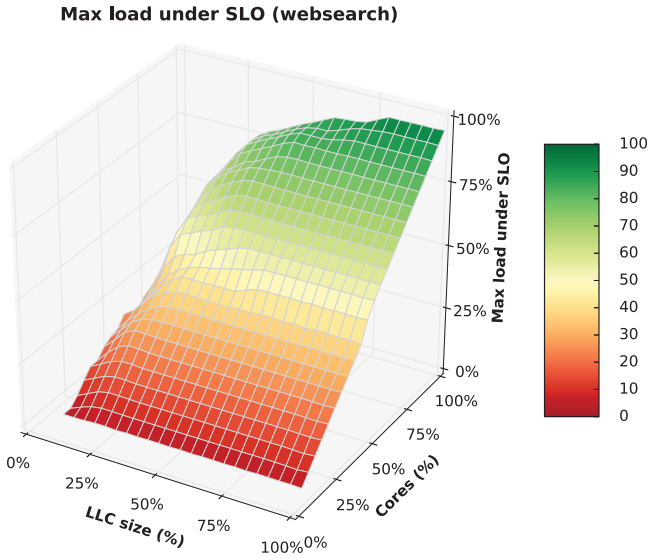



Fig. 6. Characterization of websearch showing that its performance is a convex function of cores and LLC.

subcontroller removes as many cores as needed from BE tasks to avoid saturation. Heracles estimates the bandwidth usage of each BE task using a model of bandwidth needs for the LC workload and a set of hardware counters that are proportional to the per-core memory traffic to the NUMA-local memory controllers. For the latter counters to be useful, we limit each BE task to a single socket for both cores and memory allocations using Linux `numactl`. Different BE jobs can run on either socket and LC workloads can span across sockets for cores and memory.

When the top-level controller signals BE growth and there is no DRAM bandwidth saturation, the subcontroller uses gradient descent to find the maximum number of cores and cache partitions that can be given to BE tasks. Gradient descent is an iterative algorithm that finds a local optimum by calculating the gradient of a function and then taking a step in the direction of the gradient. This process is represented mathematically as

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n). \tag{4}$$

Applied to the core and cache allocation problem, $\mathbf{x} = (\text{Cores}, \text{Cache Size})$, $F$ is the performance of the BE task, and $\gamma$ is the step size (which can change every iteration). Since the BE task is a batch workload, we can use throughput based metrics for $F$. Gradient descent terminates when a convergence criterion is satisfied, which occurs when no more cores or cache can be transferred from the LC task to the BE jobs. Gradient descent is guaranteed to converge to a global optimum when the $F$ is a convex function, as local and global optima will have the same values [Boyd and Vandenberghe 2004].

Offline analysis of both BE and LC applications (Figure 6) shows that their performance is a convex function of core and cache resources, thus guaranteeing that gradient descent will find a global optimum. Since both the number of cores and the LLC partition sizes are quantized, we use a quantized approximation to gradient descent to find an allocation of the cores and cache that is sufficiently close to the optimal allocation. Finding the optimal allocation is done by performing gradient descent in one dimension at a time, switching between increasing the cores, and increasing the cache given to BE tasks. Initially, a BE job is given one core and 10% of the LLC and starts in the

**ALGORITHM 2:** Core & Memory Sub-Controller

```
def PredictedTotalBW() :
    return LcBwModel() + BeBw() + bw_derivative
while True :
    MeasureDRAMBw()
    if total_bw > DRAM_LIMIT :
        overage = total_bw - DRAM_LIMIT
        be_cores.Remove(overage/BeBwPerCore())
        continue
    if not CanGrowBE() :
        continue
    if state == GROW_LLC :
        if PredictedTotalBW() > DRAM_LIMIT :
            state = GROW_CORES
        else:
            GrowCacheForBE()
            MeasureDRAMBw()
            if bw_derivative ≥ 0 :
                Rollback()
                state = GROW_CORES
            if not BeBenefit() :
                state = GROW_CORES
    elif state == GROW_CORES :
        needed = LcBwModel() + BeBw() + BeBwPerCore()
        if needed > DRAM_LIMIT :
            state = GROW_LLC
        else:
            be_cores.Add(1)
    sleep(2)
```

*GROW_LLC* phase. Its LLC allocation is increased as long as the LC workload meets its SLO, bandwidth saturation is avoided, and the BE task benefits. The rationale for starting the BE job with a small LLC partition and then slowly growing it is to avoid the performance cliff that we observed earlier in Section 4.3 for exclusive LLC partitioning schemes. The next phase (*GROW_CORES*) grows the number of cores for the BE job. Heracles will reassign cores from the LC to the BE job one at a time, each time checking for DRAM bandwidth saturation and SLO violations for the LC workload. If bandwidth saturation occurs first, then the subcontroller will return to the *GROW_LLC* phase. The process repeats until an optimal configuration has been converged on. The search also terminates on a signal from the top-level controller indicating the end to growth or the disabling of BE jobs. The cycle time between each iteration is 2s, to allow for DRAM bandwidth to settle and to be measured with less noise. With a 2s cycle time, the typical amount of time to convergence is about 30s.

During gradient descent, the subcontroller must avoid trying suboptimal allocations that will either trigger DRAM bandwidth saturation or a signal from the top-level controller to disable BE tasks. To estimate the DRAM bandwidth usage of an allocation prior to trying it, the subcontroller uses the derivative of the DRAM bandwidth from the last reallocation of cache or cores. Heracles estimates whether it is close to an SLO violation for the LC task based on the amount of latency slack.

**Power subcontroller:** The simple subcontroller described in Algorithm 3 ensures that there is sufficient power slack to run the LC workload at a minimum guaranteed frequency. This frequency is determined by measuring the frequency used when the

**ALGORITHM 3:** CPU Power Sub-Controller

```
1 while True :
2 |   power = PollRAPL()
3 |   ls_freq = PollFrequency(ls_cores)
4 |   if power > 0.90×TDP and ls_freq < guaranteed :
5 |   |   LowerFrequency(be_cores)
6 |   elif power ≤ 0.90×TDP and ls_freq ≥ guaranteed :
7 |   |   IncreaseFrequency(be_cores)
8 |   sleep(2)
```

**ALGORITHM 4:** Network Sub-Controller

```
1 while True :
2 |   ls_bw = GetLCTxBandwidth()
3 |   be_bw = LINK_RATE - ls_bw - max(0.05×LINK_RATE, 0.10×ls_bw)
4 |   SetBETxBandwidth(be_bw)
5 |   sleep(1)
```

LC workload runs alone at full load. Heracles uses RAPL to determine the operating power of the CPU and its maximum design power, or thermal dissipation power (TDP). It also uses CPU frequency monitoring facilities on each core. When the operating power is close to the TDP *and* the frequency of the cores running the LC workload is too low, it uses per-core DVFS to lower the frequency of cores running BE tasks in order to shift the power budget to cores running LC tasks. Both conditions must be met in order to avoid confusion when the LC cores enter active-idle modes, which also tends to lower frequency readings. If there is sufficient operating power headroom, Heracles will increase the frequency limit for the BE cores in order to maximize their performance. The control loop runs independently for each of the two sockets and has a cycle time of 2s in order to allow the CPU frequency and average power measurements to settle.

**Network subcontroller:** This subcontroller prevents saturation of network transmit bandwidth as shown in Algorithm 4. It monitors the total egress bandwidth of flows associated with the LC workload ($LC\,Bandwidth$) and sets the total bandwidth limit of all other flows as $LinkRate - LC\,Bandwidth - max(0.05 LinkRate, 0.10 LC\,Bandwidth)$. A small headroom of 10% of the current $LC\,Bandwidth$ or 5% of the LinkRate is added into the reservation for the LC workload in order to handle spikes. The bandwidth limit is enforced via HTB qdiscs in the Linux kernel. This control loop is run once every second, which provides sufficient time for the bandwidth enforcer to settle.

## 5. HERACLES EVALUATION

### 5.1. Methodology

We evaluated Heracles with the three production, latency-critical workloads from Google analyzed in Section 3. We first performed experiments with Heracles on a single leaf server, introducing BE tasks as we operated the LC workload at different levels of load. Next, we used Heracles on a websearch cluster with tens of servers, measuring end-to-end workload latency across the fan-out tree while BE tasks are also running. In the cluster experiments, we used an anonymized websearch trace that captures both the different kinds of queries as well as diurnal load variation. In all cases, we used Google servers deployed in a production environment.

For the LC workloads we focus on SLO latency. Since the SLO is defined over 60s windows, we report the worst-case latency that was seen during experiments. For the

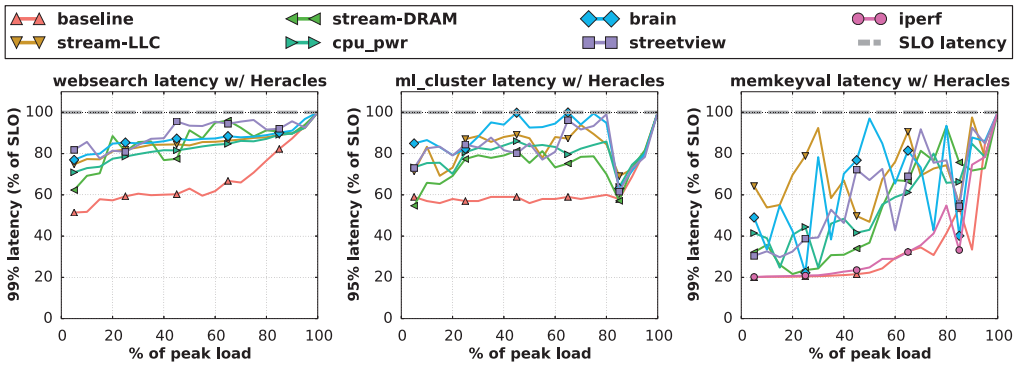



Fig. 7. Latency of LC applications colocated with BE jobs under Heracles. For clarity we omit websearch and ml_cluster with iperf as those workloads are extremely resistant to network interference.

production batch workloads, we compute the throughput rate of the batch workload with Heracles and normalize it to the throughput of the batch workload running alone on a single server. We then define the *Effective Machine Utilization (EMU)* = LC Throughput + BE Throughput. Note that Effective Machine Utilization can be above 100% due to better binpacking of shared resources. We also report the utilization of shared resources when necessary to highlight detailed aspects of the system.

The BE workloads we use are chosen from a set containing both production batch workloads and the synthetic tasks that stress a single shared resource. The specific workloads are as follow:

—*stream-LLC* streams through data sized to fit in about half of the LLC and is the same as LLC (med) from Section 3.2. *stream-DRAM* streams through an extremely large array that cannot fit in the LLC (DRAM from the same section). We use these workloads to verify that Heracles is able to maximize the use of LLC partitions and avoid DRAM bandwidth saturation.
—*cpu_pwr* is the CPU power virus from Section 3.2. It is used to verify that Heracles will redistribute power to ensure that the LC workload maintains its guaranteed frequency.
—*iperf* is an open-source network streaming benchmark used to verify that Heracles partitions network transmit bandwidth correctly to protect the LC workload.
—*brain* is a Google production batch workload that performs deep learning on images for automatic labelling [Le et al. 2012; Rosenberg 2013]. This workload is very computationally intensive, is sensitive to LLC size, and also has high DRAM bandwidth requirements.
—*streetview* is a production batch job that stitches together multiple images to form the panoramas for Google Street View. This workload is highly demanding on the DRAM subsystem.

### 5.2. Individual Server Results

**Latency SLO:** Figure 7 presents the impact of colocating each of the three LC workloads with BE workloads across all possible loads under the control of Heracles. Note that Heracles attempts to run as many copies of the BE task as possible and maximize the resources they receive. At all loads and in all colocation cases, there are *no SLO violations* with Heracles. Specifically, Heracles is able to protect the LC workload from interference caused by brain, a workload that even with the state-of-the-art OS isolation mechanisms would render any LC workload unusable. This validates that the

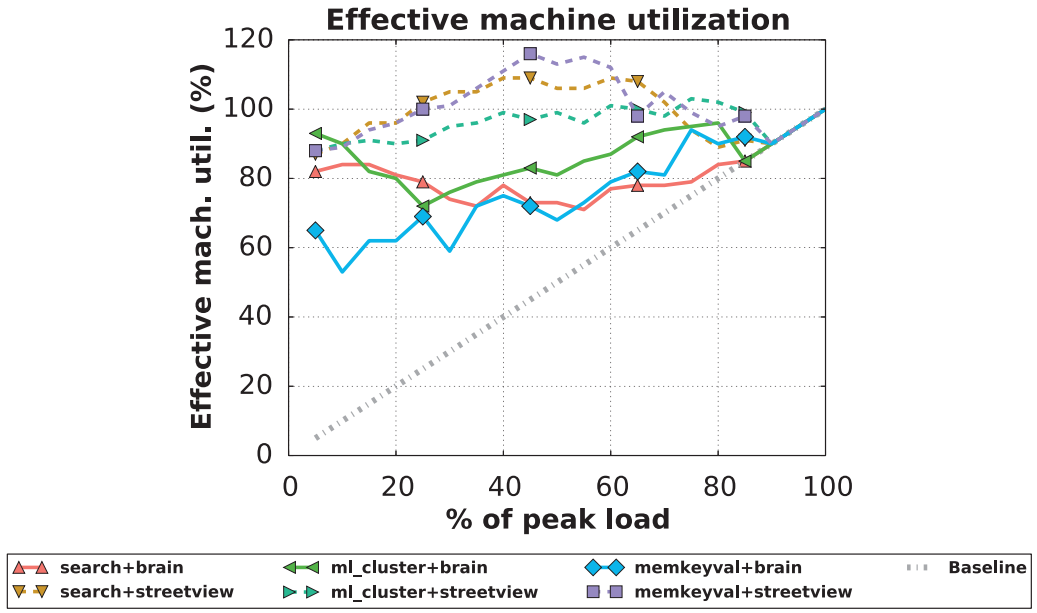



Fig. 8. EMU achieved by Heracles.

controller keeps shared resources from saturating and allocates a sufficient fraction to the LC workload at any load. Heracles maintains a small latency slack as a guard band to avoid spikes and control instability. It also validates that local information on tail latency is sufficient for stable control for applications with milliseconds and microseconds range of SLOs. Interestingly, the websearch binary and shard changed between generating the offline profiling model for DRAM bandwidth and performing this experiment. Nevertheless, Heracles is resilient to these changes and performs well despite the somewhat outdated model.

Heracles reduces the latency slack during periods of low utilization for all workloads. For websearch and ml_cluster, the slack is cut in half, from 40% to 20%. For memkeyval, the reduction is much more dramatic, from a slack of 80% to 40% or less. The larger reduction in slack for memkeyval is because the unloaded latency of memkeyval is extremely small compared to the SLO latency. The high variance of the tail latency for memkeyval is due to the fact that its SLO is in the hundreds of microseconds, making it more sensitive to interference than the other two workloads.

**Server Utilization:** Figure 8 shows the EMU achieved when colocating production LC and BE tasks with Heracles. In all cases, we achieve significant EMU increases. When the two most CPU-intensive and power-hungry workloads are combined, websearch and brain, Heracles still achieves an EMU of at least 75%. When websearch is combined with the DRAM bandwidth intensive streetview, Heracles can extract sufficient resources for a total EMU above 100% at websearch loads between 25% and 70%. Heracles is able to achieve an EMU over 100% because websearch and streetivew have complementary resource requirements, where websearch is more compute bound and streetview is more DRAM bandwidth bound. The EMU results are similarly positive for ml_cluster and memkeyval. By dynamically managing multiple isolation mechanisms, Heracles exposes opportunities to raise EMU that would otherwise be missed with scheduling techniques that avoid interference.

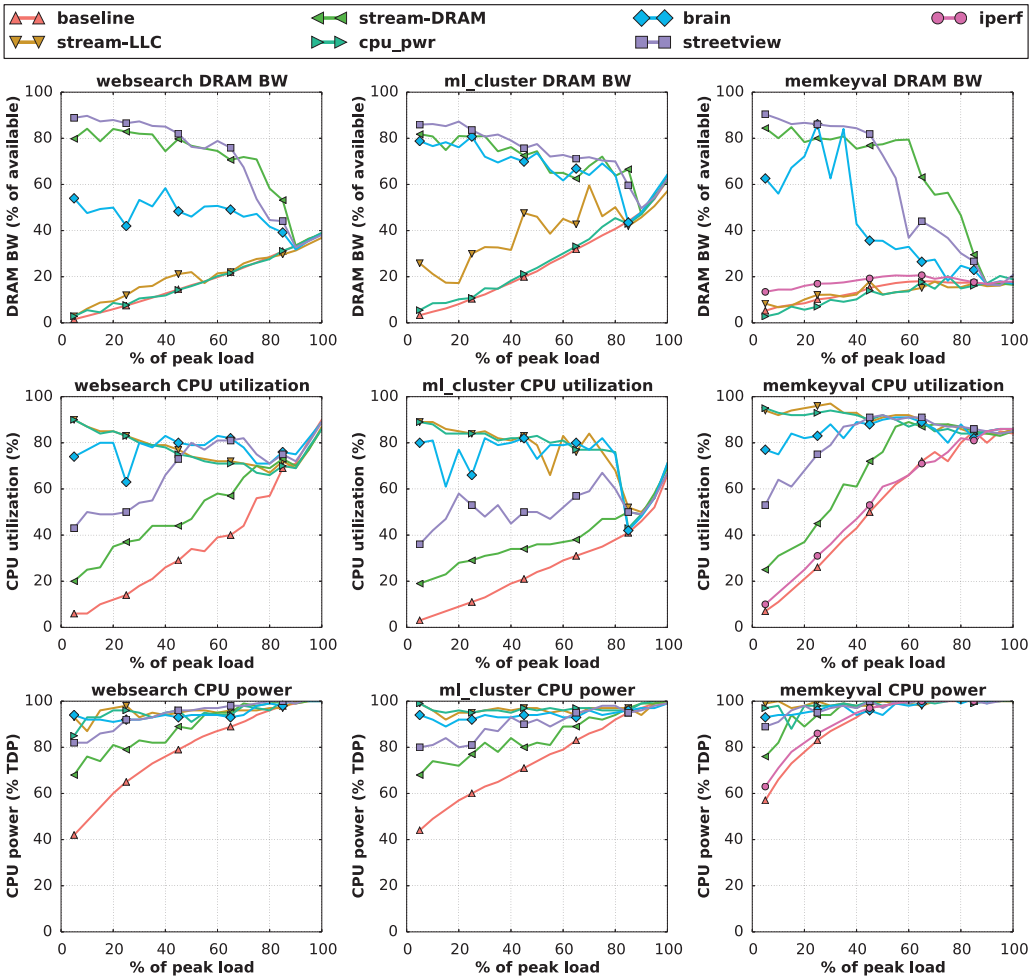



Fig. 9. Various system utilization metrics of LC applications colocated with BE jobs under Heracles.

**Shared Resource Utilization:** Figure 9 plots the utilization of shared resources (cores, power, and DRAM bandwidth) under Heracles control. For memkeyval, we include measurements of network transmit bandwidth in Figure 10.

Across the board, Heracles is able to correctly size the BE workloads to avoid saturating DRAM bandwidth. For the stream-LLC BE task, Heracles finds the correct cache partitions to decrease total DRAM bandwidth requirements for all workloads. For ml_cluster, with its large cache footprint, Heracles balances the needs of stream-LLC with ml_cluster effectively, with a total DRAM bandwidth slightly above the baseline. For the BE tasks with high DRAM requirements (stream-DRAM, streetview), Heracles only allows them to execute on a few cores to avoid saturating DRAM. As a result, these workload combinations have lower CPU utilization but very high DRAM bandwidth. However, EMU is still high, as the critical resource for those workloads is not compute but memory bandwidth.

Looking at the power utilization, Heracles allows significant improvements to energy efficiency. Consider the 20% load case: EMU was raised by a significant amount, from

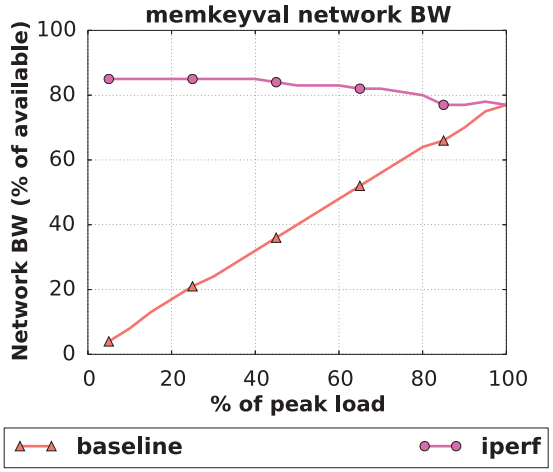



Fig. 10. Network bandwidth of memkeyval under Heracles.

20% to 60%–90%. However, the CPU power only increased from 60% to 80%, leading to an energy efficiency gain of 2.3–3.4x. Overall, Heracles achieves significant gains in resource efficiency across all loads for the LC task without causing SLO violations.

### 5.3. Websearch Cluster Results

We also evaluate Heracles on a small minicluster for websearch with tens of servers as a proxy for the full-scale cluster. The cluster root fans out each user request to all leaf servers and combines their replies. The SLO latency is defined as the average latency at the root over 30s, denoted as $\mu$/30s. The target SLO latency is set as $\mu$/30s when serving 90% load in the cluster without colocated tasks. Heracles runs on every leaf node with a uniform 99%-ile latency target set such that the latency at the root satisfies the SLO. We use Heracles to execute brain on half of the leafs and streetview on the other half. Heracles shares the same offline model for the DRAM bandwidth needs of websearch across all leaves, even though each leaf has a different shard. We generate websearch load from an anonymized, 12-hour request trace that captures the part of the daily diurnal pattern when websearch is not fully loaded and colocation has high potential.

**Latency SLO:** Figure 11 shows the latency SLO with and without Heracles for the 12-hour trace. Heracles produces no SLO violations while reducing slack by 20%–30%. Meeting the 99%-ile tail latency at each leaf is sufficient to guarantee the global SLO. We believe we can further reduce the slack in larger websearch clusters by introducing a centralized controller that dynamically sets the per-leaf tail latency targets based on slack at the root [Lo et al. 2014]. Having a centralized controller would allow a future version of Heracles to take advantage of slack in higher layers of the fan-out tree.

**Server Utilization:** Figure 11 also shows that Heracles successfully converts the latency slack in the baseline case into significantly increased EMU. Throughout the trace, Heracles colocates sufficient BE tasks to maintain an average EMU of 90% and a minimum of 80% without causing SLO violations. The websearch load varies between 20% and 90% in this trace.

**TCO:** To estimate the impact on total cost of ownership, we use the TCO calculator by Barroso et al. with the parameters from the case-study of a datacenter with low

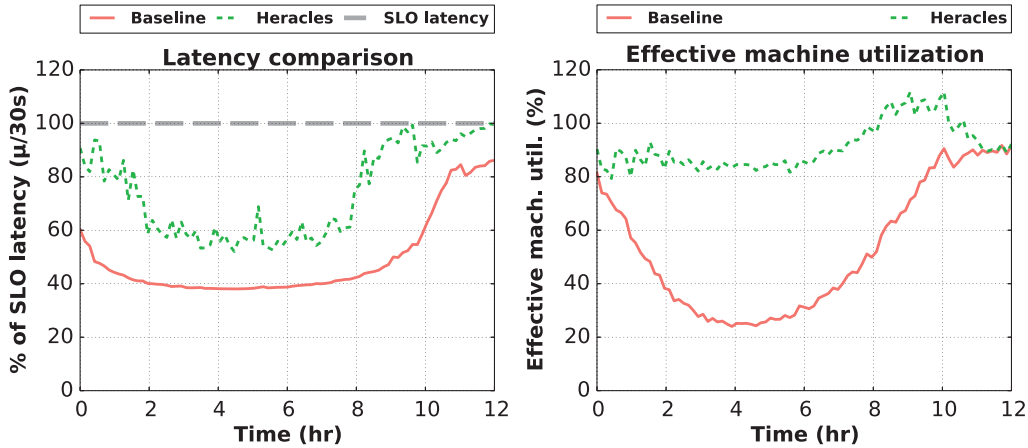



Fig. 11. Latency SLO and effective machine utilization for a websearch cluster managed by Heracles.

per-server cost [Barroso et al. 2013]. This model assumes $2,000 servers with a PUE of 2.0 and a peak power draw of 500W as well as electricity costs of $0.10/kW-hr. For our calculations, we assume a cluster size of 10,000 servers. Assuming pessimistically that a websearch cluster is highly utilized throughout the day, with an average load of 75%, Heracles' ability to raise utilization to 90% translates to a 15% throughput/TCO improvement over the baseline. This improvement includes the cost of the additional power consumption at higher utilization. Under the same assumptions, a controller that focuses only on improving energy-proportionality for websearch would achieve throughput/TCO gains of roughly 3% [Lo et al. 2014].

If we assume a cluster for LC workloads utilized at an average of 20%, as many industry studies suggest [Liu 2011; Vasan et al. 2010], then Heracles can achieve a 306% increase in throughput/TCO. A controller focusing on energy proportionality would achieve improvements of less than 7%. Heracles' advantage is due to the fact that it can raise utilization from 20% to 90% with a small increase to power consumption, which only represents 9% of the initial TCO. As long as there are useful BE tasks available, one should always choose to improve throughput/TCO by colocating them with LC jobs instead of lowering the power consumption of servers in modern datacenters. Also note that the improvements in throughput/TCO are large enough to offset the cost of reserving a small portion of each server's memory or storage for BE tasks.

## 6. RELATED WORK

**Isolation mechanisms:** There is significant work on shared cache isolation, including soft partitioning based on replacement policies [Wu and Martonosi 2008; Xie and Loh 2009], way partitioning [Ranganathan et al. 2000; Qureshi and Patt 2006], and fine-grained partitioning [Sanchez and Kozyrakis 2011; Manikantan et al. 2012; Srikantaiah et al. 2009]. Tessellation exposes an interface for throughput-based applications to request partitioned resources [Liu et al. 2009]. Most cache partitioning schemes have been evaluated with a utility-based policy that optimizes for aggregate throughput [Qureshi and Patt 2006]. Heracles manages the coarse-grained, way-partitioning scheme recently added in Intel CPUs, using a search for a right-sized allocation to eliminate latency SLO violations. We expect Heracles will work even better with fine-grained partitioning schemes when they are commercially available.

Iyer et al. explores a wide range quality-of-service (QoS) policies for shared cache and memory systems with simulated isolation features [Iyer et al. 2007; Hsu et al. 2006; Guo et al. 2007a, 2007b; Iyer 2004]. They focus on throughput metrics, such as IPC and Misses Per Instruction (MPI), and did not consider latency-critical workloads or other resources such as network traffic. Cook et al. evaluate hardware cache partitioning for throughput based applications and did not consider latency-critical tasks [Cook et al. 2013]. Wu et al. compare different capacity management schemes for shared caches [Wu and Martonosi 2008]. The proposed Ubik controller for shared caches with fine-grained partitioning support boosts the allocation for latency-critical workloads during load transition times and requires application level changes to inform the runtime of load changes [Kasture and Sanchez 2014]. Heracles does not require any changes to the LC task, instead relying on a steady-state approach for managing cache partitions that changes partition sizes slowly.

There are several proposals for isolation and QoS features for memory controllers [Iyer et al. 2007; Muralidhara et al. 2011; Jeong et al. 2012; Nesbit et al. 2006; Nagarajan and Gupta 2009; Ebrahimi et al. 2010; Li et al. 2011; Sharifi et al. 2011]. While our work showcases the need for memory isolation for latency-critical workloads, such features are not commercially available at this point. Several network interface controllers implement bandwidth limiters and priority mechanisms in hardware. Unfortunately, these features are not exposed by device drivers. Hence, Heracles and related projects in network performance isolation currently use Linux qdisc [Jeyakumar et al. 2013]. Support for network isolation in hardware should strengthen this work.

The LC workloads we evaluated do not use disks or SSDs in order to meet their aggressive latency targets. Nevertheless, disk and SSD isolation is quite similar to network isolation. Thus, the same principles and controls used to mitigate network interference still apply. For disks, we list several available isolation techniques: (1) the cgroups blkio controller [Menage 2007], (2) native command queuing (NCQ) priorities [Intel 2003], (3) prioritization in file-system queues, (4) partitioning LC and BE to different disks, and (5) replicating LC data across multiple disks that allows selecting the disk/reply that responds first or has lower load [Dean and Barroso 2013]. For SSDs: (1) many SSDs support channel partitions, separate queueing, and prioritization at the queue level, and (2) SSDs also support suspending operations to allow LC requests to overtake BE requests. The techniques mentioned above have been employed in other solutions to achieve performance isolation for storage and I/O devices. Prioritizing requests into multiple queues is used by IOFlow to achieve end-to-end performance isolation for virtualized storage [Thereska et al. 2013]. The IRIX operating system uses a mechanism similar to the blkio cgroup in order to partition disk bandwidth [Verghese et al. 1998].

**Interference-aware cluster management:** Several cluster-management systems detect interference between colocated workloads and generate schedules that avoid problematic colocations. Nathuji et al. develop a feedback-based scheme that tunes resource assignment to mitigate interference for colocated Virtual Machines (VMs) [Nathuji et al. 2010]. Bubble-flux is an online scheme that detects memory pressure and finds colocations that avoid interference on latency-sensitive workloads [Yang et al. 2013; Mars et al. 2011]. Bubble-flux has a backup mechanism to enable problematic colocations via execution modulation, but such a mechanism would have challenges with applications such as memkeyval, as the modulation would need to be done in the granularity of microseconds. DeepDive detects and manages interference between co-scheduled applications in a VM system [Novakovic et al. 2013]. CPI2 throttles low-priority workloads that interfere with important services [Zhang et al. 2013]. Finally,

Paragon and Quasar use online classification to estimate interference and to colocate workloads that are unlikely to cause interference [Delimitrou and Kozyrakis 2013, 2014].

The primary difference of Heracles is the focus on latency-critical workloads and the use of multiple isolation schemes in order to allow aggressive colocation without SLO violations at scale. Many previous approaches use IPC instead of latency as the performance metric [Yang et al. 2013; Mars et al. 2011; Novakovic et al. 2013; Zhang et al. 2013]. Nevertheless, one can couple Heracles with an interference-aware cluster manager in order to optimize the placement of BE tasks.

**Latency-critical workloads:** There is also significant work in optimizing various aspects of latency-critical workloads, including energy proportionality [Meisner et al. 2009, 2011; Lo et al. 2014; Liu et al. 2014; Kanev et al. 2014], networking performance [Kapoor et al. 2012; Belay et al. 2014], and hardware-acceleration [Lim et al. 2013; Putnam et al. 2014; Tanaka and Kozyrakis 2014]. Heracles is largely orthogonal to these projects.

## 7. CONCLUSIONS

We present *Heracles*, a heuristic feedback-based system that manages four isolation mechanisms to enable a latency-critical workload to be colocated with batch jobs without SLO violations. We evaluated Heracles on several latency-critical and batch workloads used in production at Google on real hardware and demonstrated an average utilization of 90% across all evaluated scenarios without any SLO violations for the latency-critical job. Through coordinated management of several isolation mechanisms, Heracles enables colocation of tasks that previously would cause SLO violations. Compared to power-saving mechanisms alone, Heracles increases overall cost efficiency substantially through increased utilization.

We learned several lessons while developing and implementing Heracles that were key to achieving high utilization while maintaining latency SLOs: (1) An iso-latency policy using application level performance information partitions resources more efficiently than a policy that seeks to maximize the IPC of the latency-critical task. (2) Thorough empirical characterization of several Google latency critical workloads demonstrated that shared resource saturation is the main culprit behind significant performance degradation in highly utilized systems. (3) Conversely, preventing resource saturation allows for high utilization while maintaining latency SLO targets, greatly simplifying the optimization problem.

There are several directions for future work. One such direction is to extend Heracles with support for additional shared resources, such as disks and SSDs. Another direction is to integrate Heracles more deeply with the applications it is managing. Giving Heracles more information about an application's current behavior, such as task queue length and profiling data at the function level, could enable a more dynamic and finer-grained control scheme.

## APPENDIX

Here we include the full data for the characterization of interference for different latency-critical workloads from various interference sources (Figure 1).

**websearch**

| Load | LLC (small) | LLC (med) | LLC (big) | DRAM | HyperThread | CPU power | Network | brain |
|---|---|---|---|---|---|---|---|---|
| 5% | 134% | 152% | >300% | >300% | 81% | 190% | 35% | 158% |
| 10% | 103% | 106% | >300% | >300% | 109% | 124% | 35% | 165% |
| 15% | 96% | 99% | >300% | >300% | 106% | 110% | 36% | 157% |
| 20% | 96% | 99% | >300% | >300% | 106% | 107% | 36% | 173% |
| 25% | 109% | 116% | >300% | >300% | 104% | 134% | 36% | 160% |
| 30% | 102% | 111% | >300% | >300% | 113% | 115% | 36% | 168% |
| 35% | 100% | 109% | >300% | >300% | 106% | 106% | 36% | 180% |
| 40% | 96% | 103% | >300% | >300% | 114% | 108% | 37% | 230% |
| 45% | 96% | 105% | >300% | >300% | 113% | 102% | 37% | >300% |
| 50% | 104% | 116% | >300% | >300% | 105% | 114% | 38% | >300% |
| 55% | 99% | 109% | >300% | >300% | 114% | 107% | 39% | >300% |
| 60% | 100% | 108% | >300% | >300% | 117% | 105% | 41% | >300% |
| 65% | 101% | 107% | >300% | >300% | 118% | 104% | 44% | >300% |
| 70% | 100% | 110% | >300% | >300% | 119% | 101% | 48% | >300% |
| 75% | 104% | 123% | >300% | >300% | 122% | 105% | 51% | >300% |
| 80% | 103% | 125% | 264% | 270% | 136% | 100% | 55% | >300% |
| 85% | 104% | 114% | 222% | 228% | >300% | 98% | 58% | >300% |
| 90% | 103% | 111% | 123% | 122% | >300% | 99% | 64% | >300% |
| 95% | 99% | 101% | 102% | 103% | >300% | 97% | 95% | >300% |

**ml_cluster**

| Load | LLC (small) | LLC (med) | LLC (big) | DRAM | HyperThread | CPU power | Network | brain |
|---|---|---|---|---|---|---|---|---|
| 5% | 101% | 98% | >300% | >300% | 113% | 112% | 57% | 151% |
| 10% | 88% | 88% | >300% | >300% | 109% | 101% | 56% | 149% |
| 15% | 99% | 102% | >300% | >300% | 110% | 97% | 58% | 174% |
| 20% | 84% | 91% | >300% | >300% | 111% | 89% | 60% | 189% |
| 25% | 91% | 112% | >300% | >300% | 104% | 91% | 58% | 193% |
| 30% | 110% | 115% | >300% | >300% | 100% | 86% | 58% | 202% |
| 35% | 96% | 105% | >300% | >300% | 97% | 89% | 58% | 209% |
| 40% | 93% | 104% | >300% | >300% | 107% | 90% | 58% | 217% |
| 45% | 100% | 111% | >300% | >300% | 111% | 89% | 59% | 225% |
| 50% | 216% | >300% | >300% | >300% | 112% | 92% | 59% | 239% |
| 55% | 117% | 282% | >300% | >300% | 114% | 91% | 59% | >300% |
| 60% | 106% | 212% | >300% | >300% | 114% | 90% | 59% | >300% |
| 65% | 119% | 237% | >300% | >300% | 114% | 89% | 59% | 279% |
| 70% | 105% | 220% | >300% | >300% | 119% | 89% | 63% | >300% |
| 75% | 182% | 220% | 276% | >300% | 121% | 90% | 63% | >300% |
| 80% | 206% | 212% | 250% | 287% | 130% | 92% | 67% | >300% |
| 85% | 109% | 215% | 223% | 230% | 259% | 94% | 76% | >300% |
| 90% | 202% | 205% | 214% | 223% | 262% | 97% | 89% | >300% |
| 95% | 203% | 201% | 206% | 211% | 262% | 106% | 113% | >300% |

Fig. 12. Continued on next page.

**memkeyval**

| Load | LLC (small) | LLC (med) | LLC (big) | DRAM | HyperThread | CPU power | Network | brain |
|---|---|---|---|---|---|---|---|---|
| 5% | 115% | 209% | >300% | >300% | 26% | 192% | 27% | 197% |
| 10% | 88% | 148% | >300% | >300% | 31% | 277% | 28% | 232% |
| 15% | 88% | 159% | >300% | >300% | 32% | 237% | 28% | >300% |
| 20% | 91% | 107% | >300% | >300% | 32% | 294% | 29% | >300% |
| 25% | 99% | 207% | >300% | >300% | 32% | >300% | 29% | >300% |
| 30% | 101% | 119% | >300% | >300% | 32% | >300% | 27% | >300% |
| 35% | 79% | 96% | >300% | >300% | 33% | 219% | >300% | >300% |
| 40% | 91% | 108% | >300% | >300% | 35% | >300% | >300% | >300% |
| 45% | 97% | 117% | >300% | >300% | 39% | 292% | >300% | >300% |
| 50% | 101% | 138% | >300% | >300% | 43% | 224% | >300% | >300% |
| 55% | 135% | 170% | >300% | >300% | 48% | >300% | >300% | >300% |
| 60% | 138% | 230% | >300% | >300% | 51% | 252% | >300% | >300% |
| 65% | 148% | 182% | >300% | >300% | 56% | 227% | >300% | >300% |
| 70% | 140% | 181% | 280% | >300% | 62% | 193% | >300% | >300% |
| 75% | 134% | 167% | 225% | 252% | 81% | 163% | >300% | >300% |
| 80% | 150% | 162% | 222% | 234% | 119% | 167% | >300% | >300% |
| 85% | 114% | 144% | 170% | 199% | 116% | 122% | >300% | >300% |
| 90% | 78% | 100% | 79% | 103% | 153% | 82% | >300% | >300% |
| 95% | 70% | 104% | 85% | 100% | >300% | 123% | >300% | >300% |

Each entry is color-coded as follows: is $\geq$120%, is between 100% and 120%, and is $\leq$100%.

Fig. 12. Impact of interference on shared resources on websearch, ml_cluster, and memkeyval. Each column is an antagonist and each row is a load point for the workload. The values are latencies, normalized to the SLO latency.

## ACKNOWLEDGMENTS

We sincerely thank Luiz Barroso and Chris Johnson for their help and insight in making our work possible at Google. We also thank Christina Delimitrou, Caroline Lo, and the anonymous reviewers for their feedback on earlier versions of this article.

## REFERENCES

Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM'08)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1402958.1402967

Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM'10)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1851182.1851192

Luiz Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *Computer* 40, 12 (Dec. 2007).

Luiz André Barroso, Jimmy Clidaras, and Urs Holzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (2nd ed.). Morgan & Claypool.

Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO.

Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA.

Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO.

Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press, Cambridge.

Bob Briscoe. 2007. Flow rate fairness: Dismantling a religion. *SIGCOMM Comput. Commun. Rev.* 37, 2 (March 2007). DOI:http://dx.doi.org/10.1145/1232919.1232926

Martin A. Brown. 2006. Traffic Control HOWTO. Retrieved from http://linux-ip.net/articles/Traffic-Control-HOWTO/.

Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. 2014. Long-term SLOs for reclaimed cloud computing resources. In *Proceedings of SOCC*.

Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/2485922.2485949

Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based scheduling: If you're late don't blame us!. In *Proceedings of the 5th Annual Symposium on Cloud Computing*.

Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013).

Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX.

Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT.

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1736020.1736058

H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 2011 38th Annual International Symposium on Computer Architecture*.

Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*.

Donald Gross. 2008. *Fundamentals of Queueing Theory*. John Wiley & Sons, New York NY.

Fei Guo, Hari Kannan, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. 2007a. From chaos to QoS: Case studies in CMP resource management. *SIGARCH Comput. Arch. News* 35, 1 (March 2007). DOI:http://dx.doi.org/10.1145/1241601.1241608

Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. 2007b. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC. DOI:http://dx.doi.org/10.1109/MICRO.2007.6

Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward dark silicon in servers. *IEEE Micro* 31, 4 (2011). DOI:http://dx.doi.org/10.1109/MM.2011.77

Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. 2006. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1152154.1152161

Intel. 2003. Serial ATA II Native Command Queuing Overview. Retrieved from http://download.intel.com/support/chipsets/imsm/sb/sata2_ncq_overview.pdf.

Intel. 2014. Intel® 64 and IA-32 architectures software developer's manual. 3B: System Programming Guide, Part 2 (Sep 2014).

iperf. 2011. Iperf - The TCP/UDP Bandwidth Measurement Tool. Retrieved from https://iperf.fr/.

Teerawat Issariyakul and Ekram Hossain. 2010. *Introduction to Network Simulator NS2* (1st ed.). Springer.

Ravi Iyer. 2004. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1006209.1006246

Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. 2007. QoS policies and architecture for cache/memory in CMP platforms. In *Proceeding of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1254882.1254886

Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. 2010. Web search using mobile cores: Quantifying and mitigating the price of efficiency. *SIGARCH Comput. Arch. News* 38, 3 (June 2010). DOI:http://dx.doi.org/10.1145/ 1816038.1816002

Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceeding of the 49th Annual Design Automation Conference (DAC'12)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/2228360.2228513

Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. 2013. EyeQ: Practical network performance isolation at the edge. In *Proceeding of the 10th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, Berkeley, CA.

Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. 2014. Tradeoffs between power management and tail latency in warehouse-scale applications. In *IISWC*.

Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable low latency for data center applications. In *Proceeding of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. ACM, New York, NY, Article 9. DOI:http://dx.doi.org/10.1145/2391229.2391238

Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. In *Proceeding of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*.

Wonyoung Kim, M. S. Gupta, Gu-Yeon Wei, and D. Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceeding of the IEEE 14th International Symposium on High Performance Computer Architecture, 2008 (HPCA'08)*. DOI:http://dx.doi.org/10.1109/HPCA.2008.4658633

Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. 2012. Building high-level features using large scale unsupervised learning. In *Proceeding of the International Conference in Machine Learning*.

Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceeding of the SIGOPS European Conference on Computer Systems (EuroSys)*.

Bin Li, Li Zhao, Ravi Iyer, Li-Shiuan Peh, Michael Leddige, Michael Espig, Seung Eun Lee, and Donald Newell. 2011. CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs. *J. Parallel Distrib. Comput.* 71, 5 (May 2011). DOI:http://dx.doi.org/10.1016/j.jpdc.2010.10.013

Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceeding of the 40th Annual International Symposium on Computer Architecture*.

Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *Proceeding of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA'12)*. IEEE Computer Society, Washington, DC. DOI:http://dx.doi.org/10.1109/HPCA.2012.6168955

Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceeding of the IEEE 14th International Symposium on High Performance Computer Architecture, 2008 (HPCA'08)*. DOI:http://dx.doi.org/10.1109/HPCA.2008.4658653

Huan Liu. 2011. A measurement study of server utilization in public clouds. In *Proceeding of the 2011 IEEE 9th International Conference on Dependable, Autonomic and Secure Computing (DASC)*.

Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. 2009. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar'09)*. USENIX Association, Berkeley, CA.

Yanpei Liu, Stark C. Draper, and Nam Sung Kim. 2014. SleepScale: Runtime joint speed scaling and sleep states management for power efficient data centers. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, Piscataway, NJ.

David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, Piscataway, NJ.

David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/2749469.2749475

Krishna T. Malladi, Benjamin C. Lee, Frank A. Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. 2012. Towards energy-proportional datacenter memory with mobile DRAM. *SIGARCH Comput. Arch. News* 40, 3 (June 2012). DOI:http://dx.doi.org/10.1145/2366231.2337164

R. Manikantan, Kaushik Rajan, and R. Govindarajan. 2012. Probabilistic shared cache management (PriSM). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Washington, DC.

Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM Intl. Symp. on Microarchitecture (MICRO-44'11)*.

J. Mars, Lingjia Tang, K. Skadron, M. L. Soffa, and R. Hundt. 2012. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro.* 32, 3 (May 2012). DOI:http://dx.doi.org/10.1109/MM.2012.22

Paul Marshall, Kate Keahey, and Tim Freeman. 2011. Improving utilization of infrastructure clouds. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*.

McKinsey & Company. 2008. Revolutionizing data center efficiency. In *Proceedings of the Uptime Institute Symposium*.

David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2009. PowerNap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*.

David Meisner, Christopher M. Sadler, Luiz Andr Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. 2011. Power management of online data-intensive services. In *Proceedings of the 38th ACM Intl. Symp. on Computer Architecture*. ACM, New York, NY.

Paul Menage. 2007. CGROUPS. Retrieved from https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. 2011. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/2155620.2155664

Vijay Nagarajan and Rajiv Gupta. 2009. ECMon: Exposing cache events for monitoring. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1555754.1555798

R. Nathuji, A. Kansal, and A. Ghaffarkhah. 2010. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of EuroSys, France*.

K. J. Nesbit, Nidhi Aggarwal, J. Laudon, and J. E. Smith. 2006. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006 (MICRO-39)*. DOI:http://dx.doi.org/10.1109/MICRO.2006.24

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. 2013. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. of the USENIX Annual Technical Conference (ATC'13)*.

W. Pattara-Aukom, S. Banerjee, and P. Krishnamurthy. 2002. Starvation prevention and quality of service in wireless LANs. In *The 5th International Symposium on Wireless Personal Multimedia Communications, 2002*, Vol. 3. DOI:http://dx.doi.org/10.1109/WPMC.2002.1088344

M. Podlesny and C. Williamson. 2012. Solving the TCP-incast problem with application-level scheduling. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Press, Piscataway, NJ. DOI:http://dx.doi.org/10.1109/MASCOTS.2012.21

Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi, and Xiao Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, Piscataway, NJ.

M. K. Qureshi and Y. N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. DOI:http://dx.doi.org/10.1109/MICRO.2006.49

Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. 2000. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/339647.339685

Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, New York, NY.

Chuck Rosenberg. 2013. Improving Photo Search: A Step Across the Semantic Gap. Retrieved from http://googleresearch.blogspot.com/2013/06/improving-photo-search-step-across.html.

Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. *SIGARCH Comput. Archit. News* 39, 3 (June 2011). DOI:http://dx.doi.org/10.1145/2024723.2000073

Yoon Jae Seong, Eyec Hyun Nam, Jin Hyuk Yoon, Hongseok Kim, Jin yong Choi, Sookwan Lee, Young Hyun Bae, Jaejin Lee, Yookun Cho, and Sang Lyul Min. 2010. Hydra: A block-mapped parallel flash memory solid-state disk architecture. *IEEE Trans. Comput.* 59, 7 (July 2010). DOI:http://dx.doi.org/10.1109/TC.2010.63

Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. 2011. METE: Meeting end-to-end qos in multicores through system-wide resource management. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'11)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1993744.1993747

Shekhar Srikantaiah, Mahmut Kandemir, and Qian Wang. 2009. SHARP control: Controlled shared cache management in chip multiprocessors. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/1669112.1669177

Shingo Tanaka and Christos Kozyrakis. 2014. High performance hardware-accelerated flash key-value store. In *Proceedings of the 2014 Non-volatile Memories Workshop (NVMW)*.

Lingjia Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. 2011. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the 2011 38th Annual International Symposium on Computer Architecture (ISCA)*.

Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: A software-defined storage architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 182–196. DOI:http://dx.doi.org/10.1145/2517349.2522723

Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimpi, T. Sivabalan, and Rajesh Subbiah. 2010. Worth their watts? An empirical study of datacenter servers. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. 2012. DejaVu: Accelerating resource allocation in virtualized environments. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, UK.

Ben Verghese, Anoop Gupta, and Mendel Rosenblum. 1998. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. ACM, New York, NY, 181–192. DOI:http://dx.doi.org/10.1145/291069.291044

Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM'11)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/2018436.2018443

Carole-Jean Wu and Margaret Martonosi. 2008. A comparison of capacity management schemes for shared CMP caches. In *Proceedings of the 7th Workshop on Duplicating, Deconstructing, and Debunking*, Vol. 15. Citeseer.

Yuejian Xie and Gabriel H. Loh. 2009. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/1555754.1555778

Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*.

A. Yasin. 2014. A top-down method for performance analysis and counters architecture. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44.

Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic.

Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

Received October 2015; accepted January 2016