IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 6 November 2023, accepted 17 November 2023, date of publication 24 November 2023,
date of current version 30 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3336685

RESEARCH ARTICLE

COSMOS: Coordinated Management of Cores, Memory, and Compressed Memory Swap for QoS-Aware and Efficient Workload Consolidation for Memory-Intensive Applications

MYEONGGYUN HAN[1], EUNSEONG PARK[1], YOUNGSAM SHIN[2], DEOK-JAE OH[2], YEONGON CHO[2], AND WOONGKI BAEK[3], (Member, IEEE)

[1]Department of Computer Science and Engineering, UNIST, Ulsan 44919, Republic of Korea
[2]Samsung Advanced Institute of Technology, Suwon 16678, Republic of Korea
[3]Department of Computer Science and Engineering and Graduate School of Artificial Intelligence, UNIST, Ulsan 44919, Republic of Korea

Corresponding author: Woongki Baek (wbaek@unist.ac.kr)

This work was supported in part by the Samsung Advanced Institute of Technology, Samsung Electronics Company Ltd.; in part by the National Research Foundation of Korea under Grant NRF-2021R1A2C1011482; and in part by the Institute of Information & Communications Technology Planning & Evaluation under Grant 2021-0-01817.

ABSTRACT With the rapid growth in memory demands, the slowdown of DRAM scaling, and the DRAM price fluctuations, DRAM has become one of the critical resources in cloud computing systems and datacenters. The compressed memory swap (CMS) is a promising technique that improves the effective memory capacity of the underlying computer system by compressing and storing a subset of pages in memory instead of the disk swap. While prior works have extensively investigated resource management techniques for workload consolidation, they lack the capability of dynamically allocating cores, memory, and CMS to the consolidated applications in a controlled and efficient manner. To bridge this gap, this work presents the in-depth characterization of the impact of cores, memory, and CMS on the QoS and throughput of the consolidated latency-critical (LC) and batch applications. Guided by the characterization results, we propose COSMOS, a software-based runtime system for coordinated management of cores, memory, and CMS for QoS-aware and efficient workload consolidation for memory-intensive applications. COSMOS dynamically collects the runtime data from the consolidated applications and the underlying system and allocates the resources to the consolidated applications in a way that achieves high throughput with strong QoS guarantees. Our quantitative evaluation based on a real system and widely-used memory-intensive benchmarks demonstrates the effectiveness of COSMOS in that it robustly satisfies the QoS and achieves high throughput across all the evaluated workload mixes and scenarios and significantly reduces the number of explored system states.

INDEX TERMS Cloud and datacenter computing, compressed memory swap, efficiency, quality-of-service, resource management, workload consolidation.

I. INTRODUCTION

The memory demands in cloud computing systems and datacenters are explosively growing because of the rise of emerging memory-intensive applications such as machine learning and big data applications [37], [48]. In addition, DRAM scaling has slowed down [24], [28] and there have been large fluctuations in DRAM prices [45]. As a result, DRAM has become one of the most critical and expensive components in cloud computing systems and datacenters [45].

The compressed memory swap (CMS) [39] is a promising technique to host memory-intensive applications without increasing the memory capacity of the underlying server

The associate editor coordinating the review of this manuscript and approving it for publication was Alessandro Floris.

system [23], [45]. With the CMS, pages selected as victim pages by the memory reclaim algorithm in the OS are compressed and evicted to the CMS instead of the disk swap. The CMS incurs overheads for compressing and decompressing pages when pages are transferred between the memory and CMS. However, since the page compression and decompression operations are performed using CPU cores and memory, the CMS is significantly faster than the disk swap, which incurs expensive disk I/O operations [45]. The CMS is supported by widely-used OSes such as Windows [3], Linux [6], and macOS [5] and employed in commercial cloud computing systems and datacenters [23], [45].

Workload consolidation is an effective technique to improve the resource efficiency of cloud computing systems and datacenters [11], [27]. Without workload consolidation, dedicated servers are allocated to latency-critical (LC) applications that have soft or hard deadlines in order to satisfy their quality-of-service (QoS) requirements, drastically degrading the resource efficiency of cloud computing systems and datacenters. Workload consolidation significantly improves the resource efficiency by colocating the LC and batch applications on the same physical server. The key challenge for the resource manager for workload consolidation is to find the right amounts of resources allocated to each of the LC and batch applications in order to maximize the resource efficiency while providing QoS guarantees.

Prior works have extensively investigated system software support for workload consolidation [11], [15], [16], [27], [31], [32], [33], [34], [46], [49]. However, most of the prior works present system software techniques that mitigate the performance interference caused by the contention on cores, caches, and memory bandwidth [15], [16], [27], [31], [32], [33], [34], [46], [49] but lack the capability of controlling the contention on memory capacity. While the resource manager proposed in [11] provides memory capacity partitioning, it lacks the capability of dynamic management of the CMS, which is crucial for satisfying the LC application's QoS and improving the throughput of the consolidated applications with the limited memory capacity.

To bridge this gap, this work characterizes the impact of cores, memory, and CMS on the QoS and the throughput of the consolidated applications. Based on the characterization results, we propose a system called COSMOS for coordinated management of cores, memory, and CMS for QoS-aware and efficient workload consolidation for memory-intensive applications. We quantify the effectiveness of COSMOS with various LC and batch applications in various scenarios.

Specifically, this paper makes the following contributions:

- We present the in-depth characterization of the impact of cores, memory, and CMS on the LC application's QoS and throughput of the consolidated memory-intensive LC and batch applications. Through our characterization study, we derive guidelines to effectively find an efficient system state that can significantly improve the overall throughput of the consolidated applications while satisfying the QoS.
- Guided by the characterization results, we propose COSMOS, a software-based system for coordinated management of cores, memory, and CMS for QoS-aware and efficient workload consolidation for memory-intensive applications. COSMOS dynamically collects the runtime data from the consolidated applications and the underlying system and allocates cores, memory, and CMS in a way that significantly improves the throughput of the consolidated applications while satisfying the LC application's QoS.
- We design and implement a prototype of COSMOS as a user-level runtime system on Linux. COSMOS is lightweight and readily applicable to various commodity systems without requiring the specialized hardware support or the modifications of the underlying operating systems.
- We quantify the effectiveness of COSMOS using widely-used memory-intensive LC and batch benchmarks on a real server system. Our experimental results demonstrate that COSMOS provides strong QoS guarantees and achieves high throughput across all the workload mixes with various loads for the LC application and memory overcommit ratios. In addition, COSMOS significantly reduces the number of explored system states by skipping the inefficient system states. To the best of our knowledge, our work is the first to present the in-depth characterization of the impact of cores, memory, and CMS on the QoS and throughput of the consolidated applications and design, implement, and evaluate a coordinated resource manager for cores, memory, and CMS for QoS-aware and efficient workload consolidation for memory-intensive applications on a full commodity server system.

The rest of this paper is organized as follows. Section II provides background information. Section III describes the experimental methodology. Section IV characterizes the impact of cores, memory, and CMS on the QoS and throughput of consolidated applications. Section V presents the design and implementation of COSMOS. Section VI quantifies the effectiveness of COSMOS. Section VII discusses related work. Section VIII concludes the paper.

II. BACKGROUND

A. MEMORY RECLAIM AND CMS

When the memory usage exceeds thresholds, the operating system (OS) performs memory reclaim to secure free space in memory. During memory reclaim, the OS determines a set of victim pages that need to be evicted from memory based on various metrics (e.g., recency, hotness) and moves the victim pages from memory to the swap area to reserve free space.

Memory reclaim can be conducted in a foreground or background manner. When a user-level process requests the OS to allocate a free page but the current memory usage exceeds a threshold, the OS conducts foreground memory reclaim. Since the requesting process is blocked during

the foreground memory reclaim, it can drastically degrade the latency-critical (LC) application's QoS because of the significantly increased tail latency.

When the memory usage exceeds another threshold (typically set to a smaller value than that for the foreground memory reclaim), background memory reclaim can be triggered by the OS or user-level processes even without any pending page allocation requests. One of the major advantages of background memory reclaim is that user-level processes can continue their execution during background memory reclaim.

With the conventional disk swap, victim pages that are evicted from memory are stored in the swap area in the disk. The disk swap incurs significant performance overheads because of the I/O operations that are executed to transfer pages between the memory and disk.

The compressed memory swap (CMS) mitigates the page swapping overhead by storing the victim pages in the in-memory swap area instead of the disk. While the CMS incurs overheads for compressing and decompressing pages, it is still significantly faster than the disk swap by eliminating the I/O operations. Widely-used OSes (e.g., zswap in Linux [6], memory compression in Windows [3], and compressed memory in macOS [5]) support CMS. While this work focuses on Linux and zswap, we believe that the findings from this work can be applied to other OSes and implementations of CMS.

zswap provides various compression and decompression algorithms and memory pools for victim pages [6]. Among the compression and decompression algorithms and memory pools, we use the Lempel–Ziv–Oberhumer (LZO) algorithm [4] and the zbud memory pool [18], which are the default algorithm and memory pool for zswap.

B. WORKLOAD CONSOLIDATION

Workload consolidation enables the colocation of latency-critical (LC) and batch applications on a single physical server. LC applications are user-facing and interactive applications with latency constraints. The target tail latency of an LC application determines its latency constraint (e.g., the 99th percentile latency must be lower than one millisecond) that must be satisfied to enable user-facing and interactive services and is set based on the service-level agreement between the provider and the user. The load for an LC application is defined as the number of incoming requests per second.

In contrast, batch applications are background applications without any latency constraints. The common metric to evaluate the performance of batch applications is throughput such as the number of executed iterations of the main loop per second and the amount of the input data processed per second.

The main objective of workload consolidation is to maximize the throughput of the consolidated applications while providing strong QoS guarantees for the LC application. When resources are allocated in an unmanaged manner, the LC application is likely to violate its QoS because of the performance interference caused by the contention on the resources (e.g., cores, memory capacity) shared by the consolidated LC and batch applications.

To eliminate or mitigate the performance interference, the resource manager for workload consolidation partitions resources between the consolidated LC and batch applications. Production-quality operating systems provide support for core and memory capacity partitioning. For instance, Linux provides core and memory capacity partitioning through control groups (cgroups) [1]. A *cgroup* is a set of processes that can be allocated their own resources. Cores and memory can be partitioned between the consolidated LC and batch applications by associating each application with its own cgroup and allocating disjoint sets of cores and memory to each cgroup. Linux also allows to dynamically change the amounts of the resources allocated to each cgroup.

III. EXPERIMENTAL METHODOLOGY

A. SYSTEM CONFIGURATION

In this work, we use two systems, each of which is used as the server or client system. The server system runs the consolidated latency-critical (LC) and batch applications. The server system is equipped with the 16-core Intel Xeon Gold 6226R CPU, 64 GB memory (4 × 16 GB DIMMs), a 1 TB Samsung 970 EVO Plus NVM-e SSD, and a 100 Gb NIC. 4 GB out of the 64 GB is reserved for the OS. The server system is installed with Ubuntu 22.04 and Linux kernel 6.1.11.

Cgroups is used to partition cores and memory between the LC and batch applications. In addition, `sysfs` is used to dynamically (1) control the amount of the compressed memory swap (CMS) allocated to the LC container (i.e., `/sys/module/zswap/parameters/max_pool_percent`) and (2) track the compression ratio, which is computed by collecting the number of pages stored in the CMS (i.e., `/sys/kernel/debug/zswap/stored_pages`) and the actual amount of memory used by the CMS (i.e., `/sys/kernel/debug/zswap/pool_total_size`).

The client system runs the load generator for each of the evaluated LC applications. The client system is equipped with two 32-core Intel Xeon Gold 6338 CPUs, 64 GB memory (4 × 16 GB DIMMs), a 1 TB Samsung 870 EVO SATA SSD, and a 100 Gb NIC. The client system is installed with Ubuntu 22.04 and Linux kernel 5.15.0. The client and server systems are directly connected through the 100 Gb Ethernet.

B. BENCHMARKS

We use two latency-critical (LC) benchmarks – `memcached` [2], [14] and `silo` [19], [44], which are in-memory key-value store and in-memory database, respectively. The QoS target of `memcached` is that the 99th percentile latency must be lower than 200 microseconds, which is in line with the prior works [22], [38]. We use Lancet, which is an open

TABLE 1. Loads for the LC benchmarks.

Benchmark	Low, medium, and high loads (QPS)
memcached	37,500, 75,000, and 150,000
silo	1,000, 2,000, and 4,000

loop-based load generator for memcached [21]. In line with the prior works [11], [22], we configure the key and value sizes to 30 and 200 bytes and the query type to read-only. In addition, we configure the key popularity to follow a Zipfian distribution [40] with a skewness parameter of 0.99 and the query inter-arrival time to follow an exponential distribution in order to emulate the access [8], [47] and traffic (e.g., micro-bursts) [19], [30] patterns commonly observed in datacenters.

The QoS target of silo is that the 99th percentile latency must be lower than one millisecond, which is in line with the prior works [12], [35]. We use the load generator included in TailBench for silo [19]. The load generator for silo also generates queries with an exponential inter-arrival time distribution based on the observations (e.g., micro-bursts) from the prior works [19], [30].

We also use five batch benchmarks – betweenness centrality (BC) [9], breath-first search (BFS) [9], canneal [10], connected components (CC) [9], and stream [29]. The metric used to quantify the throughput of the batch benchmarks is the number of executed iterations of the main loop per second.

We use the aforementioned LC and batch benchmarks because they are memory-intensive and are widely used for cloud computing systems and datacenter research [11], [12], [15], [16], [35], [45], [47]. The thread count of each benchmark is set to 16, which is same as the number of cores in the CPU on the evaluated server system.[1]

In this work, we refer to a container that contains the LC application as the LC container. In addition, we refer to a container that consists of one or more batch applications as the batch container.

We investigate the impact of cores, memory, and compressed memory swap on the QoS of throughput of the consolidated containers and evaluate the effectiveness of COSMOS with various loads for the LC container. Table 1 summarizes low, medium, and high loads (in queries per second (QPS)) for the LC benchmarks.

We also conduct experiments with various memory overcommit ratios (MORs). The MOR is defined in Equation 1,

[1]There are mainly two widely-used mechanisms to control the concurrency of applications – (1) dynamic threading [36], [43] and (2) thread packing [13], [41]. With dynamic threading, the concurrency of an application is controlled by dynamically adjusting the number of threads of the application. With thread packing, the concurrency of an application is controlled by dynamically adjusting the number of cores allocated to the application. A major disadvantage of dynamic threading is the limited applicability because numerous applications lack the support for dynamic threading. In contrast, thread packing can be applied to all applications regardless of whether they support dynamic threading or not. Because of the advantage of thread packing, we consider it as a mechanism for dynamic concurrency control. To ensure the cores in the evaluated CPU can fully be utilized, we configure the thread count of each benchmark to 16.

TABLE 2. Working-set sizes.

Benchmark	Working-set sizes with low, medium, and high memory overcommit ratios (GB)
memcached	42.1, 45.0, and 51.1
silo	42.1, 45.2, and 51.3
BC	22.5, 24.0, and 27.0
BFS	22.6, 24.1, and 27.3
canneal	22.5, 24.2, and 27.1
CC	22.6, 24.1, and 27.3
stream	22.5, 24.0, and 27.0

where M is the total memory capacity (excluding the amount of the memory reserved for the OS) of the underlying server system and w_{LC} and w_{Batch} denote the working-set size of the LC and batch containers, respectively. Table 2 summarizes the working-set sizes of the evaluated benchmarks with low (i.e., 1.075), medium (i.e., 1.15), and high (i.e., 1.3) MORs in GB. For example, if we consider the workload mix of memcached and BC with the medium MOR, the working-set sizes of memcached and BC are 45.0 GB and 24.0 GB, respectively. The MOR is then computed to be 1.15 (i.e., $\frac{45.0+24.0}{M} = 1.15$, where M is 60 GB) on the evaluated server system using Equation 1.

$$\text{Memory overcommit ratio} = \frac{w_{\text{LC}} + w_{\text{Batch}}}{M} \tag{1}$$

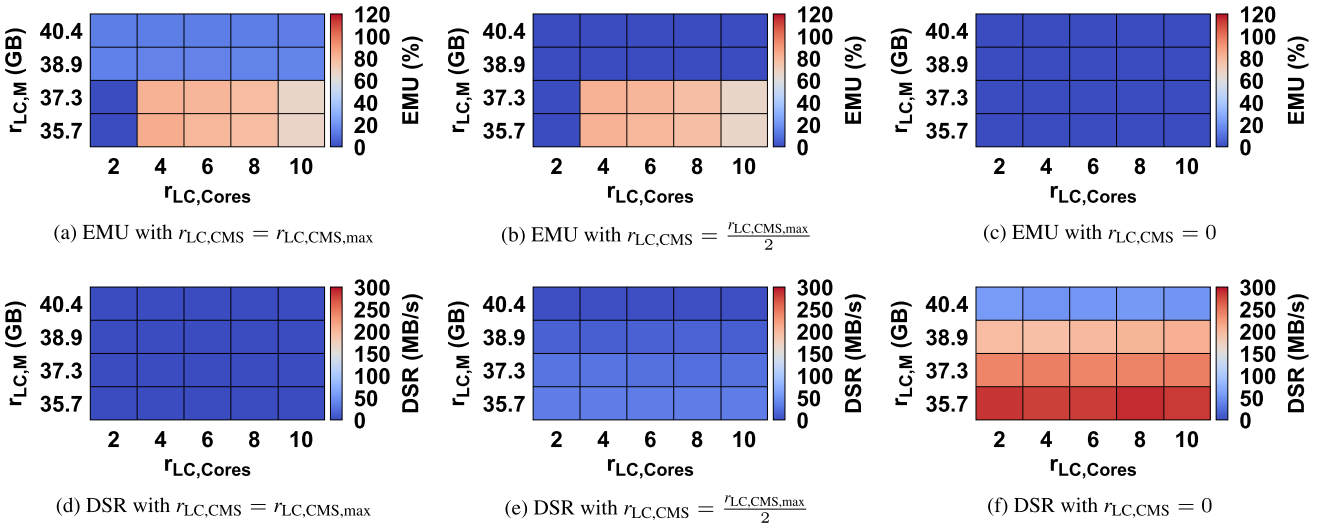
IV. CHARACTERIZATION

In this section, we characterize the impact of cores, memory, and compressed memory swap (CMS) on the QoS and the throughput of the consolidated containers. While we only present the experimental results with memcached (i.e., the latency-critical (LC) container) and stream (i.e., the batch container) for conciseness, other benchmarks exhibit similar data trends. We execute the consolidated containers using the following configurations – (1) low load and low memory overcommit ratio (MOR), (2) low load and high MOR, (3) high load and low MOR, and (4) high load and high MOR.

In each of the configurations, we vary the number of cores and the amounts of the memory and CMS allocated to the LC container and analyze their impact on the QoS and throughput of the consolidated containers. We denote the core count, the amount of memory, and the amount of the CMS allocated to the LC container as $r_{\text{LC,Cores}}$, $r_{\text{LC,M}}$, and $r_{\text{LC,CMS}}$, respectively. In addition, we denote the working-set size of the LC container and the average compression ratio of the pages stored in the CMS as w_{LC} and γ_{LC}.

The upper bound of $r_{\text{LC,CMS}}$ is then computed using Equation 2. Intuitively, Equation 2 indicates that there is no need to further increase $r_{\text{LC,CMS}}$ once it becomes large enough to fit w_{LC} in memory and CMS or $r_{\text{LC,CMS}}$ cannot exceed $r_{\text{LC,M}}$ because the CMS consumes the memory allocated to the LC container.

$$r_{\text{LC,CMS,max}} = \min\left(\frac{w_{\text{LC}} - r_{\text{LC,M}}}{\gamma_{\text{LC}} - 1}, r_{\text{LC,M}}\right) \tag{2}$$



(a) EMU with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (b) EMU with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (c) EMU with $r_{\text{LC,CMS}} = 0$

(d) DSR with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (e) DSR with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (f) DSR with $r_{\text{LC,CMS}} = 0$

FIGURE 1. Impact of cores, memory, and CMS allocated to the LC container with low load and low MOR.

In line with the prior works on workload consolidation [11], [16], [27], we use *effective machine utilization* (EMU), which is a system-wide metric that quantifies the throughput of the consolidated containers. If the LC container violates its QoS, EMU becomes zero.

In contrast, if the LC container satisfies its QoS, EMU is computed to be a positive value. A larger EMU value indicates that the consolidated containers achieve higher throughput. To compute EMU, we first compute the normalized throughput of each of the consolidated applications by dividing the throughput of the application with workload consolidation (i.e., the hardware resources are shared by the consolidated applications) by the solo-run throughput of the application when it is allocated all of the hardware resources. We then compute EMU by summing up the normalized throughput of each of the consolidated applications.[2] EMU can be higher than 100% if the consolidated applications can maintain high throughput (in a non-linear manner) even when relatively small amounts of resources are allocated to them.

Figure 1 shows the EMU and disk swap rates (DSRs) of `memcached` and `stream` with low load and low MOR. Each cell in the heat maps represents a single data point. Each heat map reports 20 data points collected from 20 configurations.

First, when a sufficient amount of the CMS (e.g., $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$) is allocated to the LC container, its QoS is satisfied (i.e., EMU $\neq$ 0) across wide ranges of $r_{\text{LC,Cores}}$ and $r_{\text{LC,M}}$. This is mainly because the LC container requires a relatively small amount of memory with a low MOR and a relatively small number of cores with a low load.

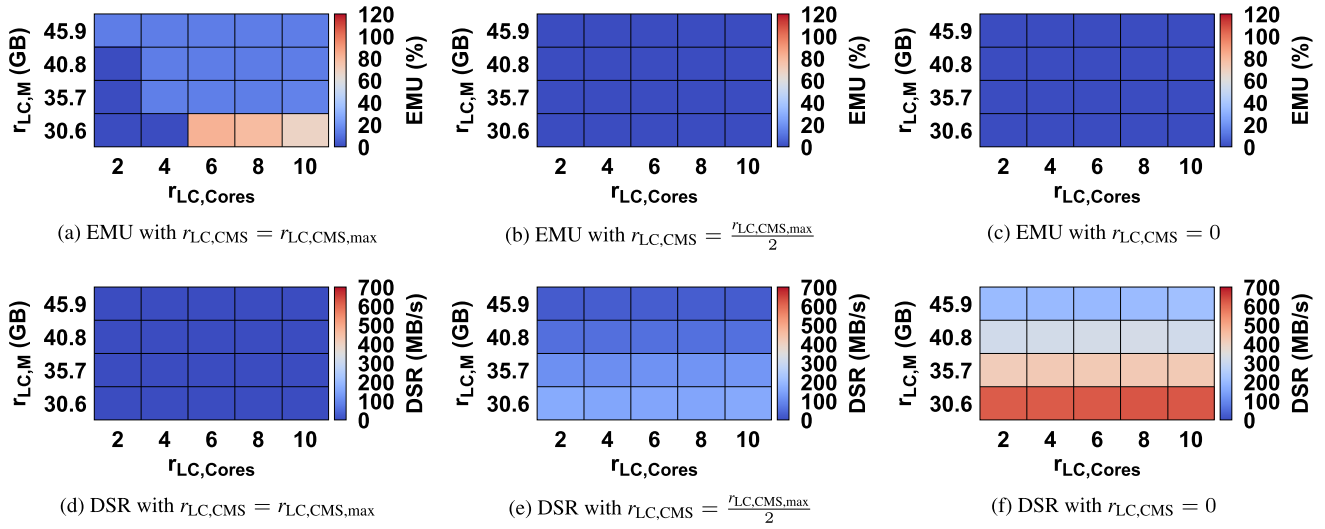
Even when a sufficient amount of the CMS is allocated to the LC container, its QoS is violated (i.e., EMU = 0) with insufficient cores and memory (e.g., $r_{\text{LC,Cores}} = 2$, $r_{\text{LC,M}} = 35.7$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 1a). This is mainly because the contention on the cores shared by the threads of the LC container and the memory reclaim threads. When a smaller amount of memory is allocated to the LC container, more of its data is transferred between the memory and CMS. This increases the CPU utilization of the memory reclaim threads, causing the contention on cores with the threads of the LC container.

Second, when a sufficient amount of the CMS (e.g., $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$) is allocated to the LC container, the EMU tends to increase as the number of cores and the amount of memory allocated to the LC container decreases. The EMU is maximized when the number of cores and the amount of memory allocated to the LC container are small yet just enough to satisfy the LC container's QoS (e.g., $r_{\text{LC,Cores}} = 4$, $r_{\text{LC,M}} = 37.3$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 1a).

We also observe that the EMU gradually increases as the LC container's core count decreases. For example, Figure 1b shows that the EMU increases from 66.3% to 82.5% as the LC container's core count decreases from 10 to 4 when $r_{\text{LC,M}} = 37.3$ GB. This is mainly because the batch container gradually achieves higher throughput as it is allocated more cores.

In contrast, the EMU abruptly increases only when the amount of the memory allocated to the LC container is small enough to make the working-set of the batch container fit in memory. Except for this abrupt change, the EMU minimally changes as the amount of the memory allocated to the LC container decreases. This is mainly because the hotness among the data accessed by the batch container (i.e., `stream`) is uniform. With the uniform memory access pattern, the batch container exhibits high performance only when its entire working set fits in memory. The EMU changes more gradually when the batch container includes applications (e.g., `BC`) that exhibit non-uniform memory access patterns.

[2]Note that the throughput of the LC application (i.e., the load for the LC application) is also used to compute EMU.



(a) EMU with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (b) EMU with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (c) EMU with $r_{\text{LC,CMS}} = 0$

(d) DSR with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (e) DSR with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (f) DSR with $r_{\text{LC,CMS}} = 0$

FIGURE 2. Impact of cores, memory, and CMS allocated to the LC container with low load and high MOR.

Third, when a relatively small amount of the CMS is allocated to the LC container, its QoS is satisfied with fewer configurations. For example, when $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$, the QoS is satisfied with only eight configurations (c.f., 18 configurations when $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$) out of 20 configurations. This is mainly because more memory is needed to make the working set of the LC container fit in memory when a smaller amount of the CMS is allocated to the LC container. When the working set of the LC container does not fit in memory by being allocated with insufficient amounts of the memory and CMS, victim pages are evicted to the disk swap. This incurs the QoS violation of the LC container because of frequent I/O operations.

In an extreme case where $r_{\text{LC,CMS}} = 0$, the LC container's QoS is violated across all the configurations. This indicates that the use of the CMS is required to satisfy the QoS when its working-set size exceeds its allocated memory size.

Figure 2 shows the EMU and disk swap rates (DSRs) of `memcached` and `stream` with the low load and high MOR. First, the overall EMU data trends are similar to the case with the low load and low MOR in that the EMU tends to increase when the number of cores and the amount of memory allocated to the LC container are smaller.

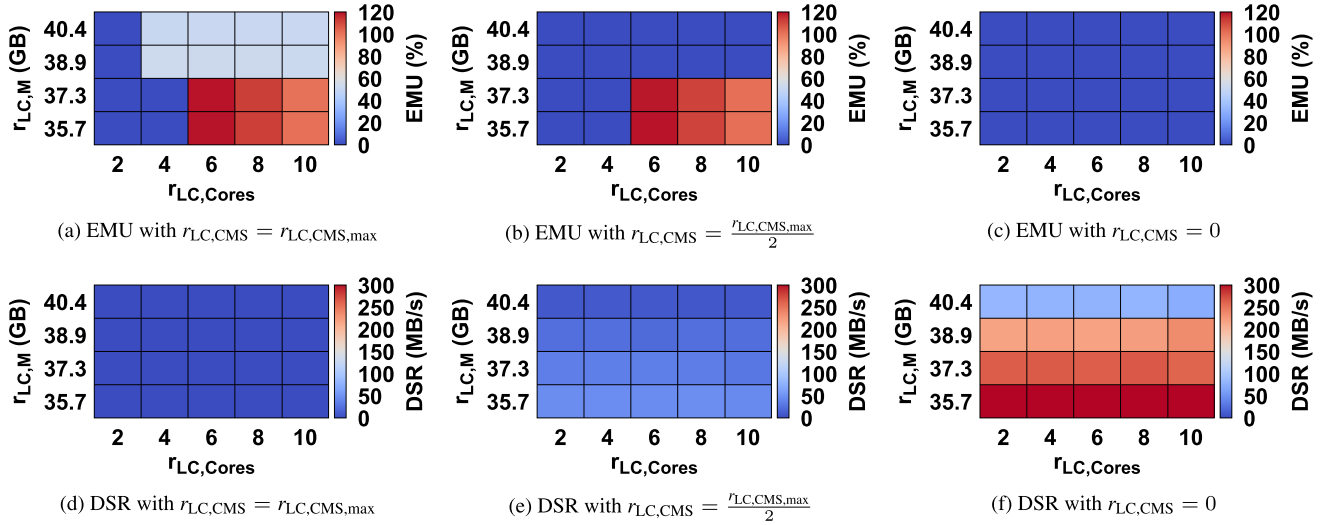
Second, the highest EMU achieved with the low load and high MOR (e.g., $r_{\text{LC,Cores}} = 6$, $r_{\text{LC,M}} = 30.6$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 2a) is similar to the highest EMU achieved with the low load and low MOR (e.g., $r_{\text{LC,Cores}} = 4$, $r_{\text{LC,M}} = 37.3$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 1a). This is mainly because the load for the LC container is same and the batch container achieves similar performance when allocated a sufficient amount of memory.

Third, in comparison with the case with the low load and low MOR, the LC container needs to be allocated a larger number of cores and a smaller amount of memory to achieve the highest EMU (e.g., $r_{\text{LC,Cores}} = 4$, $r_{\text{LC,M}} = 37.3$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 1a vs. $r_{\text{LC,Cores}} = 6$, $r_{\text{LC,M}} = 30.6$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 2a). Since the working-set size of the batch container increases with the high MOR, the LC container needs to be allocated a smaller amount of memory in order to make the working set of the batch container fit in memory. As a result, a larger portion of the data of the LC container is transferred between the memory and CMS, increasing the CPU utilization of the memory reclaim threads. To mitigate the contention on the cores shared by the threads of the LC container and the memory reclaim threads, more cores need to be allocated to the LC container.

Fourth, in comparison with the case with the low load and low MOR, the LC container requires a larger amount of the CMS to satisfy its QoS. For instance, the LC container's QoS is violated across all the configurations when $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ with the low load and high MOR (Figure 2b), whereas it is satisfied with eight configurations when $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ with low load and low MOR (Figure 1b). Since the working-set size of the LC container increases with the high MOR, the LC container requires a larger amount of the CMS to make its working set fit in memory.

Figure 3 shows the EMU and disk swap rates (DSRs) of `memcached` and `stream` with the high load and low MOR. First, the overall EMU data trends are similar to the aforementioned cases in that the EMU tends to increase as the core count and the amount of the memory allocated to the LC container decrease.

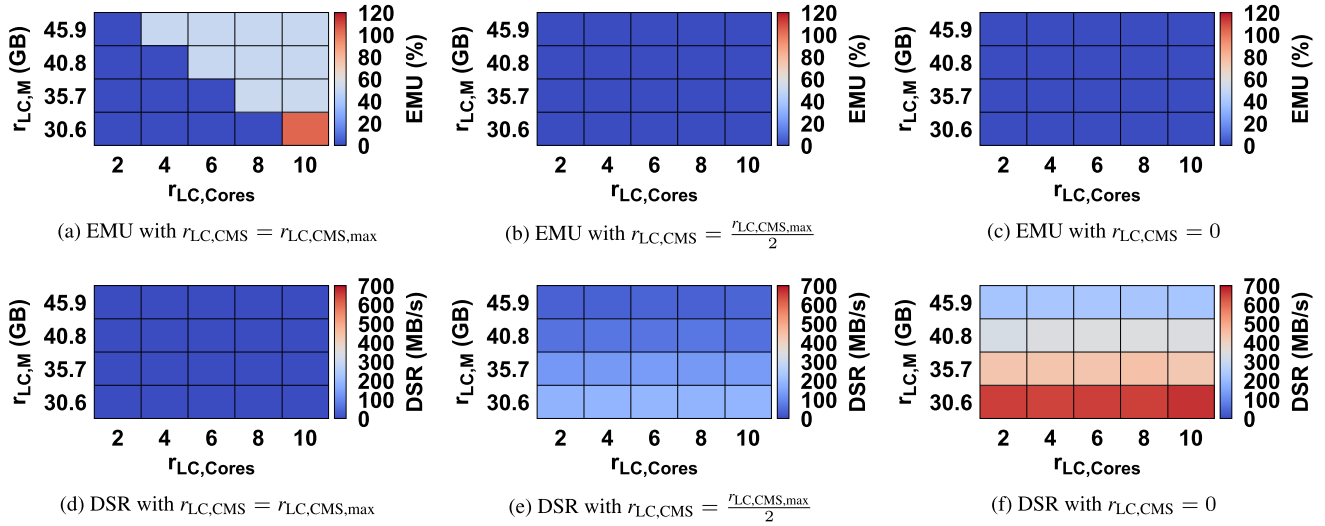
Second, the highest EMU achieved with the high load and low MOR is higher than that with the cases with the low load. Since the LC container is applied with the high load, the portion of the EMU contributed by the LC container increases in comparison with the cases with the low load.



(a) EMU with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (b) EMU with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (c) EMU with $r_{\text{LC,CMS}} = 0$

(d) DSR with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (e) DSR with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (f) DSR with $r_{\text{LC,CMS}} = 0$

FIGURE 3. Impact of cores, memory, and CMS allocated to the LC container with high load and low MOR.



(a) EMU with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (b) EMU with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (c) EMU with $r_{\text{LC,CMS}} = 0$

(d) DSR with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (e) DSR with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (f) DSR with $r_{\text{LC,CMS}} = 0$

FIGURE 4. Impact of cores, memory, and CMS allocated to the LC container with high load and high MOR.

Figure 4 shows the EMU and disk swap rates (DSRs) of `memcached` and `stream` with the high load and high MOR. First, the overall EMU trends are similar to the aforementioned cases. Second, the LC container's QoS is satisfied in fewer configurations than the other cases. This is because the LC container requires larger amounts of resources to satisfy its QoS with the high load and high MOR.

Third, the highest EMU achieved with the high load and high MOR is higher than those achieved with the cases with the low load. Since the throughput of the LC container increases with the high load, the overall EMU also increases.

In contrast, the highest EMU achieved with the high load and high MOR is lower than that achieved with the high load and low MOR. Since the working-set size of the batch container increases with the high MOR, the LC container needs to be allocated with a smaller amount of memory. To make its working set fit in memory, the LC container requires a larger amount of the CMS. Since the CPU utilization of the memory reclaim threads increases with more frequent data transfers between the memory and CMS, the LC container requires a larger number of cores, decreasing the highest EMU achieved with the high load and high MOR.

Our characterization results clearly motivate the need for coordinated management of cores, memory, and CMS to significantly improve the EMU with strong QoS guarantees. The findings learned from the characterization study are summarized as follows.

- **Impact of resources** (**C1**): Cores, memory, and CMS have significant impact on the LC container's QoS and the throughput of the consolidated containers in that EMU widely varies depending on how the resources are allocated to the consolidated containers.

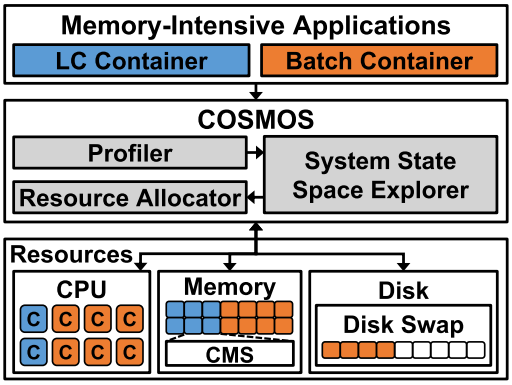


FIGURE 5. Overall architecture of COSMOS.

- **Impact of loads and MORs** (**C2**): There is no single configuration of cores, memory, and CMS that delivers high EMU across various loads and MORs in that the configuration of cores, memory, and CMS that results in high EMU widely varies across various loads and MORs.
- **Allocation of memory and CMS** (**C3**): The EMU of the consolidated container is significantly improved (i.e., satisfying the QoS and achieving high throughput) when the amount of the memory allocated to the LC container is small enough to make the working-set of the batch container fit in memory but the amounts of the memory and CMS allocated to the LC container are large enough to make its working-set fit in memory.
- **Allocation of cores** (**C4**): With a smaller amount of the memory and a larger amount of the CMS allocated to the LC container, the LC container requires a larger number of cores that are sufficient to execute not only its threads but also the memory reclaim threads for satisfying its QoS.

V. DESIGN AND IMPLEMENTATION

Our characterization study shows that resources, loads, and memory overcommit ratios have significant impact on the latency-critical (LC) container's QoS and the throughput of the consolidated containers (i.e., **C1** and **C2** in Section IV). Based on the characterization results, we propose COSMOS, a software-based runtime system that dynamically allocates cores, memory, and compressed memory swap (CMS) to the consolidated LC and batch containers to achieve high throughput while satisfying the QoS for given load and memory overcommit ratio. COSMOS consists of three components – the (1) profiler, (2) system state space explorer, and (3) resource allocator. Figure 5 illustrates the overall architecture of COSMOS.

A. PROFILER

The profiler of COSMOS dynamically collects the runtime data that the system state space explorer uses to make resource allocation decisions. Specifically, it collects the load and tail latency data from the latency-critical (LC) container and the throughput data from the batch container.

In addition, the profiler dynamically collects the working-set sizes of the LC and batch containers, which are available through procfs on Linux. Further, it collects the average compression ratio of the pages stored in the CMS through sysfs at runtime.

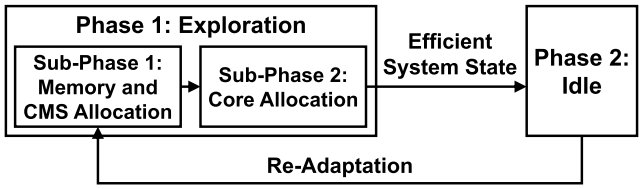


FIGURE 6. Execution flow of the system state space explorer.

B. SYSTEM STATE SPACE EXPLORER

The system state space explorer (SSSE) of COSMOS dynamically explores the system state space to discover a system state that delivers high throughput while providing strong QoS guarantees for the latency-critical (LC) container. It uses effective machine utilization (EMU), which is defined in Section IV, as the throughput metric.

As shown in Equation 3, we define a system state (i.e., s) as a vector of three elements (i.e., $r_{\text{LC,Cores}}$, $r_{\text{LC,M}}$, and $r_{\text{LC,CMS}}$), which denote the number of cores, the amount of memory, and the amount of CMS allocated to the LC container (see Section IV for the definitions of $r_{\text{LC,Cores}}$, $r_{\text{LC,M}}$, and $r_{\text{LC,CMS}}$). Note that the remaining resources are allocated to the batch container. We then define the system state space as the set of all the valid system states.

$$s = (r_{\text{LC,Cores}}, r_{\text{LC,M}}, r_{\text{LC,CMS}}) \tag{3}$$

Figure 6 shows the execution flow of the SSSE. In addition, Algorithm 1 shows the top-level function (i.e., `exploreSystemStateSpace`) executed by the SSSE. It operates in a periodic manner. In each period, the SSSE first allocates the resources to the consolidated containers (Lines 9–10 in Algorithm 1) based on the resource allocation plan encoded in the current system state through the resource allocator (Section V-C).[3] It then collects the runtime data (Lines 11–17 in Algorithm 1) – the load for the LC container, the tail latency of the LC container, the throughput of the batch container, the working-set sizes of the consolidated containers, and the average compression ratio of the pages stored in the CMS based on the profiler (Section V-A). If the current system state satisfies the LC container's QoS and achieves higher throughput than the highest throughput that has been discovered so far, it updates the best system state (i.e., s_{best}) to the current system state (Lines 24–27 in Algorithm 1). It comprises two phases – the (1) exploration and (2) idle phases.

[3]The period is set to one second in this work.

Algorithm 1 The exploreSystemStateSpace Function

```
1:  phase ← exploration; subphase ← memoryAndCMS
2:  L ← 0; Q ← 0; T ← 0; w_LC ← 0; w_Batch ← 0; γ_LC ← 1
3:  s_best ← s_invalid; T_best ← 0
4:  isBatchThroughputIncreased ← false
5:  procedure exploreSystemStateSpace
6:      createReclaimThreads()                          ▷ Create the memory reclaim threads
7:      s_next ← getInitialState()
8:      while true do
9:          s_curr ← s_next
10:         applySystemState(s_curr)
11:         sleep(τ)                                    ▷ Period: one second
12:         L ← getLoad()
13:         Q ← getTailLatency()
14:         T ← getBatchThroughput()
15:         w_LC ← getLCWorkingSetSize()
16:         w_Batch ← getBatchWorkingSetSize()
17:         γ_LC ← getCompressionRatio()
18:         if needToReadapt() = true then              ▷ Check if re-adaptation needs to be triggered
19:             resetVariables()                        ▷ Reset the global variables
20:             s_next ← getInitialState()
21:             phase ← exploration
22:         else
23:             if phase = exploration then
24:                 if isQoSSatisfied(Q) = true and T > T_best then
25:                     s_best ← s_curr
26:                     T_best ← T
27:                     isBatchThroughputIncreased ← true
28:                 else
29:                     isBatchThroughputIncreased ← false
30:                 end if
31:                 s_next ← getNextSystemState(s_curr)
32:                 if s_next = s_curr then
33:                     s_next ← s_best
34:                     phase ← idle
35:                 end if
36:             end if
37:         end if
38:     end while
39: end procedure
```

1) EXPLORATION PHASE

During the exploration phase, it gradually explores the system state space to find an efficient system state that achieves high throughput while satisfying the LC container's QoS. Specifically, in each period, it invokes the `getNextSystemState` function (Line 31 in Algorithm 1), which is shown in Algorithm 2.

The `getNextSystemState` function determines the system state, which is explored in the next period and expected to satisfy the LC container's QoS and achieve higher throughput than the current system state. The exploration phase comprises two sub-phases –(1) memory and CMS allocation sub-phase (Lines 3–13 in Algorithm 2) and (2) core allocation sub-phase (Lines 14–24). It begins with the memory and CMS allocation sub-phase.

The SSSE builds on the third and fourth observations (i.e., **C3** and **C4** in Section IV) from the characterization study. The third observation is that the throughput of the consolidated containers is likely to significantly improve when the working-sets of the consolidated containers fit in memory through the use of the CMS. Guided by this observation, it attempts to directly allocate the memory and CMS to the LC container in a way that satisfies the following requirements – (1) **R1**: the amounts of the memory and CMS allocated to the LC container are enough to hold the working-set of the LC container and (2) **R2**: the amount of

Algorithm 2 The getNextSystemState Function

```
1:  procedure getNextSystemState(s_curr)
2:      s_next ← s_curr
3:      if subphase = memoryAndCMS then
4:          if isInitialState(s_curr) = true then
5:              s_next ← setLCMemory(s_curr, max(M − w_Batch, w_LC/γ_LC))
6:          else
7:              if isQoSViolated(Q) = true then
8:                  s_next ← increaseLCMemory(s_curr)    ▷ Granularity: 5% of the working-set size of the LC container
9:              else
10:                 subphase ← core
11:                 s_next ← decreaseLCCore(s_curr)                                  ▷ Granularity: 1 core
12:             end if
13:         end if
14:     else
15:         if isQoSSatisfied(Q) = true then
16:             if isBatchThroughputIncreased = true then
17:                 s_next ← decreaseLCCore(s_curr)
18:             else
19:                 s_next ← s_curr
20:             end if
21:         else
22:             s_next ← increaseLCMemory(s_curr)
23:         end if
24:     end if
25:     if isMemoryChanged(s_curr, s_next) = true then    ▷ Check if the amount of memory allocated to LC will be changed
26:         r_LC,CMS,max ← getCMSMax(s_next, w_LC, γ_LC)                  ▷ Compute r_LC,CMS,max using Equation 2
27:         s_next ← setLCCMS(s_next, r_LC,CMS,max)
28:     end if
29:     return s_next
30: end procedure
```

the remaining memory is just enough to hold the working-set of the batch container (Lines 4–5 and 25–28 in Algorithm 2).[4] This design approach has an advantage of reducing the number of explored system states by skipping inefficient system states.

The SSSE then checks if the LC container's QoS is satisfied with this state. If the QoS is satisfied, it completes the memory and CMS allocation sub-phase and transitions to the core allocation sub-phase (Lines 9–12 in Algorithm 2).

If the LC container's QoS is violated, the SSSE gradually (i.e., 5% of the working-set size of the LC container) increases the amount of the memory and accordingly adjusts the amount of the CMS allocated to the LC container (Lines 7–8 and 25–28 in Algorithm 2). It repeats this process until the QoS is satisfied.

The SSSE transitions to the core allocation sub-phase after completing the memory and CMS allocation sub-phase. During the core allocation sub-phase, it gradually reclaims cores from the LC container and determines the right number of cores for the LC container, which is just enough to satisfy the QoS.

Specifically, the SSSE reduces the LC container's core count by one in each period (Lines 16–17 in Algorithm 2). If the LC container's QoS is satisfied even when allocated with the minimum number of cores or reducing the LC container's core count provides no throughput gain, it terminates the core allocation sub-phase and transitions to the idle phase (Lines 18–19 in Algorithm 2).

If the LC container's QoS is violated with the current core count, the SSSE keeps increasing the amount of the memory (Lines 21–23 in Algorithm 2) and accordingly decreasing the amount of the CMS (Lines 25–28) allocated to the LC container until the QoS is satisfied again. We have made this design decision based on the fourth observation from the characterization study. The observation is that the LC container tends to require a smaller number of cores when allocated with a larger amount of memory and a smaller amount of the CMS because of the decreased CPU utilization of the memory reclaim threads. If the size of the hot data of the batch container is smaller than its working-set, the throughput can be improved by allocating more cores to the

[4] If it is impossible to satisfy both **R1** and **R2** because the working-set sizes of the LC and batch containers are too large for the total memory capacity even with the use of the CMS, the SSSE utilizes the entire memory allocated to the LC container as the memory pool of the CMS in order to maximize the amount of the remaining memory, which is allocated to the batch container.

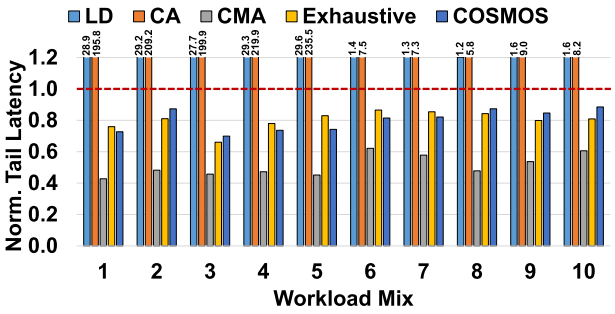


FIGURE 7. Quality of service.

batch container (and reducing the amount of the memory allocated to the batch container).

If the LC container's QoS is satisfied with an increased amount of memory and a decreased amount of the CMS, the SSSE attempts to reduce the LC container's core count. It repeats the aforementioned process and then transitions to the idle phase.

2) IDLE PHASE

As the SSSE enters the idle phase, it first sets the system state to the best system state (i.e., s_{best}) that achieves the highest throughput while satisfying the QoS among all the explored system states (Lines 32–35 in Algorithm 1). During the idle phase, it keeps monitoring the consolidated containers and the server system but performs no adaptation activities. If a change is detected (e.g., a significant increase or decrease in the load for the LC container), it transitions to the exploration phase and re-triggers the adaptation process (Lines 18–21 in Algorithm 1).

C. RESOURCE ALLOCATOR

The resource allocator of COSMOS dynamically allocates the resources based on the system state determined by the system state space explorer. The resource allocator builds on cgroups to dynamically allocate cores and memory to the consolidated containers. In addition, the resource allocator uses sysfs to adjust the amount of the CMS allocated to the latency-critical container at runtime.

VI. EVALUATION

In this section, we evaluate the effectiveness of COSMOS. Specifically, we aim to investigate the following – (1) QoS and throughput, (2) the sensitivity to the load and memory overcommit ratio, (3) the number of explored system states, and (4) the effectiveness of dynamic resource management.

A. QOS AND THROUGHPUT

We quantify the effectiveness of COSMOS in terms of QoS and throughput. To keep the discussion focused, we first report the experimental results collected with the medium load and medium memory overcommit ratio (MOR) that represent a common scenario in this section. We then report the experimental results collected with all the loads and MORs in Section VI-B.

TABLE 3. Evaluated workload mixes.

Mix	Benchmarks	Mix	Benchmarks
1	memcached and BC	6	silo and BC
2	memcached and BFS	7	silo and BFS
3	memcached and canneal	8	silo and canneal
4	memcached and CC	9	silo and CC
5	memcached and stream	10	silo and stream

Based on the two latency-critical (LC) benchmarks and five batch benchmarks discussed in Section III-B, we create 10 workload mixes. Each workload mix consists of the LC and batch containers, which contain the LC and batch benchmarks, respectively. Table 3 summarizes the 10 workload mixes evaluated in this work.

For each of the 10 workload mixes in Table 3, we execute it using the following resource allocation policies – (1) Linux default (LD), which employs the default resource allocation policy (i.e., no core or memory partitioning and compressed memory swap (CMS) enabled with the memory pool size of 20% of the total memory capacity) of Linux, (2) core allocation (CA), which dynamically allocates cores to the consolidated containers, (3) core and memory allocation (CMA), which dynamically allocates cores and memory to the consolidated containers and represents the approach adopted by PARTIES [11] with respect to memory management, (4) exhaustive, which executes the workload mix with a system state, which is discovered by exhaustively exploring the system state space through extensive offline profiling and exhibits the highest throughput while satisfying the QoS among all the explored system states,[5] and (5) COSMOS, which dynamically allocates cores, memory, and CMS to the consolidated containers based on COSMOS.

Figure 7 shows the normalized tail latency of the LC container in each of the 10 workload mixes with the medium load and medium MOR. Each bar in Figure 7 reports the tail latency normalized to the target tail latency of the LC container. We observe the following data trends.

First, the LD and CA versions fail to satisfy the LC container's QoS across all the workload mixes. The LD version violates the QoS because of the contention on the cores that are shared by the consolidated containers.

Since insufficient amounts of the memory and CMS are allocated to the LC container with the CA version, the working-set of the LC container does not fit in memory and the victim pages are evicted to the disk swap. Because of the

[5]Note that the exhaustive version is impractical because it requires highly time- and resource-consuming extensive offline profiling for each workload mix. Furthermore, the exhaustive version still requires separate extensive offline profiling for each of the datasets even for the same workload mix. This is because workload mixes tend to exhibit different characteristics with different datasets. Despite the impracticality of the exhaustive version, we compare COSMOS with the exhaustive version to demonstrate that COSMOS can achieve high throughput with strong QoS guarantees without requiring extensive offline profiling.

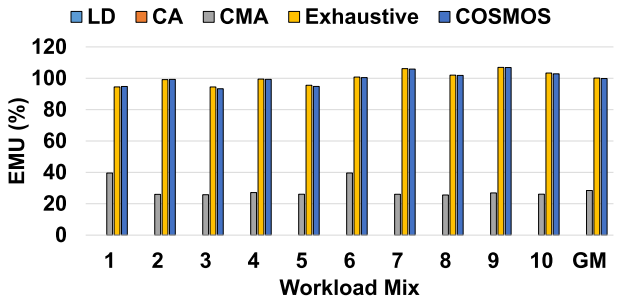


FIGURE 8. Effective machine utilization.

frequent accesses to the disk swap, the CA version violates the LC container's QoS.

Second, the CMA version satisfies the LC container's QoS across all the workload mixes. Since the number of cores and the amount of memory allocated to the LC container are sufficient, it satisfies the QoS. However, achieving tail latency that is significantly lower than the target tail latency is sub-optimal because it indicates that the LC container is allocated excessive resources, some of which could have been reallocated to the batch container to improve the overall throughput. As discussed later in this section (i.e., Figure 8), the CMA version delivers low throughput because of the excessive resources allocated to the LC container.

Third, COSMOS robustly satisfies the LC container's QoS across all the workload mixes. By dynamically analyzing the resource requirements of the LC container and allocating cores, memory, and CMS in a coordinated and efficient manner, COSMOS provides strong QoS guarantees for the LC container.

Fourth, the tail latency of COSMOS is closer to the target tail latency than that of the CMA version. COSMOS allocates a smaller amount of memory to the LC container through the use of the CMS than the CMA version in order to secure a sufficient amount of memory to the batch container. While it makes the tail latency of COSMOS closer to the target tail latency, it is an effective trade-off in that COSMOS still satisfies the LC container's QoS and significantly improves the throughput (as discussed later).

Figure 8 shows the EMU of various versions of the workload mixes. The rightmost bars denote the geometric mean of each version. First, the EMU of the LD and CA versions across all the workload mixes is zero. EMU is computed to be zero for the LD and CA versions because they fail to satisfy the LC container's QoS across all workload mixes.

Second, the CMA version delivers low throughput across the workload mixes. Since it lacks the capability of dynamically allocating the CMS to the LC container, it allocates a larger amount of memory to the LC container than the case in which the CMS is also used. Therefore, it allocates an insufficient amount of memory to the batch container, achieving low throughput.

The CMA version of the workload mixes 1 and 6 that include BC in the batch container exhibits higher EMU than other workload mixes. With BC, some of the nodes in the graph are accessed more frequently than others. Even if the CMA version allocates an insufficient amount of memory to BC, pages that contain more frequently accessed nodes are likely to remain in the memory without being evicted to the disk swap by the memory reclaim algorithm. This makes the CMA version of the workload mixes 1 and 6 access the disk swap less frequently, resulting in higher EMU.

Third, COSMOS consistently achieves high throughput across all the evaluated workload mixes. Specifically, COSMOS delivers 351.1% higher (on average) throughput than the CMA version and the throughput similar (i.e., 0.31% lower on average) to the exhaustive version, which uses a system state that is discovered through extensive offline profiling and achieves the highest throughput while satisfying the QoS. Our experimental results show the effectiveness of COSMOS in the sense that it is capable of dynamically finding an efficient system state with high throughput and QoS guarantees for each workload mix without the need for offline profiling.

COSMOS delivers higher EMU with the workload mixes that include silo (i.e., the workload mixes 6–10) in the LC container and BFS or CC (i.e., the workload mixes 2, 4, 7, and 9) in the batch container. As for silo, since it requires fewer cores to satisfy its QoS than memcached, COSMOS achieves higher EMU by allocating more cores to the batch container.

As for BFS and CC, they exhibit high throughput even when the amount of the memory allocated to them is smaller than their working-set size because some of the nodes are not accessed because of the connectivity of the graph. COSMOS dynamically identifies their characteristics and allocates a larger amount of the memory and a smaller amount of the CMS to the LC container. This reduces the CPU utilization of the memory reclaim threads because of less frequent data transfers between the memory and CMS. Since the LC container requires a smaller number of cores with the reduced CPU utilization of the memory reclaim threads, COSMOS achieves higher EMU by allocating more cores to the batch container (i.e., BFS or CC).

B. SENSITIVITY

We investigate the sensitivity of the QoS and throughput achieved by COSMOS to the load for the latency-critical (LC) and the memory overcommit ratio (MOR) of the consolidated containers. We first analyze the sensitivity to the MOR. Figure 9 shows the EMU of the CMA and exhaustive versions and COSMOS, which is averaged (using geometric mean) across all of the 10 workload mixes. The data reported in Figure 9 is collected with the medium load and by sweeping the MOR from low to high.

First, COSMOS robustly satisfies the LC container's QoS across all the workload mixes and MORs. Note that it would be impossible to compute the average EMU using geometric mean if COSMOS failed to satisfy the QoS with any of

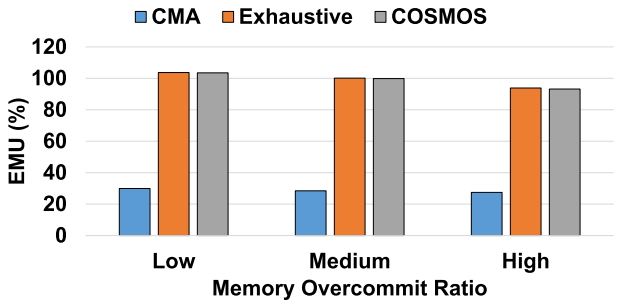


FIGURE 9. Sensitivity to the memory overcommit ratio.

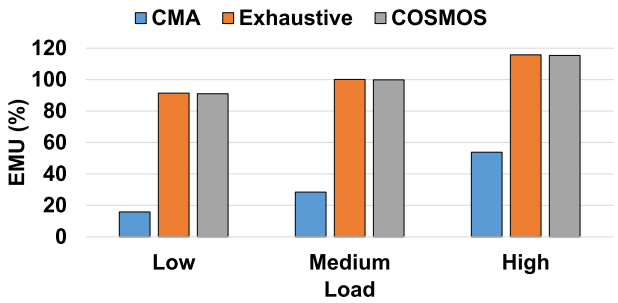


FIGURE 10. Sensitivity to the load for the LC container.

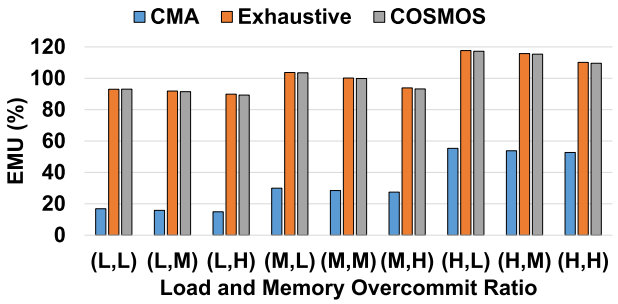


FIGURE 11. Sensitivity to the load and memory overcommit ratio.

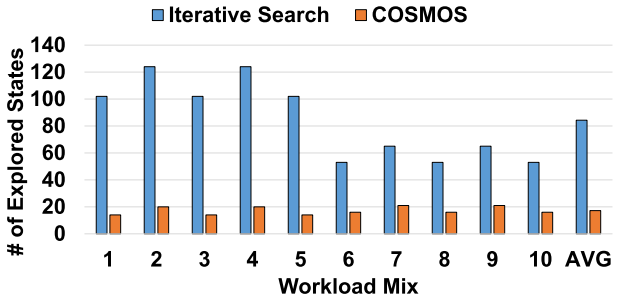


FIGURE 12. Number of the explored system states.

the workload mixes and MORs because EMU would be computed to be zero.

Second, COSMOS consistently achieves high throughput across all the workload mixes and MORs. Specifically, COSMOS significantly outperforms the CMA version and delivers the throughput similar to the exhaustive version which executes each of the workload mixes with a system state that is discovered through extensive offline profiling and achieves the highest throughput.

Third, the throughput of COSMOS gradually decreases as the MOR increases. This is mainly because COSMOS allocates a smaller amount of memory and a larger amount of CMS to the LC container with a higher MOR to make the working-set of the batch container fit in memory. With a larger amount of the CMS allocated to the LC container, it requires a larger number of cores because the CPU utilization of the memory reclaim threads increases. Consequently, COSMOS allocates a smaller number of cores to the batch container with a higher MOR, resulting in lower throughput.

We now investigate the sensitivity to the load for the LC container. Figure 10 shows the EMU of the CMA and exhaustive versions and COSMOS, which is averaged (using geometric mean) across the workload mixes. The data reported in Figure 10 is collected by sweeping the load from low to high with the medium MOR.

First, similarly to the sensitivity data trends with the MOR, COSMOS robustly satisfies the LC container's QoS and achieves high throughput across all the workload mixes and loads. Specifically, COSMOS significantly outperforms the CMA version and delivers the EMU similar to the exhaustive version which discovers an efficient system state through extensive offline profiling.

Second, the throughput of COSMOS gradually increases as the load for the LC container increases. Since the portion of the EMU contributed by the LC container increases with a higher load, the overall EMU also increases.

For completeness, we report the EMU of the CMA and exhaustive versions and COSMOS across all the loads and MORs in Figure 11. L, M, and H in Figure 11 indicate low, medium, and high, respectively. We observe that COSMOS exhibits similar sensitivity trends when the load or MOR is fixed at low or high to the ones (i.e., the load or MOR is fixed at medium) shown in Figures 9 and 10. In addition, COSMOS consistently achieves high throughput across all the evaluated loads and MORs.

C. EXPLORED SYSTEM STATES

We investigate the impact of the optimization (i.e., skipping inefficient system states) applied to COSMOS on the number of explored system states. To this end, we create a synthetic version of COSMOS that iteratively explores the system state space without applying the optimization (Section V-B).

Figure 12 shows the number of the system states explored by the synthetic version and COSMOS with the medium load and medium memory overcommit ratio (MOR). The optimization applied to COSMOS is effective for reducing the number of the explored system states. Specifically, the

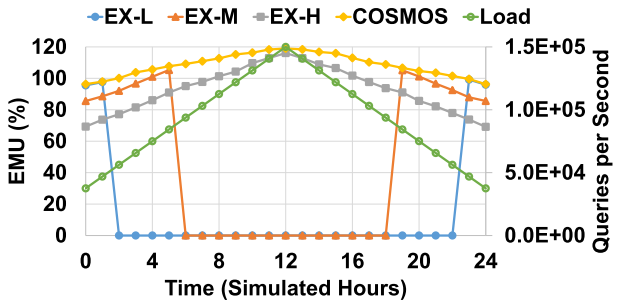


FIGURE 13. Effectiveness of dynamic resource management.

full version of COSMOS explores 79.6% fewer system states on average than the synthetic version.

D. DYNAMIC RESOURCE MANAGEMENT

We investigate the effectiveness of dynamic resource management supported by COSMOS. We use `memcached` (i.e., the latency-critical (LC) container) and `stream` (i.e., the batch container) with the medium memory overcommit ratio (MOR). The load that the load generator applies to the LC container dynamically varies over the time between the low and high loads in Table 1 by following the diurnal pattern, which is commonly observed in production datacenters [7], [26], [30], [47]. The load generator simulates a 24-hour period. Specifically, the load generator is configured to make each hour in the 24-hour diurnal pattern correspond to four minutes for keeping the total experiment time manageable.

We execute the consolidated containers using four resource allocation policies – (1) `EX-L`, which statically allocates cores, memory, and compressed memory swap to the consolidated containers based on a system state that is discovered through exhaustive search (with extensive offline profiling) when the LC container is applied with the low load, (2) `EX-M`, which statically allocates resources based on a system state discovered through exhaustive search with the medium load, (3) `EX-H`, which statically allocates resources based on a system state discovered through exhaustive search with the high load, and (4) COSMOS, which dynamically allocates resources using COSMOS. Figure 13 shows the EMU of the four versions. We observe the following data trends.

First, the `EX-L` and `EX-M` versions fail to satisfy the LC container's QoS (i.e., EMU $=$ 0) as the load applied to the LC container increases. Since the `EX-L` (or `EX-M`) version statically allocates resources based on a system state discovered through exhaustive search with the low (or medium) load, it allocates insufficient resources to the LC container when the load is relatively high. This results in QoS violations with relatively high loads.

Second, the `EX-H` version exhibits relatively low throughput as the load applied to the LC container decreases. Since the `EX-H` version statically allocates resources based on a system state discovered through exhaustive search with the high load, it allocates excessive resources to the LC container and delivers low throughput when the load is relatively low.

Third, COSMOS robustly satisfies the LC container's QoS and achieves high throughput across all the evaluated loads. When COSMOS detects a change in the load, it re-triggers the adaptation process to find a system state that exhibits high throughput while satisfying the QoS with the changed load and dynamically allocates resources to the consolidated containers based on the newly-discovered system state. With its re-adaptation capability, COSMOS achieves high throughput with strong QoS guarantees across all the evaluated loads.

Overall, our experimental results demonstrate the effectiveness of COSMOS in that it robustly satisfies the LC container's QoS and delivers high throughput of the consolidated containers across all the evaluated workload mixes, loads, and MORs and significantly reduces the number of explored system states.[6]

VII. RELATED WORK

Prior works have extensively investigated resource management techniques for workload consolidation [11], [15], [16], [27], [31], [32], [33], [34], [46], [49]. Most of the prior works focus on partitioning resources such as cores and memory bandwidth and lack the consideration of dynamically partitioning the memory capacity between the consolidated applications [15], [16], [27], [31], [32], [33], [34], [46], [49]. Our work differs in that it investigates system software support for coordinated management of cores, memory, and compressed memory swap (CMS) for QoS-aware and efficient workload consolidation for memory-intensive applications.

The technique proposed in [11] considers memory capacity partitioning between the consolidated applications. While insightful, it lacks the consideration of dynamic allocation of the CMS, which is crucial to improve the throughput of the consolidated applications with strong QoS guarantees for the latency-critical application. Our work is significantly different in the sense that it presents the in-depth characterization of the impact of cores, memory, and CMS on the QoS and throughput of the consolidated applications and design, implement, and evaluate a software-based runtime system for QoS-aware and efficient workload consolidation for memory-intensive applications based on a real system.

Prior works have explored memory offloading techniques based on the CMS [23], [45]. While the prior works have a similarity to our work in that they show that the CMS can effectively be used to improve the resource efficiency of cloud computing systems and datacenters, they lack the consideration of dynamic management of the CMS in the context of workload consolidation. Our work significantly differs in the sense that it investigates coordinated resource

[6]Our experimental results also show that the performance overhead of COSMOS is small. Specifically, its CPU utilization is 0.27% on average, which is low.

management of cores, memory, and CMS for QoS-aware and efficient workload consolidation.

Prior works have studied the feasibility of the CMS for embedded applications [17], [20], [25], [42]. The prior works show that the CMS can be used to reduce the response time and improve the interactivity of applications on embedded systems with limited memory capacity. However, they lack the capability of holistically allocating the resources to the consolidated applications. Our work is considerably different in that it proposes a software-based runtime system that dynamically allocates cores, memory, and CMS to the consolidated applications in a coordinated manner for QoS-aware and efficient workload consolidation.

VIII. CONCLUSION

In this work, we present the in-depth characterization of the impact of cores, memory, and compressed memory swap (CMS) on the QoS and throughput of the consolidated applications. Guided by the characterization results, we propose COSMOS, a software-based runtime system for QoS-aware and efficient workload consolidation for memory-intensive applications. Our quantitative evaluation based on a real system and widely-used benchmarks demonstrates that COSMOS robustly satisfies the QoS and achieves high throughput across all the evaluated workload mixes and scenarios and significantly reduces the number of explored system states by skipping inefficient system states. As future work, we plan to investigate lightweight and flexible architectural support for CMS. We would also like to explore the design and implementation of efficient compression and decompression algorithms for CMS.

REFERENCES

[1] *Cgroups(7)—Linux Manual Page*. Accessed: Nov. 6, 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/cgroups.7.html

[2] *Memcached—A Distributed Memory Object Caching System*. Accessed: Nov. 6, 2023. [Online]. Available: https://memcached.org/

[3] *Memory Compression in Windows 10 RTM*. Accessed: Nov. 6, 2023. [Online]. Available: https://learn.microsoft.com/en-us/shows/seth-juarez/memory-compression-in-windows-10-rtm

[4] Oberhumer.com. *LZO Real-Time Data Compression Library*. Accessed: Nov. 6, 2023. [Online]. Available: http://www.oberhumer.com/opensource/lzo/

[5] *OS X Mavericks Core Technologies Overview*. Accessed: Nov. 6, 2023. [Online]. Available: https://images.apple.com/media/us/osx/2013/docs/OSX_Mavericks_Core_Technology_Overview.pdf

[6] *Zswap*. Accessed: Nov. 6, 2023. [Online]. Available: https://www.kernel.org/doc/html/v4.18/vm/zswap.html

[7] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Renton, WA, USA: USENIX Association, Apr. 2018, pp. 405–417.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE joint Int. Conf. Meas. Model. Comput. Syst. (SIGMETRICS)*. New York, NY, USA: Association for Computing Machinery, Jun. 2012, pp. 53–64.

[9] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," 2015, *arXiv:1508.03619*.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Architectures Compilation Techn. (PACT)*, New York, NY, USA, Oct. 2008, pp. 72–81.

[11] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.* New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 107–120.

[12] S. Chen, A. Jin, C. Delimitrou, and J. F. Martínez, "ReTail: Opting for learning simplicity to enable QoS-aware power management in the cloud," in *Proc. IEEE Int. Symp. High-Performance Comput. Archit. (HPCA)*, Apr. 2022, pp. 155–168.

[13] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: Adaptive DVFS and thread packing under power caps," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2011, pp. 175–185.

[14] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, p. 5, Aug. 2004.

[15] M. Han, S. Yu, and W. Baek, "Secure and dynamic core and cache partitioning for safe and efficient server consolidation," in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2018, pp. 311–320.

[16] M. Han and W. Baek, "SDRP: Safe, efficient, and SLO-aware workload consolidation through secure and dynamic resource partitioning," *IEEE Trans. Services Comput.*, vol. 15, no. 4, pp. 1868–1882, Jul. 2022.

[17] J. Hwang, J. Jeong, H. Kim, J. Choi, and J. Lee, "Compressed memory swap for QoS of virtualized embedded systems," *IEEE Trans. Consum. Electron.*, vol. 58, no. 3, pp. 834–840, Aug. 2012.

[18] S. Jennings, "Transparent memory compression in Linux," Presented at LinuxCon North America, New Orleans, LA, USA, Sep. 2013.

[19] H. Kasture and D. Sanchez, "Tailbench: A benchmark suite and evaluation methodology for latency-critical applications," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10.

[20] J. Kim, C. Kim, and E. Seo, "*ezswap*: Enhanced compressed swap scheme for mobile devices," *IEEE Access*, vol. 7, pp. 139678–139691, 2019.

[21] M. Kogias, S. Mallon, and E. Bugnion, "Lancet: A self-correcting latency measuring tool," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*. Renton, WA, USA: USENIX Association, Jul. 2019, pp. 881–896.

[22] N. Kulkarni, F. Qi, and C. Delimitrou, "Pliant: Leveraging approximation to improve datacenter resource efficiency," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 159–171.

[23] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.* New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 317–330.

[24] S.-H. Lee, "Technology scaling challenges and opportunities of memory devices," in *IEDM Tech. Dig.*, Dec. 2016, pp. 1.1.1–1.1.8.

[25] C. Li, L. Shi, Y. Liang, and C. J. Xue, "SEAL: User experience-aware two-level swap for mobile devices," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 4102–4114, Nov. 2020.

[26] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 301–312.

[27] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, Jun. 2015, pp. 450–462.

[28] C. A. Mack, "Fifty years of Moore's law," *IEEE Trans. Semicond. Manuf.*, vol. 24, no. 2, pp. 202–207, May 2011.

[29] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Comput. Soc. Tech. Committee Comput. Archit. (TCCA) Newslett.*, Dec. 1995, pp. 19–25.

[30] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proc. 38th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, Jun. 2011, pp. 319–330.

[31] K. Nikas, N. Papadopoulou, D. Giantsidi, V. Karakostas, G. Goumas, and N. Koziris, "DICER: Diligent cache partitioning for efficient workload consolidation," in *Proc. 48th Int. Conf. Parallel Process.* New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 1–10.

[32] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 167–179.

[33] J. Park, S. Park, and W. Baek, "CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proc. 14th EuroSys Conf.*, New York, NY, USA, Mar. 2019, pp. 10:1–10:16.
[34] J. Park, S. Park, M. Han, J. Hyun, and W. Baek, "Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn.*, New York, NY, USA, Nov. 2018, pp. 5:1–5:14.
[35] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving low tail latency for microsecond-scale networked tasks," in *Proc. 26th Symp. Oper. Syst. Princ.* New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 325–341.
[36] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via OS level monitoring," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Nov. 2011, pp. 116–125.
[37] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "ZeRO-offload: Democratizing billion-scale model training," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*. Berkeley, CA, USA: USENIX Association, Jul. 2021, pp. 551–564.
[38] F. Romero and C. Delimitrou, "Mage: Online and interference-aware scheduling for multi-scale heterogeneous systems," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn.* New York, NY, USA: Association for Computing Machinery, Nov. 2018, pp. 1–13.
[39] S. Roy, R. Kumar, and M. Prvulovic, "Improving system performance with compressed memory," in *Proc. 15th Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2001, p. 7.
[40] A. I. Saichev, Y. Malevergne, and D. Sornette, *Theory of Zipf's Law and Beyond* (Lecture Notes in Economics and Mathematical Systems). Berlin, Germany: Springer, 2009.
[41] H. Sasaki, S. Imamura, and K. Inoue, "Coordinated power-performance optimization in manycores," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, Sep. 2013, pp. 51–61.
[42] T. Song, M. Kim, G. Lee, and Y. Kim, "Prediction-guided performance improvement on compressed memory swap," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2022, pp. 1–6.
[43] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps," in *Proc. 13th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, New York, NY, USA: Association for Computing Machinery, 2008, pp. 277–286.
[44] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. 24th ACM Symp. Oper. Syst. Princ.* New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 18–32.
[45] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "TMO: Transparent memory offloading in datacenters," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.* New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 609–621.
[46] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, Jun. 2013, pp. 607–618.
[47] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in *Proc. 14th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*. New York, NY, USA: USENIX Association, Nov. 2020, pp. 191–208.
[48] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX Association, 2012, p. 2.
[49] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems," in *Proc. 21st Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, New York, NY, USA, 2016, pp. 33–47.

MYEONGGYUN HAN received the bachelor's degree from UNIST, in 2018, where he is currently pursuing the integrated M.S. and Ph.D. degrees with the Department of Computer Science and Engineering. His research interests include parallel and distributed computing and computer systems security.

EUNSEONG PARK, photograph and biography not available at the time of publication.

YOUNGSAM SHIN received the B.S. degree from the Hankuk University of Foreign Studies and the M.S. degree from the Pohang University of Science and Technology, in 2001. He is currently a Staff Researcher with the Samsung Advanced Institute of Technology. His research interests include computer architecture, system software, and hardware and software co-optimization.

DEOK-JAE OH received the B.S. degree in electrical engineering from Hongik University and the Ph.D. degree from the Graduate School of Convergence Science and Technology, Seoul National University, in 2022. He is currently a Staff Researcher with the Samsung Advanced Institute of Technology. His research interests include computer architecture, processing near memory, and hardware and software co-optimization.

YEONGON CHO received the M.S. degree from the Department of Electrical and Computer Engineering, Sungkyunkwan University. He is currently a Principal Researcher with the Samsung Advanced Institute of Technology. He has developed in-house Samsung reconfigurable processor and mobile GPU. He is also leading processing in/near memory project. His research interests include computer architecture and microarchitecture.

WOONGKI BAEK (Member, IEEE) received the B.S. and M.S. degrees from Seoul National University and the Ph.D. degree in electrical engineering from Stanford University, in 2011. He is currently a Professor with the Department of Computer Science and Engineering, UNIST. His research interests include architecture, system software, and programming models for parallel and distributed computing. He is a member of ACM.