PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services

Shuang Chen
Cornell University
sc2682@cornell.edu

Christina Delimitrou
Cornell University
delimitrou@cornell.edu

José F. Martínez
Cornell University
martinez@cornell.edu

Abstract

Multi-tenancy in modern datacenters is currently limited to a single latency-critical, interactive service, running alongside one or more low-priority, best-effort jobs. This limits the efficiency gains from multi-tenancy, especially as an increasing number of cloud applications are shifting from batch jobs to services with strict latency requirements.

We present PARTIES, a QoS-aware resource manager that enables an arbitrary number of interactive, latency-critical services to share a physical node without QoS violations. PARTIES leverages a set of hardware and software resource partitioning mechanisms to adjust allocations dynamically at runtime, in a way that meets the QoS requirements of each co-scheduled workload, and maximizes throughput for the machine. We evaluate PARTIES on state-of-the-art server platforms across a set of diverse interactive services. Our results show that PARTIES improves throughput under QoS by 61% on average, compared to existing resource managers, and that the rate of improvement increases with the number of co-scheduled applications per physical host.

CCS Concepts • Computer systems organization → Cloud computing; Real-time system architecture.

Keywords Cloud computing, datacenters, quality of service, resource management, resource partitioning, isolation, interference

1 Introduction

Cloud computing has become ubiquitous by offering resource flexibility and cost efficiency [9, 11, 31]. Resource flexibility is achieved as users elastically scale their resources on demand. Cost efficiency is achieved through multi-tenancy, i.e., by scheduling jobs from multiple users on the same physical host to increase utilization. Unfortunately, multi-tenancy often comes at a performance penalty, as co-scheduled applications contend for shared resources, leading to interference and performance unpredictability. Interference is particularly destructive for interactive, latency-critical (LC) services, which must meet strict quality of service (QoS) guarantees.

Prior work has tackled interference in three ways. The first approach is to simply disallow interactive services from sharing resources with other applications to avoid interference [42, 43, 51, 53]. This preserves the QoS of the LC applications, but lowers the resource efficiency of the system. The second approach is to avoid co-scheduling applications that are likely to interfere with each other [19–22, 24, 44]. This improves utilization, although it limits the options of applications that can be co-scheduled. The third approach focuses on eliminating interference altogether, by partitioning resources among co-scheduled services, using OS- and hardware-level isolation techniques [23, 35, 36, 42, 43, 49]. This approach protects QoS for the LC service, and allows best-effort (BE) workloads to benefit from unused resources. Unfortunately, this approach is currently limited to at most one interactive service per physical host, co-scheduled with one or more BE jobs. Alternatively, if multiple interactive applications are co-scheduled on a physical host, their load is dialed down considerably, leading to underutilization [61].

Cloud applications are progressively shifting from batch to low-latency services. For example, traditionally throughput-bound applications, like big data and graph analytics, are now moving to in-memory computation, with frameworks like Spark [60] and X-Stream [48], which brings task execution latencies to a few milliseconds or seconds. Furthermore, cloud applications are undergoing a major redesign from large, monolithic services that encompass the entire application functionality in a single binary, to hundreds or thousands of loosely-coupled microservices [28–30, 52]. While the end-to-end latency of a large-scale service remains in the granularity of several milliseconds or seconds, each microservice must meet much tighter latency constraints, often...
in the order of a few hundreds of microseconds. Additionally, each microservice resides in a small, mostly stateless container, which means that many containers need to be scheduled on one physical host to maximize utilization. Consequently, techniques that only allow one high-priority LC service per machine are not general enough to manage these new application scenarios.

In this paper, we present PARTIES (PARTitioning for multiple Interactive Services), a cloud runtime system that allows two or more LC services to meet their QoS constraints while sharing a highly-utilized physical host. PARTIES leverages an online monitoring framework that operates at the granularity of a few hundred milliseconds, to quickly detect QoS violations. Upon detection, the runtime boosts the allocation of one or more resources for the LC service whose latency suffers the most. PARTIES assumes no a priori knowledge of incoming applications, making it applicable in settings like public clouds where user-submitted applications are not known in advance. PARTIES uses both OS- and hardware-level partitioning mechanisms available in modern platforms, including containers, thread pinning, cache partitioning, frequency scaling, memory capacity partitioning, and disk and network bandwidth partitioning to satisfy the instantaneous resource needs of each co-scheduled interactive service.

Finding the optimal resource allocation for each LC service over time requires exhaustive allocation exploration, which quickly becomes intractable. Instead, PARTIES ensures fast convergence by exploiting the observation that certain resources are fungible, i.e., can be traded off with each other, to only explore a few allocations before arriving to a satisfactory decision. PARTIES is dynamic, adjusting its decisions to fluctuating load, without the need for resource overprovisioning. Once all services meet their QoS targets, PARTIES additionally optimizes for server utility by progressively reclaiming excess resources from each LC application, which can potentially be allocated to BE jobs.

We first characterize the sensitivity of six popular and diverse, open-source LC services to different resource allocations, and to interference in shared resources, and show that resource isolation is both essential and effective at reducing contention. We then introduce the concept of resource fungibility, i.e., the fact that resources can be traded for each other to arrive to equivalent application performance. Fungibility improves the controller’s flexibility and convergence speed. We evaluate PARTIES on a high-end server platform across a diverse mix of LC services and input loads. We compare PARTIES to Heracles, a controller designed for a single LC service and multiple BE jobs, and show that PARTIES operates the server at near-capacity, and achieves up to 61% higher aggregate throughput, while meeting the QoS target of each LC service. PARTIES allows an arbitrary number of LC jobs to be co-scheduled, increasing the cluster manager’s flexibility, and making it applicable for scenarios where large numbers of microservices share hardware resources.

## Related Work

Improving the resource efficiency of multicore systems through application colocation has been a very active research field over the past few years [12, 25, 34, 56]. These approaches typically account for resource contention, although they are geared towards batch applications, and optimize for long-term goals (e.g., throughput or fairness). As such, they are not directly applicable to interactive services that must meet short-term tail latency constraints.

Past work on improving resource efficiency in datacenters falls into two categories. First, there are cluster schedulers that infer the expected interference of a given application colocation [13, 18, 20–22, 24, 33, 45, 58, 59, 61, 62], and either adjust allocations at runtime, or completely disallow resource sharing when the predicted latency violates QoS. While this approach protects the QoS of LC services, it is overly conservative, and limits the set of applications that can share a physical node. The second approach proposes fine-grained resource partitioning mechanisms that altogether eliminate interference [35, 36, 43, 49, 55, 57]. These techniques achieve more aggressive resource sharing, but either require microarchitectural changes, which are not readily available in production systems, or target batch applications.

The most relevant work to PARTIES is Heracles [43], a multi-resource controller that leverages a set of hardware and software isolation mechanisms to improve server utilization by colocating a single interactive service with one or more BE workloads. Both PARTIES and Heracles rely on resource partitioning to guarantee cross-application isolation, with PARTIES additionally supporting memory capacity and disk bandwidth isolation. However, while Heracles is designed to manage a single LC application, PARTIES provides a general scheduling framework that manages an arbitrary number of co-scheduled interactive services. We provide a detailed comparison of the two schemes in Section 5.

## Characterization

To quantify the impact of resource interference and allocation, we study six popular, open-source LC services. Table 1 shows the specs of our experimental platform. 8 physical cores are exclusively allocated to network interrupts (IRQ cores) per socket. This is the minimum core count needed

<table>
<thead>
<tr>
<th>Table 1. Platform Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
</tr>
<tr>
<td><strong>OS</strong></td>
</tr>
<tr>
<td><strong>Virtualization Technology</strong></td>
</tr>
<tr>
<td><strong>Sockets</strong></td>
</tr>
<tr>
<td><strong>Cores/Socket</strong></td>
</tr>
<tr>
<td><strong>Threads/Core</strong></td>
</tr>
<tr>
<td><strong>Base/Max Turbo Frequency</strong></td>
</tr>
<tr>
<td><strong>Default Frequency Driver</strong></td>
</tr>
<tr>
<td><strong>L1 Inst/Data Cache</strong></td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
</tr>
<tr>
<td><strong>L3 (Last-Level) Cache</strong></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td><strong>Disk</strong></td>
</tr>
<tr>
<td><strong>Network Bandwidth</strong></td>
</tr>
</tbody>
</table>
to handle network interrupts across the server socket when operating at max load. Allowing LC threads to share cores with IRQ cores leads to both lower throughput and higher latency [16]. 8GB of memory is exclusively allocated to the OS. Each application is instantiated in a separate container. Finally, we enable hyperthreading and Turbo boosting.

### 3.1 Latency-critical applications

We characterize six open-source interactive applications:

- **Memcached** [27] is a high-performance memory object caching system. Such in-memory key-value stores have become a critical tier in cloud services that optimize for low latency [39–41]. We use Memcached 1.4.36 compiled from source [1], and configure its dataset to hold 32 million items, each with a 30B key and a 200B value.

- **Xapian** [6] is a web search engine included in the Tailbench suite [37]. We follow Tailbench’s setup to configure Xapian to represent a leaf node in a distributed web search service. The node’s indexes are built from a snapshot of the English version of Wikipedia.

- **NGINX** [3] is a high-performance HTTP server, currently responsible for over 41% of live websites (circa Jan 2019 [4]). We use NGINX 1.12.0 compiled from source, and set it up as a front-end serving static files. The input dataset consists of one million html files of 1KB each.

- **Moses** [38] implements a statistical machine translation (SMT) system, a vital component of online translation systems and intelligent personal assistants (IPA), and is configured as outlined in Tailbench [37].

- **MongoDB** [2] is one of the most popular NoSQL database systems, and is widely used in industry for back-end data storage [26]. We use MongoDB 3.2.16 compiled from source, and compose a dataset with one billion records, each with 10 fields and 100B per field.

- **Sphinx** [54] is a speech recognition system with acoustic, phonetic, and language models, configured as in [37].

To quantify the maximum input load the server can sustain and how latency reacts to increasing load, we start from low request-per-second (RPS) and gradually inject higher request rates, until the application starts dropping requests on the server side. Figure 1 shows the relationship between tail latency and RPS. All applications experience a rapid increase in tail latency after exceeding a load threshold, typically between 60% and 80% of their maximum RPS. We therefore set the QoS target of each application as the 99\textsuperscript{th} percentile latency of the knee, as marked in Figure 1. We denote the RPS at the knee in each case as max load, which is the maximum throughput the machine can sustain while meeting QoS.

Table 2 reports the QoS target in terms of 99\textsuperscript{th} percentile (tail) latency, max load (maximum RPS achieved under the QoS target), and various microarchitectural characteristics for each application at max load. The six applications have a diverse set of characteristics: their QoS targets range from microseconds to seconds; they involve different amounts of user-space, kernel-space, and I/O processing; their instruction and data footprints vary widely; and they vary in their memory, disk, and network bandwidth demands. This ensures a high coverage of the design space of cloud services.

### 3.2 Testing strategy

We use open-loop workload generators as clients for all applications to ensure that latency measurements at high loads...
Table 3. List of experimental setups for studying resource interference (left), and isolation mechanisms per resource (right).

<table>
<thead>
<tr>
<th>Shared Resource</th>
<th>Method of Generating Interference</th>
<th>Isolation Mechanism</th>
<th>Software/Hardware Isolation Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyper-thread</td>
<td>8 compute-intensive microbenchmarks are colocated on the same hyperthreads as LC applications.</td>
<td>Core Isolation</td>
<td>Linux’s <code>cpuset cgroup</code> is used to allocate specific core IDs to a given application.</td>
</tr>
<tr>
<td>CPU</td>
<td>8 compute-intensive microbenchmarks are colocated on the same physical cores as the LC applications, but different hyperthreads.</td>
<td>Power Isolation</td>
<td>The ACPI frequency driver with the “userspace” governor to set selected cores to a fixed frequency (1.2-2.2GHz with 100MHz increments), or with the “performance” governor to run at turbo frequency (the highest possible frequency based on load, power consumption, CPU temperature, etc [15]).</td>
</tr>
<tr>
<td>Power</td>
<td>12 compute-intensive microbenchmarks (power viruses) are mapped on the 12 logical cores of the 6 idle CPUs.</td>
<td>LLC Isolation</td>
<td>Intel’s Cache Allocation Technology (CAT) [7, 32] is used for LLC way partitioning. It indirectly regulates memory bandwidth as well because memory traffic is highly correlated with cache hit rate. There is no mechanism available on the evaluated server platform to partition memory bandwidth directly.</td>
</tr>
<tr>
<td>LLC Capacity</td>
<td>We launch a cache-thrashing microbenchmark which continuously streams an array the size of the LLC. It runs on an idle core in the same socket as the LC application.</td>
<td>Memory Isolation</td>
<td>Linux’s <code>memory cgroup</code> is used to limit the maximum amount of memory capacity per container.</td>
</tr>
<tr>
<td>LLC Bandwidth</td>
<td>We launch 12 cache-thrashing microbenchmarks whose aggregate footprint is the size of the LLC, i.e., each thrashes 1/12 of the LLC.</td>
<td>Memory Isolation</td>
<td>Linux’s <code>memory cgroup</code> is used to limit the maximum amount of memory capacity per container.</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>We launch 12 memory-thrashing microbenchmarks, generating 50GB/s of memory traffic (upper limit on the evaluated machine).</td>
<td>Memory Isolation</td>
<td>Linux’s <code>memory cgroup</code> is used to limit the maximum amount of memory capacity per container.</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>We launch one memory-thrashing microbenchmark that streams 120GB out of the 128GB memory (8GB is reserved for OS).</td>
<td>Memory Isolation</td>
<td>Linux’s <code>memory cgroup</code> is used to limit the maximum amount of memory capacity per container.</td>
</tr>
<tr>
<td>Disk Bandwidth</td>
<td>We launch <code>dd</code> with <code>of=/dev/null</code>.</td>
<td>Disk Isolation</td>
<td>Linux’s <code>blkio cgroup</code> is used to throttle the maximum disk read bandwidth per container.</td>
</tr>
<tr>
<td>Network Bandwidth</td>
<td>We launch one <code>iperf3</code> client on an idle core, and direct its traffic to an idle machine running the <code>iperf3</code> server using 100 connections, each at 100Mbps bandwidth.</td>
<td>Network Isolation</td>
<td>Linux’s <code>qdisc</code> traffic control scheduler [14] with hierarchical token bucket (HTB) queuing discipline is used to limit the egress network bandwidth.</td>
</tr>
</tbody>
</table>

Table 4. Impact of resource interference. Each row corresponds to one type of resource. Values in the table are the maximum percentage of `max load` for which the server can satisfy QoS when the LC application is running under interference. Cells with smaller numbers/darker colors mean that applications are more sensitive to that type of interference.

<table>
<thead>
<tr>
<th>Hyper-thread</th>
<th>Memcached</th>
<th>Xapian</th>
<th>NGINX</th>
<th>Moses</th>
<th>MongoDB</th>
<th>Sphinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>10%</td>
<td>50%</td>
<td>60%</td>
<td>70%</td>
<td>100%</td>
<td>30%</td>
</tr>
<tr>
<td>Power</td>
<td>60%</td>
<td>80%</td>
<td>90%</td>
<td>90%</td>
<td>100%</td>
<td>70%</td>
</tr>
<tr>
<td>LLC Capacity</td>
<td>90%</td>
<td>90%</td>
<td>70%</td>
<td>80%</td>
<td>100%</td>
<td>30%</td>
</tr>
<tr>
<td>LLC Bandwidth</td>
<td>0%</td>
<td>60%</td>
<td>70%</td>
<td>30%</td>
<td>90%</td>
<td>0%</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>0%</td>
<td>50%</td>
<td>40%</td>
<td>10%</td>
<td>70%</td>
<td>0%</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>20%</td>
<td>80%</td>
</tr>
<tr>
<td>Disk Bandwidth</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Network Bandwidth</td>
<td>100%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>80%</td>
</tr>
</tbody>
</table>

are accurate [50, 63]. For Memcached, we use an in-house load generator, similar to mutilate [46], but converted to open-loop. For NGINX and MongoDB, we modified popular open-source generators, wrk2 [5] and YCSB [17], from closed- to open-loop. For Moses, Sphinx and Xapian, we use the open-loop load generators provided by Tailbench [37]. All the load generators use exponential inter-arrival time distributions [39] to simulate a Poisson process, where requests are sent continuously and independently at a constant average rate. We also use a Zipfian distribution for the request popularity [8, 47], and limit input loads to read-only, which correspond to the majority of requests in production.

To understand the impact of interference on tail latency, we colocate each LC workload with microbenchmarks [18] that stress different parts of the system. To ensure sufficient resources for the contentious microbenchmarks, we instantiate each application with 8 threads pinned to 8 hyperthreads on 8 different physical cores. Excluding the 8 IRQ cores, another 6 physical cores with 12 hyperthreads are available to the microbenchmarks. We study 9 shared resources in total, detailed in Table 3.

3.3 Interference Study

To understand the impact of interference on tail latency, we colocate each LC workload with microbenchmarks [18] that stress different parts of the system. To ensure sufficient resources for the contentious microbenchmarks, we instantiate each application with 8 threads pinned to 8 hyperthreads on 8 different physical cores. Excluding the 8 IRQ cores, another 6 physical cores with 12 hyperthreads are available to the microbenchmarks. We study 9 shared resources in total, detailed in Table 3.

3.3.1 Interference Analysis

Table 4 shows the diverse impact of resource interference across the six applications. Usually applications are most sensitive to resources whose utilization they saturate. For instance, Moses and Sphinx have high demand for cache capacity and memory bandwidth, seen by their high LLC MPKI and memory bandwidth usage in Table 2. Interference
in these resources results in QoS violations for these applications even at low loads. However, high usage of a resource does not always correlate with sensitivity to interference in the same resource. For example, Memcached is highly sensitive to memory bandwidth interference despite not using a lot of bandwidth itself (Table 2). Its sensitivity is instead caused by its very stringent QoS target: since Memcached requests have to finish in a few hundred microseconds, they cannot tolerate high memory access latencies.

For all applications except for MongoDB, time sharing the same hyperthread incurs unsustainably high latencies. Even after eliminating the overhead of context switching, sharing a physical core even on different hyperthreads results in significantly lower throughput. Since colocation on the same physical cores also leads to contention in L1 and L2 caches, which cannot be mitigated by either software or hardware isolation mechanisms, we disallow sharing of a physical core.

3.4 Isolation Study
The study above shows that 1) for each of the studied applications, there are resources that, when contented for, lead to QoS violations; 2) for each shared resource, there are applications that suffer from its interference. To eliminate destructive interference, modern platforms have incorporated software and hardware isolation mechanisms (Table 3). We use these isolations mechanisms to understand the sensitivity to resource allocations, and the trade-offs between allocations of different resources.

We first run each application alone, using isolation mechanisms to cap the amount of allocated resources. This helps disentangle sensitivity to resource allocation from sensitivity to resource contention. We then colocate each application with contentious microbenchmarks, and study to what extent isolation mechanisms eliminate interference. We first study compute-related resources, including cores, power, and LLC, and then focus on storage-related resources, including memory and disk. We do not study network bandwidth in depth since, unlike other resources, it acts as a threshold, i.e., QoS can only be met when network bandwidth is sufficient, and does not improve thereafter.

3.4.1 Core, Power, and LLC Isolation
Figure 2a and 2b show sensitivity to compute-related resources when applications are at 30% and 90% of their respective max load. All applications except for MongoDB are most sensitive to core allocations, violating QoS when cores are insufficient. When cores are sufficient, both frequency and cache ways can be reduced while still meeting QoS. MongoDB is dominated by I/O traffic, and hence only requires a single core at the lowest frequency to meet QoS at high load.

Most applications are not highly sensitive to LLC allocations especially at low load. This is because cloud services have large datasets that do not fit in the LLC to begin with. However, for some applications like xapian, cache demand still increases at high load because of data reuse among concurrent requests [36]. Additionally, LLC isolation serves as an
indirect way to reduce memory bandwidth contention, which applications are more sensitive to, as shown in Table 4, and for which there is no direct isolation mechanism. To show this effect, we repeat the same experiment but colocating each application with microbenchmarks that thrash memory bandwidth on idle cores. As seen in Figure 2c, the LLC demand increases substantially compared to the stand-alone applications of Figure 2a, making partitioning critical. Furthermore, the demand for cores and frequency also increases, because faster computation is needed to hide the high memory access latency caused by increased cache misses.

3.4.2 Memory and Disk Isolation

Most studied applications do not involve disk operations, and increasing their memory capacity allocation beyond the size of their respective datasets does not improve performance. However, applications like MongoDB are I/O-intensive, and use memory as a software cache to relieve traffic to persistent storage. As shown in Figure 3, as memory capacity increases, MongoDB achieves the same latency at lower disk bandwidth, as more requests hit in memory.

3.4.3 Resource fungibility

Key to the effectiveness of PARTIES is the observation that resources are fungible, i.e., they can be traded for each other. In Section 4, we show that this reduces the time PARTIES needs to find an allocation that satisfies QoS for two reasons: (1) for a given application and load, there is more flexibility in the resources that can be used to meet QoS for all co-scheduled applications; and (2) the heuristic that explores the space of possible allocations can be kept relatively simple, as it is sufficient to find one satisfactory resource allocation. Indeed, Figures 2a-2c, and 3 all clearly show that, for any application at any given load, there are multiple feasible resource tuples. For instance, when Moses is at high load, <10 cores, 11 cache ways, turbo frequency>, <14 cores, 11 cache ways, 1.8GHz>, and <14 cores, 2 cache ways, 2.2GHz> are all valid allocations that meet QoS.

4 PARTIES Design

PARTIES is a feedback-based controller that dynamically adjusts resource allocations between co-scheduled LC applications using fine-grained monitoring and resource partitioning, with the objective to meet all applications’ QoS constraints. Below, we describe PARTIES in detail.

4.1 Design Principles

PARTIES is designed following four design principles:

- **Resource allocation decisions are dynamic and fine-grained.** As shown in Section 3, LC applications are very sensitive to resource allocations, with suboptimal decisions – even by a small amount – leading to QoS violations. Fine-grained monitoring detects such short resource demand bursts, and prevents them from occurring.

- **No a priori application knowledge and/or profiling is required.** Creating an offline profile of resource interactions in all possible application colocations, even if feasible, would be prohibitively expensive. Moreover, obtaining this information is not always possible, especially in the context of a public cloud hosting previously-unknown workloads. Instead, resource fungibility (Section 3.4.3) enables PARTIES to find viable allocations entirely online and in a timely fashion, and without relying on per-application empirically-tuned parameters.

- **The controller recovers from incorrect decisions fast.** Since PARTIES explores the allocation space online, inevitably some of its decisions may be counterproductive. By leveraging fine-grained online monitoring, PARTIES quickly detects and recovers from such events.

- **Migration is only used as a last resort.** When the aggregate resource demand of co-scheduled applications exceeds the server’s total capacity, meeting QoS for all services becomes impossible. In such cases, workload migration is the only remedy left. Because of the high overhead of migration, if it becomes necessary, PARTIES selects the application whose performance will be impacted the least from migration, either because it is stateless, or because it has a very relaxed QoS target (see Section 4.2.1).

4.2 PARTIES Controller

PARTIES consists of a monitoring and a resource allocation component. The former monitors per-application tail latency, memory capacity and network bandwidth usage, while the latter uses them to determine appropriate resource allocations, and enforces them using isolation mechanisms.

4.2.1 Main controller operation

As shown in Algorithm 1, the controller starts from fair allocation, where each application receives an equal partition of all managed resources, and all processors run at nominal frequency. After initialization, tail latencies and resource utilizations are sampled every 500ms, and based on the measurements, resources may be adjusted, depending on each application’s tail latency slack:

- If at least one application has little or negative slack, i.e., QoS is (about to be) violated, PARTIES will assign more resources to it, starting with application S with the smallest slack. This operation is carried out by upsize(S).

- When all applications comfortably satisfy their target QoS, PARTIES will reduce the resource allocation of application
When choosing which application to migrate, as per our design principles, PARTIES chooses the one with the most relaxed QoS target. When migration is invoked, the process involves: 1) choosing an application to migrate; 2) creating a new instance of the application on a more lightly-loaded machine; 3) redirecting requests from the previous instance to the new one; 4) terminating the previous instance. When choosing which application to migrate, as per our design principles, PARTIES chooses the one that will incur the least migration overhead. Compared to stateful applications, stateless services do not require migration of data in memory, thus introduce lower migration overheads. When all colocated jobs are stateful, an adjustment is acceptable as long as QoS is still satisfied post-adjustment. If the adjustment is not beneficial, the controller switches to a different resource in the next interval. Moreover, if in downsize, the action is immediately reverted to quickly recover from the previously incorrect decision. To prevent unnecessary QoS violations due to aggressive downsizing in the future, downsizing of this application is disabled for 30 seconds. The action is not reverted in downsize in case the application lacks multiple resources (e.g., it needs

---

**Algorithm 1:** PARTIES’ main function.

```
// Start from fair allocation of all resources initialization();
while TRUE do
    monitor tail latency and resource utilization for 500ms;
    adjust_network_bandwidth_partition();
    for each application A do
        slack[A] ← (target[A] - latency[A]) / target[A];
    end
    find application L with the largest slack;
    find application S with the smallest slack;
    if slack[S] < 0.05 then
        upsize(S);
    else
        if slack[L] > 0.2 then
            downsize(L);
        end
    end
    if cannot meet all applications’ QoS targets for one minute
        migrate();
    end
end
```

**Algorithm 2:** PARTIES’ upsize(A) function increases the resource allocation of application A to resolve its QoS violation. action is a global variable that corresponds to a pair of <direction, resource> that reflects which resource will be adjusted and how, for each application.

```
// Choose a resource to adjust
if action[A].direction ≠ UP then
    action[A] ← next_action(action[A], UP)
end

// Choose a resource to adjust
if previous_latency[A] < latency[A] then
    monitor tail latency and resource utilization for 500ms;
    if latency[A] > previous_latency[A] then
        action[A] ← next_action(action[A], UP);
    end
end
```

**Algorithm 3:** PARTIES’ downsize(A) function reclaims excess resources from app A to improve total utility.

```
// Choose a resource to adjust
if action[A].direction ≠ DOWN then
    action[A] ← next_action(action[A], DOWN)
end

// Choose a resource to adjust
if slack[A] < 0.05 then
    revert_back(A);
    action[A] ← next_action(action[A], DOWN);
end
```

When migration is invoked, the process involves: 1) choosing an application to migrate; 2) creating a new instance of the application on a more lightly-loaded machine; 3) redirecting requests from the previous instance to the new one; 4) terminating the previous instance. When choosing which application to migrate, as per our design principles, PARTIES chooses the one that will incur the least migration overhead. Compared to stateful applications, stateless services do not require migration of data in memory, thus introduce lower migration overheads. When all colocated jobs are stateful, PARTIES chooses the one with the most relaxed QoS target.

Network bandwidth allocations are adjusted every 500ms via `adjust_network_bandwidth_partition()`. As mentioned in Section 3.4, network bandwidth acts as a threshold as opposed to a tradeable resource, thus the controller allocates bandwidth based on each application’s usage. If bandwidth slack is less than 0.5Gbps, the bandwidth partition is increased by 0.5Gbps at a time. If slack is larger than 1Gbps, unused bandwidth is reclaimed in steps of 0.5Gbps.

### 4.2.2 Upsizing and Downsizing Allocations

The upsize and downsize functions (Algorithm 2 and 3) shift resources to or from an application. To do so, they first select a resource to adjust, and then evaluate the impact of the adjustment by monitoring latency and utilization. In upsize, an adjustment is beneficial if latency decreases, while in downsize, an adjustment is acceptable as long as QoS is still satisfied post-adjustment. If the adjustment is not beneficial, the controller switches to a different resource in the next interval. Moreover, if in downsize, the action is immediately reverted to quickly recover from the previously incorrect decision. To prevent unnecessary QoS violations due to aggressive downsizing in the future, downsizing of this application is disabled for 30 seconds. The action is not reverted in upsize in case the application lacks multiple resources (e.g., it needs...
more memory capacity and more cores). This may result in temporary oversubscription, however, excess resources will be reclaimed later once QoS is comfortably met.

### 4.2.3 Resource Ordering

The most important step in upsize and downsize is deciding which resource to adjust (function next_action). We represent allocation decisions as <direction, resource> pairs. There are ten actions in total: <UP/DOWN, CORE/CACHE/FREQ/MEM/DISK>, which correspond to increasing or decreasing cores, cache space, frequency, memory capacity, and disk bandwidth. Because PARTIES does not assume any a priori knowledge about each application’s characteristics, it picks the initial resource to adjust randomly. This ensures that the controller is generally applicable to any LC application regardless of which resources it is sensitive to. Since decisions happen at a sub-second granularity, even if the controller does not select the most critical resource first, it will at worst select it within the next four decision intervals.

PARTIES remembers the most recent action for each application. Figure 4 illustrates the detailed transitions between actions. For instance, if application A needs to be upped, action[A] will land in a random state in one of the UP wheels. Assume it lands in memory capacity in the storage wheel (i.e., action[A] is assigned to <UP, MEM>). If adjusting memory capacity does not improve latency, action[A] moves to the next resource (i.e., disk bandwidth) in the same wheel. If A is indeed in need of memory capacity and/or disk bandwidth, its latency should drop. If not, the controller will start adjusting compute resources by randomly selecting a resource in the compute wheel. Unlike resources in the storage wheel where the benefit in performance is almost always immediate, adjusting compute resources may require multiple rounds before there are noticeable performance gains. Indeed, when an application is severely starved for compute resources, fine-grained adjustments, e.g., in frequency, are not enough to dissipate the long queues that have built up in the system. Every time the controller completes a turn in the compute wheel, it checks memory utilization before deciding whether to initiate another round or to jump to the storage wheel. If memory slack is large and latency does not drop after scaling compute up, there is a high probability that the allocated compute resources are not yet sufficient. On the other hand, if memory is almost saturated, the QoS violation is likely due to an increasing dataset, in which case the controller jumps to the storage wheel.

**Skipping states:** There are a few corner cases that require states of a wheel to be skipped. First, when an application already has the max/min amount of a resource $K$, and the next action requires upsize/downsize that resource, next_action is called again to select a different resource. Second, for in-memory applications like Memcached, which will exhibit out-of-memory errors when memory is insufficient, the <DOWN, Mem Slack < 1GB> state in the storage wheel is skipped if memory capacity slack is less than 1GB. These services are easily identifiable by monitoring their disk bandwidth usage.

### 4.2.4 Enforcing Resource Allocations

PARTIES uses interfaces provided by the OS and the hardware platform (Table 3) to enforce resource isolation. Algorithm 4 shows how resources are adjusted. When attempting to upsize application $A$, the find_victim_application() function looks for an application to reclaim resources from. If BE jobs are present, PARTIES always reclaim resources from them first. Otherwise, the victim is usually the LC application with the highest tail latency slack. The only exception is when action[A] is <UP, MEMORY>, to avoid applications being killed by the OS due to out of memory (OOM) errors, find_victim_application() returns the application with the greatest memory capacity slack. It only returns the LC application with the greatest tail latency slack when no service has memory slack larger than 1GB. On the other hand, when attempting to downsize LC application $A$, if BE jobs are present, the controller yields the reclaimed resources to them. Otherwise, the reclaimed resources remain idle. Each interval adjusts resources at a fine granularity (one physical core, one cache way, 100MHz frequency, 1GB memory, or 1GB’s disk bandwidth), to minimize the impact on the victim application in upsize(), or on the application $A$ itself in downsize(). Finally, in upsize(), if both the upsized and victim application are in the same resource of an UP wheel, the victim will move to the next resource in the wheel to break the resource ping-ponging between the two applications.

### 4.3 Discussion

**What does PARTIES need to know about applications?**

PARTIES does not need any offline profiling, or a priori application knowledge except for their QoS targets. However, to reduce out-of-memory errors for in-memory applications during resource adjustments, a short online profiling is in need to classify if an application is in-memory. To do so, PARTIES monitors each application’s disk bandwidth usage for one second at the start of each application. An application is classified as in-memory when it does not involve I/O at all.
Algorithm 4: PARTIES’ take_action(A) function.

```java
if action[A].direction == UP then
    // find a BE or LC application to reclaim resources from
    V = find_victim_application();
    move resources from V to A;
    if V is latency critical and action[V] == action[A] then
        // avoid moving the same resource back and forth
        between two applications
        action[V] = next_action(action[V], UP);
    end
else
    // find an LC application to give reclaimed resources to
    V = find_recipient_application();
    move resources from A to V;
end
```

How is latency monitored? We monitor the latency of all requests on the client, via each service’s workload generator. Since we instantiate enough clients to avoid client-side saturation, the end-to-end latencies mostly reflect server-side delays. In a private cloud, internal applications are already instrumented to report their performance, therefore the cloud provider has access to all necessary performance metrics. In a public cloud, the applications either report their own performance, or allow the cloud provider to insert probe points to measure it. In a distributed deployment, a per-node local PARTIES agent will interact with a cluster scheduler which records end-to-end latencies that account for request fanout, and reports per-server QoS targets. We also examined monitoring low-level performance metrics (e.g., CPI [61]). Although they can distinguish nominal from heavily-problematic behavior, they are less effective when requiring fine-grained decisions, e.g., capturing relative performance slack across co-scheduled applications.

How are the controller parameters determined? The controller uses multiple threshold and step constants:

- The decision interval is set to 500ms by default. Although more frequent monitoring enables faster detection of QoS violations, overly fine-grained latency polling leads to noisy and unstable results, as there are not enough requests accumulated for tail latency to converge. Longer intervals provide better stability, but delay convergence.

- The latency slack for upsizing an allocation is set to 5% by default. Larger values make the controller more proactive at detecting potential QoS violations, however, they are also prone to raising false alarms which hurt resource efficiency. The slack for downsizing an allocation is set to 20% by default. Smaller values can result in overly aggressive resource reclaimations which hurt performance, while larger values lead to poor utilization. The two thresholds are configured based on a sensitivity study on a subset of the examined applications, and their effectiveness is validated with the remaining LC services.

- The timer that triggers migration is set to 1min based on PARTIES’s worst-case convergence time (see Section 5.4 for details on convergence). Shortening it would cause unnecessary migrations, while lengthening it would allow long-standing QoS violations.

- Finally, the granularity of resources adjusted per interval is set to 1 core, 1 cache way, 100MHz frequency, 1GB of memory, and 1Gbps disk bandwidth. Coarser granularity can lead to overly aggressive resource reclamation and QoS violations, while finer granularity prolongs convergence.

What if upsizing one application violates the QoS of another? Since PARTIES orders applications by increasing latency slack, if upsizing one application causes a QoS violation for the victim, the latter will also be upsized when it has the smallest latency slack. Moreover, PARTIES reduces resource ping-ponging by ensuring that when the victim application is upsized, it will not start from the same resource as the application it yielded resources to (see take_action function in Algorithm 4). Resource fungibility allows PARTIES to arrive to an acceptable resource allocation, as different resource vectors have an equivalent effect on performance.

Will an application keep getting QoS violations because of unsuccessful downsizing? This could happen when removing any resource brings latency slack from 20%+ to 5%. In practice, this happens rarely as large slack usually signals excessive allocated resources. However, to prevent this pathological case, downsizing an application is disabled for 30s once an incorrect `downsize` action was reverted.

How frequent is migration? Migration happens only when the migration timer expires. In practice, migrations are rare and only occur when the server is oversubscribed, i.e., the aggregate load exceeds the machine’s capacity.

How are job schedulers influenced in the presence of PARTIES? PARTIES is a per-node resource manager that runs locally and manages co-scheduled applications placed by the cluster-level scheduler. The scheduler periodically interacts with PARTIES to ensure that individual machines are neither overloaded nor oversubscribed.

5 Evaluation

5.1 Methodology

We evaluate PARTIES on a high-end server; details on our platforms and LC applications can be found in Section 3.1. Since PARTIES is an intra-node manager, it can simply be replicated across multiple machines. Only in the case of migrations a central coordinator with global cluster visibility is required to determine the destination machine.

In addition to LC applications, we create a multi-threaded BE job running in a separate container. The BE application consists of 14 threads of compute-intensive, and 14 threads of memory-thrashing microbenchmarks. Its throughput is defined as the aggregate throughput across microbenchmarks.

We first evaluate scenarios where applications run at constant loads, and later explore diurnal load patterns. We inject
applications with loads from 10% to 100% of their respective max load (Section 3.1), in 10% load increments. We test all load combinations for a given N-app mix, for a total of 10^N combinations. For each run, we allow 30s of warm-up and 60s of measurement, repeated 3 times. This is long enough for PARTIES to converge in all cases when the machine is not oversubscribed. If a satisfactory allocation in which all apps meet their QoS cannot be found after 1min, we signal that PARTIES is unable to deliver QoS for that configuration.

5.2 Constant Load

5.2.1 PARTIES Effectiveness

Figure 5 shows colocations of 2 LC-application mixes under PARTIES. In general, an application can operate at high load without violating QoS whenever the colocated application runs at a modest fraction of its own max load (typically 40-60%). MongoDB is a particularly amenable co-runner due to its low compute demands, with both MongoDB and its colocated application successfully running close to their respective max load. The only exception is when both applications are MongoDB instances, in which case the aggregate throughput cannot exceed 160% because of I/O contention.

PARTIES is designed to support any number of LC applications. To illustrate this, we also show results for three- and six-application mixes in Figure 6. To conserve space, we show only the most challenging of the 3-application mixes, namely those with Memcached and Xapian, which have the strictest QoS of all studied applications (Table 2). PARTIES again meets QoS for all co-scheduled applications, up until the point where the machine becomes oversubscribed. As before, MongoDB’s I/O-bound behavior enables more resources to be yielded to the other services.

5.2.2 Comparison with Heracles

Heracles [43] is the most relevant prior work on resource allocation for LC applications. Unlike PARTIES, Heracles is designed for a single LC job running with one or more low-priority BE jobs. Thus, when evaluating Heracles with multiple LC services, we select the one with the strictest QoS as the LC application, and treat the others as BE jobs. Note that in Heracles, there is no partitioning between BE jobs.

We compare PARTIES and Heracles using Effective Machine Utilization (EMU), a metric used in the Heracles evaluation [43], defined as the max aggregate load of all colocated applications, where each application’s load is expressed as a percentage of its max load, as before. Note that EMU can be above 100% due to better bin-packing of shared resources. Figure 7 shows the EMU achieved by Heracles for 2- up to 6-app mixes. PARTIES achieves 13% higher EMU for 2-app mixes on average. This difference increases with the number of co-scheduled applications. For 6-app mixes, PARTIES achieves on average 61% higher EMU than Heracles.

There are several factors that justify these results:

- Heracles suspends BE jobs upon detecting a QoS violation, which is counterproductive when the colocated jobs are also latency-critical. PARTIES instead adjusts the partitioning of multiple resources to find a configuration that meets the QoS of all co-scheduled LC applications.
- There is no resource partitioning between BE jobs in Heracles, which is problematic when there are 3 or more colocated LC applications. This leads to lower EMU for Heracles, with the gap between Heracles and PARTIES increasing with the number of colocated services.
- Heracles uses several resource subcontrollers that operate independently from each other. For example, Heracles may adjust frequency and cores at the same time, which may be too aggressive in downsize and too conservative in upsize. It also does not leverage the fact that these two resources are tradeable with each other. Instead in PARTIES, only one resource is adjusted in each interval.
- Heracles does not support partitioning of memory capacity or disk bandwidth. This particularly shows up when multiple I/O-bound workloads, e.g., 2 instances of MongoDB, are colocated on the same physical host.

5.2.3 Comparison with Other Resource Controllers

Next, we examine the most challenging three-app mix, Memcached, Xapian, and NGINX, which have the strictest QoS requirements of all studied applications. In addition to Heracles, we also compare with two other controllers:

- Unmanaged: No isolation mechanisms are used, and the ACPI frequency driver is set to the default “ondemand” governor. The unmanaged environment relies on the OS to schedule applications and manage resources.
- Oracle: An ideal manager that always finds a viable allocation, if one exists, via exhaustive offline profiling.
Figure 6. Colocation of 3- and 6-app mixes. The heatmap values are the max percentage of the third app’s (or NGINX in the 6-app mix) max load that can be achieved without QoS violations when Memcached and Xapian run at the fraction of their max loads indicated in the x and y axes, respectively. In the 6-app mix, Moses, Sphinx, and MongoDB are at 10%, 10% and 100% of their respective max load (not shown).

Figure 7. Violin plots of Effective Machine Utilization (EMU) with constant load for 2- to 6-app mixes. Red markers show min, mean, and max EMU.

Figure 8. Colocation of Memcached (M), Xapian (X) and NGINX (N) with different resource managers. The values in the heatmaps are the max percentage of N’s max load achieved without QoS violations when M and X run at the fraction of their max loads indicated in the y and x axes.

5.3 Fluctuating Load
We now evaluate how PARTIES behaves with dynamically changing load. Datacenter applications often experience fluctuations in their load, such as diurnal patterns where load is high at daytime, and gradually decreases during the night. To simulate this scenario, we choose a three-application mix with Memcached, Xapian, and Moses. This mix includes the two applications with the strictest QoS, plus the one with the most pressure on memory bandwidth (Moses). We vary the load of Memcached from 10% to 60%, and set the load of Moses and Xapian at 10 and 20% of their respective max load.

Figure 9 shows how PARTIES dynamically tunes resources to adjust to Memcached’s load variation. Since adjusting network bandwidth is trivial, and these applications do not contend for memory capacity and disk bandwidth, we only show compute-related resources in the plot.

In the beginning, all three services are lightly loaded. PARTIES starts with a fair allocation of all resources. As the system is lightly loaded, PARTIES detects a large slack in the tail latency of all services (Memcached first, then Moses and Xapian), and decreases their core and cache allocations. The BE job therefore gets more resources, and higher throughput.

At 25s, the load of Memcached quickly ramps up from 10% to 60% of its max load. Unmanaged and Heracles start faltering when Memcached’s load increases to 40% at around 60s. Unmanaged results in a dramatic increase in latency, whereas Heracles detects the QoS violation, and pauses all other applications for five minutes, as specified in [43]. As a result,
Moses and Xapian experience rapidly increasing latencies, and eventually drop requests. PARTIES detects latency spikes as load increases, and gradually moves more resources to Memcached. At the same time, the BE throughput drops due to fewer available resources. When Memcached is at 60% of its max load, both Memcached and Xapian start experiencing QoS violations. PARTIES upsizes their resources accordingly. As a result, it finds a valid allocation within 20s, preventing any further QoS violations.

At around 120s, even though Memcached’s load starts to decrease, resources are not reclaimed immediately, as latency slack is still small. To prevent a potential latency surge, PARTIES downsizes an application only when its latency slack is larger than 0.2; this happens at 135s. Subsequently, the BE throughput increases, as the aggregate LC load decreases. Even though there are still occasional short QoS violations during this period, latencies recover quickly, as incorrect downsize actions are immediately reverted.

5.4 PARTIES Overhead

PARTIES is currently implemented as a user-level runtime that polls the latency and resource utilization of applications, and interacts with the OS and hardware to adjust allocations. The runtime is pinned on core 0, taking 15% of its CPU utilization (monitoring and resource adjustment each take 10% and 5%, respectively).

Figure 10 shows convergence time for 2- to 6-LC app mixes with constant loads. PARTIES takes a few seconds (when the initial partition works), up to sixty seconds (worst case of all six LC applications colocated) to converge to an allocation without QoS violations. In general, convergence time depends on the load of each application, and the number of colocated applications. Note that, although the total search space grows exponentially with the number of colocated interactive applications, convergence time in practice grows much more slowly: when moving from 2- to 6-LC application mixes, average convergence time increases by 2.8x even though the search space increases by several orders of magnitude. This is because PARTIES does not attempt to find the optimal resource allocation: rather, it stops the exploration the moment all applications meet their QoS, which greatly reduces the exploration time. PARTIES then relies on downsize() (Algorithm 3) to further close the gap between the selected and optimal allocations.

6 Conclusion

We have presented PARTIES, an online resource controller that enables multiple latency-critical applications to share a physical host without QoS violations. PARTIES leverages both hardware and software isolation mechanisms to preserve QoS, and assumes no a priori information about any of the co-scheduled services. We have evaluated PARTIES against state-of-the-art mechanisms, and showed that it achieves considerably higher throughput, while satisfying QoS in the face of varying loads, and that its gains increase with the number of co-scheduled applications.

Acknowledgments

We thank Daniel Sanchez, David Lo, and the anonymous reviewers for their feedback. This work was supported in part by Air Force award FA9550-5-1-0311; by NSF and the Semiconductor Research Corporation (SRC) through the DEEP3M Center, part of the E2CDA program; and by DARPA and SRC through the CRISP Center, part of the JUMP program. Christina Delimitrou was supported by NSF award CNS-1422088, by a Facebook Faculty Award, and by a John and Norma Balen Sesquicentennial Faculty Fellowship.
References


