SLO beyond the Hardware Isolation Limits

Abstract

Performance isolation is a keystone for SLO guarantees with shared resources in cloud and datacenter environments. To meet SLO requirements, the state of the art relies on hardware QoS support (e.g., Intel RDT) to allocate shared resources such as last-level caches and memory bandwidth for co-located latency-critical applications. As a result, the number of latency-critical applications that can be deployed on a physical machine is bounded by the hardware allocation capability. Unfortunately, such hardware capability is very limited—even the latest Intel Xeon E5 v3 processors support at most four partitions for last-level caches, i.e., at most four applications can have dedicated resource allocation. This paper discusses the feasibility and unexplored challenges of providing SLO guarantees beyond the limits of hardware capability. We present CoCo to show the feasibility and the benefits. CoCo schedules applications to time-share interference-free partitions as a transparent software layer. Our evaluation shows that CoCo outperforms non-partitioned and the round-robin approaches by up to 9× and 1.2×.

1 Introduction

Service-level objectives (SLOs) guarantee in shared environments such as cloud and datacenters is predominantly done by resource isolation [6, 9, 11, 12, 22–24, 31, 37]. Without resource isolation, the performance of an application can be interfered with and impaired by co-located applications that compete for shared resources such as CPU, last-level caches, and memory bandwidth. In shared cloud and datacenter environments, such interference is inevitable and hard to mitigate, leading to unpredictable performance [3, 6, 13, 21, 23, 24, 38].

To provide performance isolation, hardware QoS support [1, 8] and OS kernel extensions are developed. The key idea is to reserve and allocate dedicated resources for applications with strict SLO requirements, i.e., latency-critical (LC) applications. For example, Intel RDT (Resource Director Technologies) provides cores with fine-grained isolation support for last-level caches [9] and memory bandwidth [14]. Control groups (cgroups) [4, 5] can be used to leverage the hardware support by assigning tasks to cores, and to limit the I/O bandwidth and memory capacity that are available to a group of tasks. The state-of-the-art resource management solutions [6, 11, 12, 22–24, 37] all rely on those supports to meet SLO requirements by reserving dedicated resources for LC applications; meanwhile, the unreserved resources could be shared among best-effort applications (e.g., batch jobs) to improve resource utilization.

Currently, the number of LC applications that can be deployed on a physical machine is bounded by the hardware allocation capability. For example, PARTIES [6], the state-of-the-art resource partitioning technique, cannot support more applications than the number of hardware partitions (defined as CLOSs, Classes of Services, in Intel RDT). Unfortunately, the hardware allocation capability is very limited—even the latest Intel Xeon E5 v3 family processors only support four CLOSs [25]. That means that at most four LC applications can be deployed no matter the number of CPU cores or the memory size.

The fact that resource isolation is bounded by the limited hardware allocation capability hinders improvements of resource utilization. For instance, four LC applications are consolidated on a physical machine with a 4-CLOS processor. Each application is assigned 10% of the last-level cache (in total 40%) to guarantee its SLO. The wasted 60% of the cache capacity could otherwise be utilized to support more LC applications. The problem becomes emergent, considering recent trends of microservices and serverless computing which encourages "micro" applications with stringent SLO requirements [7, 28, 36]. Certainly, a fundamental solution to this problem is to redesign hardware QoS support to expose as many CLOSs as possible. However, deploying new hardware takes time, let alone the fundamental challenges in scaling cache associativity (discussed in §6). We believe that the discrepancy between the hardware QoS support and the scale of CPU cores will continue to exist if not become worse.

This paper investigates the feasibility of providing SLO guarantees beyond the limits of hardware allocation capability, and discusses the related issues and limitations (see §3.4). Our aim is to support SLO guarantees for more applications than the number of CLOSs provided by the hardware and to exploit unused hardware resources. We believe that providing SLO guarantees beyond the hardware isolation limits is significant—instead of passively waiting for hardware innovations, we seek for software-only solutions to efficiently utilize the CLOSs available on the physical machine.

Our key insight is to treat CLOS-based partitions as scarce resources and enable *time-sharing* among co-located applications without breaking application SLOs.

This is in contrast to state-of-the-art approaches (e.g., PARTIES [6]) that statically assign each application to a partition and increase or decrease the amount of resources allocated to that partition so as to guarantee the application SLO. Since each application has different sensitivities with regards to the amount of resources and time slices associated with a partition, the key challenge is to design the time-shareability based on the characteristics of co-located applications, and not to trade off SLOs for improving resource utilization.

We present CoCo, a coordinated container scheduler that transparently allows tens of SLO-oriented LC workloads to share a limited number of partitions of last-level caches and memory bandwidth. CoCo profiles the sensitivity of the target workload regarding resource limits. CoCo configures the CLOS partitions based on the sensitivity characteristics and dynamically schedules workloads to different partitions based on their SLO requirements. The core of CoCo is a novel application-aware time-sharing algorithm that maximizes the overall performance and resource utilization (see §3). We implemented a prototype of CoCo as a user-level runtime system for Linux, which requires no modification to the underlying kernel or co-located applications.[1] Our evaluation shows that CoCo improves the maximum affordable client load without SLO violations by up to 9× and 1.2×, compared with non-partitioned and the round-robin approach [30]. CoCo is the first step in the direction of providing SLOs beyond the hardware limits but there are still unexplored challenges that need to be addressed (see §3.4).

2 Hardware QoS Support

As hardware manufacturers continue to add additional cores onto processors, more workloads can be consolidated together. However, co-located workloads could contend for shared resources, such as last-level caches (LLC) and memory-bandwidth. For LC workloads, the unexpected delay for fetching data from the main memory (due to cache contention and invalidation) can negatively impact performance. Hardware QoS support is recently provided by many hardware manufacturers (e.g, Intel [9, 14], AMD [1], ARM [32], IBM [10]) to support resource isolation. Most existing hardware QoS support is for LLC cache allocation enforcement, LLC cache occupancy monitoring, LLC code-data prioritization, and memory bandwidth enforcement/allocation.

For instance, Intel's CAT [9] and MBA [14] are developed for addressing the shared resource contention for LLC and main memory bandwidth respectively. The goal of CAT/MBA is to enable resource allocation based on partitions or CLOSs. Similar CLOS-based resource partitioning features are also provided by AMD processors [1]. The processor exposes a set of CLOSs into which applications (or individual threads) can be assigned. A given CLOS used for CAT means the same thing as a CLOS used for MBA. The number of CLOSs are typically limited by the hardware. For example, both Intel(R) Xeon(R) E5 v3 family processors and AMD EPYC 7002 series only support at most 4 CLOSs [25].

For CAT, a capacity bit-mask provides a hint to the hardware indicating the cache space an application should be limited to (not necessarily way partitioning). A mask bit set to "1" specifies that a particular CLOS can access the cache subset represented by that bit and vice versa. For MBA, a programmable request rate controller is introduced between the cores and the high-speed interconnect to control memory accesses. It enables indirect control over memory bandwidth for cores over-utilizing bandwidth relative to their CLOS configuration. The memory bandwidth throttling value is of the format of a percentage (with 100% meaning full access) and the limit and granularity of MBA are machine-related.

3 CoCo: SLO-aware Partition Sharing

We realize our idea of sharing CLOS partitions in CoCo, a coordinated container scheduler that transparently enables running tens of SLO-oriented LC workloads beyond the hardware isolation limits. CoCo makes interference-free CLOS partitions shareable in a way that no SLO of co-located workloads would be violated, without overprovisioning. Instead of statically assigning an LC workload to a partition and controlling the amount of resources allocated to the partition [6, 12, 22, 23], CoCo treats each partition as a scarce resource and enables time-sharing the resource among co-located workloads.

CoCo is application-aware. It characterizes the sensitivity of workloads with regard to resource limits through profiling (§3.1). CoCo does this by running the workload along with resource limiters which throttling resource access to simulate performance interference. CoCo will dynamically configure the CLOS resource allocation with Intel CAT/MBA and schedule containers to share the time-slices of each CLOS without violating SLOs in the runtime. This is done based on the profiled sensitivity characteristics (§3.2). Finally, the scheduler in CoCo adjusts to workload changes by adjusting parameters based on runtime monitoring.

3.1 Profiling Sensitivity

The objective of profiling is to understand the sensitivity of the target workload to the scarcity of the same type of resource [6, 12]. For example, with highest contention intensity in terms of last-level cache (LLC), the 99% tail

[1]CoCo is open-sourced and available at: https://anonymous.4open.science/r/a3f84af9-5901-4400-89dd-98b3ef15cc1d/.

latency is 8× of the median latency in Memcached, while the 99% tail latency is 28× of the median latency in Nginx. Intuitively, the scheduler should allocate larger time slices to Nginx workloads to be associated with the CLOS with more LLC ways, compared to Memcached workloads. One workload could have different sensitivities to different resource shortages as well [6, 12]. For Nginx, the performance degrades the same facing both memory bandwidth contention or LLC contention; but LLC shortage is more disastrous compared with memory bandwidth shortage for Memcached and MongoDB workloads. With this sensitivity profiling, the scheduler can optimize for both performance and resource utilization efficiency.

To quantify the subtle, diverse sensitivities discussed above, it requires a metric that can be measured simply and practically. CoCo uses a metric termed *slowdown*. Assume that N workloads are co-located on the same physical server and that the CPU equipped in the server provides the LLC with L ways, the maximum memory bandwidth of B, and the maximum MBA level of 100% (i.e., no throttling). The resource allocation state S_i of workload i ($i \in [0, N-1]$) is defined as (l_i, m_i), where l_i and m_i denote the LLC ways and MBA levels allocated to the workload i. The slowdown of workload i with the resource allocation state S_i is defined in Eq. 1, where SL_{full} and SL_{S_i} denote the load which sustains the SLO when the workload is allocated full resources and the resources specified in state S_i, respectively.

$$slowdown_{S_i} = \frac{SL_{full}}{SL_{S_i}} \tag{1}$$

Given the same system state S_i, the higher the slowdown value is, the more sensitive that the workload i is when facing the LLC or memory bandwidth throttling. Consequently, a workload with a higher slowdown value should be allocated to a partition with larger capacity and longer time window, compared with other co-located workloads.

The workloads are profiled with a resource limiter starting from ($l_i = L$, $m_i = 100\%$) and reducing the allocated LLC and memory bandwidth partition step by step (step size is the minimum adjustment configuration defined by CAT/MBA on each machine). For each step, the slowdown values of the workload are calculated.

3.2 Dynamic Partitioning

Dynamic partitioning configures container-to-core and core-to-CLOS mappings to meet SLOs of all co-located workloads. The scheduling algorithm designed and implemented in CoCo is a multi-level queue-based weighted round-robin (MQ-WRR) where the design requirements are: (i) to give preference to workloads with high slowdown values, (ii) to take the difference of workload sensitivity to shared-resources into account, and (iii) to avoid starvation. The CLOS configurations in Intel CAT/MBA

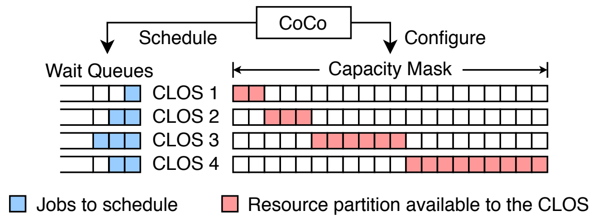

Figure 1: An example of CoCo scheduling.

are modeled as queues. Each CLOS is associated with a subset of caches that do not overlapp with another CLOS. There is one CLOS reserved particularly for the scheduling process of CoCo and other background jobs (e.g., system jobs) running on the physical server that are not latency-critical. All other CLOSs are for scheduling of co-located LC workloads running in containers.

Figure 1 gives an example of CoCo scheduling using Intel CAT. Each CLOS is configured using capacity bitmasks which represent cache capacity and indicate the degree of isolation between classes. Once CAT is configured, the processor allows access to portions of the cache according to the established CLOS. We use `CLOS1` for running background jobs that do not have critical SLOs, as well as CoCo's scheduling process. `CLOS2-4` are configured to have 3, 6, and 9 bits respectively. Each CLOS has a wait queue consisting of containers that are going to be scheduled to the CLOS in the next time slice. Each CLOS also has a working set (which is usually singleton for minimizing interference; but it can also serve two containers that are sensitive to different types of shared-resources) consisting of the container that is running right now. CoCo is scheduling the containers in the weighted round-robin manner onto the multi-level queues. Among N workloads, the scheduling time slice on a CLOS (with state s) associated to workload i is set to be $weight_i$, which is defined as:

$$weight_i = \frac{slowdown_{S_i}}{\sum_{k=0}^{N} slowdown_{S_k}} \tag{2}$$

Memory bandwidth partitioning using MBA is similar to cache partitioning. Instead of setting bit-masks, MBA allows directly indicating the percentages of allowed memory bandwidth (an integer between 0 and 100).

If no schedule can meet all SLOs, admission control will kick in to keep only affordable workloads.

3.3 Handling Conflicts

Although Intel CAT and MBA provide trustworthy partitioning of LLC and memory bandwidth to each CLOS, there may be conflicts when configuring CAT and MBA at the same time. Since MBA uses a programmable rate controller between the cores and the interconnect, LLC and memory controller, bandwidth to LLC may also be reduced when reducing memory bandwidth using MBA.

For instance, the effect of increasing the bit length of the capacity bit-mask for `CLOS1` has a higher chance to be canceled out by setting a lower throttling value for `CLOS1`. Therefore, it is hard to throttle memory-bandwidth-intense workloads which also uses the off-core caches effectively.

To avoid partition conflicts between CAT and MBA, CoCo always increases or decreases memory bandwidth together with LLC capacity. Otherwise, throttling bandwidth-intensive workloads, which also use the off-core caches, will be compromised by increasing memory bandwidth but decreasing LLC capacity. As shown in §4.2, monotonically configuring CAT and MBA yields a 29% improvement of affordable client load.

Once a set of CLOSs are configured by setting the bitmasks, the hardware will get the hint on how to partition the cache space for each CLOS. After a change of CLOS configuration, the changed cache space (ΔC in terms of cache ways) will be added to or removed from the CLOS. However, the cached pages in the cache space ΔC will not be flushed, which limits the new owner of the CLOS (which ΔC belongs to) from using the newly allocated cache space. Similarly, a change of CLOS-process association has the same problem. For instance, `CLOS-A` and `CLOS-B` own cache partition C_a and C_b respectively. After a configuration change, `CLOS-A` owns C_b and `CLOS-B` owns C_a. But cache hits from `CLOS-A` can still lead to access to C_a, which is no longer owned by `CLOS-A` and thus limits the usage of C_a by `CLOS-B`. If the access frequency from `CLOS-A` is high, the least-frequently-used cache replacement policy does not help with CLOS migration.

To resolve cache misses after CLOS migration, CoCo flushes and invalidates the cache in the original cache partition for each CLOS after each configuration change by paying up to 6.1% overhead of cache warmup (§4.2).

3.4 Discussion

CoCo is the first step in the direction towards time-sharing of CLOSs in hardware QoS support without violating SLOs, thus improving resource utilization on physical machines. There are still unexplored challenges that need to be addressed. First, just like statically allocating an LC workload to a CLOS, it cannot serve unlimited client load without SLO violations. We call the point where the client load is no longer affordable the *upper bound*. The application is usually scaled out to more machines when the client load is beyond the upper bound. By time-sharing, the upper bound for each LC workload may be reduced; but we believe the cost can be offset easily through on-demand autoscaling and load-balancing in the cloud or datacenter environments [19, 20, 24, 26, 29].

Second, sensitivity characterization in CoCo is workloads or application inputs dependent, and can not fully observe all system states. For instance, sensitivity can vary with non-modeled system states such as PCIe bandwidth contention or disk I/O contention. Essentially, sensitivity should be a distribution among the factors of workloads and system states, in the presence of hidden resources. To address that, reinforcement learning based approaches [2, 24] under partially observable environments provide possible directions without using painstakingly tuned heuristics.

4 Preliminary Results

Our evaluation is conducted on an application-hosting server with an AMD EPYC 7302P processor with 16 cores and 126 GB of memory, which supports 4 CLOSs with 20-way CAT partitioning and 0-100% range of MBA throttling. We run co-located applications on this server. We use a client machine to create workloads by sending client requests to the application server. The client server has an Intel(R) Core(TM) i7-7700 processor with 8 CPUs and 16 GB memory.

We choose three representative LC applications, Nginx [18], Memcached [15], and MongoDB [17]. To drive the workloads, we use wrk2 [33], memtier [16], and YCSB [35]. We run each application in a separate Docker container on the application server with the workload generation benchmarks set up in the client server. No other background jobs are running on both machines.

Table 1: Impact of CAT/MBA on LC applications in terms of load retainment (normalized to full allocation).

CAT Config.	**Full**	**9-bit Mask**	**6-bit Mask**	**3-bit Mask**
Memcached	100%	88.1 ± 5%	83.8 ± 2%	80 ± 4%
Nginx	100%	75 ± 4%	62 ± 3%	33 ± 3%
MongoDB	100%	58.3 ± 5%	37.3 ± 1%	26 ± 2%

MBA Config.	**Full**	**80%**	**60%**	**40%**	**20%**
Memcached	100%	91.4 ± 6%	87.2 ± 4%	82 ± 5%	78.4 ± 5%
Nginx	100%	93 ± 4%	90.1 ± 3%	87.3 ± 2%	81.1 ± 4%
MongoDB	100%	82.5 ± 5%	74 ± 3%	69.9 ± 4%	64.2 ± 4%

4.1 Sensitivity of Co-located Workloads

As discussed in §3.1, different applications have different sensitivities to resource contention. To cater to different sensitivities of different workloads, we need to understand how Intel CAT and MBA throttling will affect the performance. Based on the characterization, we can then compute the weights used in the MQ-WRR dynamic resource partitioning algorithm (recall §3.2).

Table 1 show the results. The max load is measured without associating any CAT mask (i.e., full-mask) or without adding any MBA throttling rate (i.e., 100% bandwidth) to the workload. We gradually increase the load of the client requests to a point at which the pre-defined

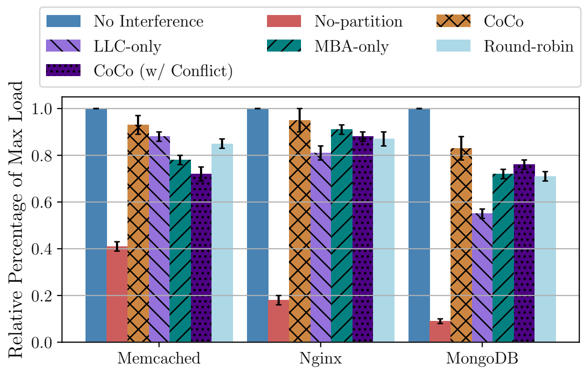

Figure 2: Affordable client load without SLO violations (normalized to no-interference, which is 100%).

SLO is violated within a 95% confidence interval. Then we choose 9-bit, 6-bit, and 3-bit masks as three examples to calculate the slowdown values for LLC, and choose 80%, 60%, 40%, and 20% throttling rates to calculate the slowdown values for MBA.

To do so, we pin each co-located workload to a set of cores (we assign no fewer cores than threads to avoid the CPU being the bottleneck) by changing the `cpuset.cpus` of the `cgroup` configuration for that container, and then associate the corresponding CAT and MBA CLOSs to the cores. After configuration, we gradually increase the load of client requests to a point at which the SLO of the application is violated and mark it as *L*. The ratio of that load *L* to the max load (i.e., with no throttling) is then the load retainment value shown in the table.

4.2 Effectiveness of CoCo

We show the effectiveness of CoCo by comparing it with (i) no partitioning and (ii) standard round-robin scheduling [30]. Since the physical server on which the co-located workloads sit is able to configure a max of 4 CLOSs and in total 20 LLC ways (20-bit mask), we divide the LLC ways into 4 CLOSs, each of which is allocated 2, 3, 6, 9 bits of LLC masks and 10%, 10%, 30%, 50% of memory bandwidth respectively. The co-located workloads are two Memcached instances, two Nginx instances, and two MongoDB instances. The reason why we did not compare with static partitioning is that there is no way for six workloads to statically share four CLOSs. To show the benefit of coordinating CAT and MBA together, we added the evaluation of CoCo with only CAT, only MBA, and conflicting CAT/MBA.

Figure 2 shows the results comparatively. CoCo improves the load retainment compared with no-partitioning by 2.2–9.1×. We summarize several key results: (a) CoCo outperforms CAT-only or MBA-only dynamic resource partitioning by coordinating CAT and MBA together. (b) CoCo schedules the partition in CAT to be positively correlated with the partition in MBA (never reduces the LLC partition when increasing the memory partition) and thus outperforms CoCo with conflicts. (c) CoCo outperforms standard round-robin scheduling by considering the slowdown value of each workload. Round-robin wastes resource partitions on unnecessary workloads and ignores workloads that are more sensitive. While achieving the improvement on load retainment, CoCo brings overhead to the end-to-end latency of 2.4–6.1% due to the partition context switch and cache flush during CLOS migration.

5 Related Work

Using resource partitioning to alleviate contention of shared resources between co-located applications is well studied [6,12,22–24]. For example, Heracles [12] presents a feedback-based resource controller that enables an LC task to be co-located with any number of best-effort (BE) tasks without breaking the SLO of the LC task. However, it can support at most one LC task. PARTIES [6] and CLITE [23] both provide a QoS-aware dynamic resource manager which allows a number of LC applications and BE tasks to safely share one commodity server without violating SLOs. CoPart [22] dynamically analyzes the characteristics of the co-located applications and partitions the LLC and memory bandwidth in a coordinated manner to maximize the overall fairness of the applications. The number of LC applications that the above-mentioned approaches can support is also limited by the hardware features (i.e., the number of CLOSs in Intel CAT/MBA).

6 Concluding Remarks

CoCo is the first step in the direction towards guaranteeing SLOs for more LC applications beyond the hardware allocation capability by treating partitions as scarce resources and designing sharing policies.

The root of the problem is the scarcity of CLOSs supported by existing hardware QoS support. It is perhaps time to rethink hardware QoS support. According to our understanding, the main challenge for way partitioning to support more applications comes from the difficulty of scaling cache associativity to the number of cores, as physical constraints result in increased latency and energy consumption; therefore, simply increasing the number of CLOSs cannot solve the problem because the cache associativity would become the bottleneck. On the other hand, it is hard to partition below the way level [27,34].

In summary, we believe that the discrepancy between the hardware QoS support and the scale of CPU cores and memory will continue to exist if not become worse. Software-based solutions like CoCo that exploit the CLOS shareability will play an important role in providing SLOs and improving hardware resource utilization.

References

[1] AMD64 technology platform QoS extensions. https://developer.amd.com/wp-content/resources/56375.pdf, Accessed 2021/02/01.

[2] Subho Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar Iyer. Inductive-bias-driven reinforcement learning for efficient schedules in heterogeneous clusters. In *Proceedings of the International Conference on Machine Learning (ICML 2020)*, pages 629–641. PMLR, 2020.

[3] Sergey Blagodurov, Alexandra Fedorova, Evgeny Vinnik, Tyler Dwyer, and Fabien Hermenier. Multi-objective job placement in clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015)*, pages 1–12. IEEE, 2015.

[4] Sergey Blagodurov, Daniel Gmach, Martin Arlitt, Yuan Chen, Chris Hyser, and Alexandra Fedorova. Maximizing server utilization while meeting critical SLAs via weight-based collocation management. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 277–285. IEEE, 2013.

[5] cgroups. https://man7.org/linux/man-pages/man7/cgroups.7.html, Accessed 2021/02/01.

[6] Shuang Chen, Christina Delimitrou, and José F. Martínez. PARTIES: QoS-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, page 107–120, 2019.

[7] Yu Gan and Christina Delimitrou. The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, 17(2):155–158, 2018.

[8] Hardware platform QoS technologies. https://wiki.xenproject.org/wiki/Intel_Platform_QoS_Technologies.

[9] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA 2016)*, pages 657–668. IEEE, 2016.

[10] Power systems: Virtual I/O server. ftp://public.dhe.ibm.com/systems/power/docs/hw/p9/p9hb1.pdf, Accessed 2021/02/01.

[11] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 2019)*, pages 272–285, 2019.

[12] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*, page 450–462, 2015.

[13] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 409–425, 2018.

[14] Intel Memory Bandwidth Allocation (MBA) Feature. https://xenbits.xen.org/docs/unstable/features/intel_psr_mba.html, Accessed 2021/02/01.

[15] Memcached. https://memcached.org/, Accessed 2021/02/01.

[16] Memtier Benchmark. https://github.com/RedisLabs/memtier_benchmark, Accessed 2021/02/01.

[17] MongoDB. https://www.nginx.com/, Accessed 2021/02/01.

[18] Nginx. https://www.nginx.com/, Accessed 2021/02/01.

[19] Joe Novak, Sneha Kumar Kasera, and Ryan Stutsman. Cloud functions for fast and robust resource auto-scaling. In *Proceedings of the 11th International Conference on Communication Systems & Networks (COMSNETS 2019)*, pages 133–140. IEEE, 2019.

[20] Joe Novak, Sneha Kumar Kasera, and Ryan Stutsman. Auto-scaling cloud-based memory-intensive applications. In *Proceedings of the 13th International Conference on Cloud Computing (CLOUD 2020)*, pages 229–237. IEEE, 2020.

[21] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the USENIX Annual Technical Conference (ATC 2013)*, pages 219–230, 2013.

[22] Jinsu Park, Seongbeom Park, and Woongki Baek. CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys 2019)*, 2019.

[23] T. Patel and D. Tiwari. CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA 2020)*, pages 193–206, 2020.

[24] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, pages 805–825, 2020.

[25] Resource director technology (RDT) codebase. https://github.com/torvalds/linux/blob/master/arch/x86/kernel/cpu/resctrl/core.c#L196, Accessed 2021/02/01.

[26] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys 2020)*, pages 1–16, 2020.

[27] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling ways and associativity. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 187–198. IEEE, 2010.

[28] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1063–1075. IEEE, 2019.

[29] Parminder Singh, Pooja Gupta, Kiran Jyoti, and Anand Nayyar. Research on auto-scaling of web applications in cloud: survey, trends and future directions. *Scalable Computing: Practice and Experience*, 20(2):399–432, 2019.

[30] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. FLIN: Enabling fairness and enhancing performance in modern NVMe solid state drives. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA 2018)*, pages 397–410. IEEE, 2018.

[31] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling SLOs in network function virtualization. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2018)*, pages 283–297, 2018.

[32] Xiaodong Wang, Shuang Chen, Jeff Setter, and José F Martínez. SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA 2017)*, pages 121–132. IEEE, 2017.

[33] wrk2. https://github.com/giltene/wrk2, Accessed 2021/02/01.

[34] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. DCAPS: dynamic cache allocation with partial sharing. In *Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys 2018)*, 2018.

[35] YCSB. https://github.com/brianfrankcooper/YCSB, Accessed 2021/02/01.

[36] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with ServerlessBench. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 2020)*, pages 30–44, 2020.

[37] Haishan Zhu and Mattan Erez. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016)*, pages 33–47, 2016.

[38] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 45(1):1–28, 2012.