Caladan: Mitigating Interference at Microsecond Timescales

Joshua Fried and Zhenyuan Ruan, *MIT CSAIL;* Amy Ousterhout, *UC Berkeley;* Adam Belay, *MIT CSAIL*

https://www.usenix.org/conference/osdi20/presentation/fried

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

Caladan: Mitigating Interference at Microsecond Timescales

Joshua Fried, Zhenyuan Ruan, Amy Ousterhout†, Adam Belay
MIT CSAIL, †UC Berkeley

Abstract

The conventional wisdom is that CPU resources such as cores, caches, and memory bandwidth must be partitioned to achieve performance isolation between tasks. Both the widespread availability of cache partitioning in modern CPUs and the recommended practice of pinning latency-sensitive applications to dedicated cores attest to this belief.

In this paper, we show that resource partitioning is neither necessary nor sufficient. Many applications experience bursty request patterns or phased behavior, drastically changing the amount and type of resources they need. Unfortunately, partitioning-based systems fail to react quickly enough to keep up with these changes, resulting in extreme spikes in latency and lost opportunities to increase CPU utilization.

Caladan is a new CPU scheduler that can achieve significantly better quality of service (tail latency, throughput, etc.) through a collection of control signals and policies that rely on fast core allocation instead of resource partitioning. Caladan consists of a centralized scheduler core that actively manages resource contention in the memory hierarchy and between hyperthreads, and a kernel module that bypasses the standard Linux Kernel scheduler to support microsecond-scale monitoring and placement of tasks. When colocating memcached with a best-effort, garbage-collected workload, Caladan outperforms Parties, a state-of-the-art resource partitioning system, by 11,000×, reducing tail latency from 580 ms to 52 μs during shifts in resource usage while maintaining high CPU utilization.

1 Introduction

Interactive, data-intensive web services like web search, social networking, and online retail commonly distribute requests across thousands of servers. Minimizing tail latency is critical for these services because end-to-end response times are determined by the slowest individual response [4, 14]. Efforts to reduce tail latency, however, must be carefully balanced with the need to maximize datacenter efficiency; large-scale datacenter operators often pack several tasks together on the same machine to improve CPU utilization in the presence of variable load [22, 57, 66, 71]. Under these conditions, tasks must compete over shared resources such as cores, memory bandwidth, caches, and execution units. When shared resource contention is high, latency increases significantly; this slowdown of tasks due to resource contention is called *interference*.

The need to manage interference has led to the development of several hardware mechanisms that *partition* resources. For example, Intel's Cache Allocation Technology (CAT) uses way-based cache partitioning to reserve portions of the last level cache (LLC) for specific cores [21]. Many systems use these partitioning mechanisms to improve performance isolation [8, 12, 28, 38, 62, 73]. They either statically assign enough resources for peak load, leaving significant CPU utilization on the table, or else make dynamic adjustments over hundreds of milliseconds to seconds. Because each adjustment is incremental, converging to the right configuration after a change in resource usage can take dozens of seconds [8, 12, 38].

Unfortunately, real-world workloads experience changes in resource usage over much shorter timescales. For example, network traffic was observed to be very bursty in Google's datacenters, sometimes consuming more than a dozen cores over short time periods [42], and a study of Microsoft's Bing reports highly bursty thread wakeups on the order of microseconds [27]. Phased resource usage is also common. For example, we found that tasks that rely on garbage collection (GC) periodically consume all available memory bandwidth (§2). Detecting and reacting to such sudden changes in resource usage is not possible with existing systems.

Our goal is to maintain both high CPU utilization and strict performance isolation (for throughput and tail latency) under realistic conditions in which resource usage, and therefore interference, changes frequently. A key requirement is faster reaction times, as even microsecond delays can impact latency after an abrupt increase in interference (§2). There are two challenges toward achieving microsecond reaction times. First, there are many types of interference in a shared CPU (hyperthreading, memory bandwidth, LLC, etc.), and obtaining the right control signals that can accurately detect each of them over microsecond timescales is difficult. Second, existing systems face too much software overhead to either gather control signals or adjust resource allocations quickly.

To overcome these challenges, we present an interference-aware CPU scheduler, called **Caladan**. Caladan consists of a centralized, dedicated scheduler core that collects control signals and makes resource allocation decisions, and a Linux Kernel module, called KSCHED, that efficiently adjusts resource allocations. Our scheduler core distinguishes between high-priority, latency-critical (LC) tasks and low-priority, best-effort (BE) tasks. To avoid the reaction time limitations imposed by hardware partitioning (§3), Caladan relies exclusively on core allocation to manage interference.

Caladan uses a carefully selected set of control signals and corresponding actions to quickly and accurately detect and respond to interference over microsecond timescales. We

observe that interference has two interrelated effects: first, interference slows down the execution speed of cores (more cache misses, higher memory latency, etc.), impacting the *service times* of requests; second, as cores slow down, *compute capacity* drops; when it falls below offered load, queueing delays increase dramatically.

Caladan's scheduler targets these effects. It collects fine-grained measurements of memory bandwidth usage and request processing times, using these to detect memory bandwidth and hyperthreading interference, respectively. It then restricts cores from the antagonizing BE task(s), eliminating most of the impact on service times. For LLC interference, Caladan cannot eliminate service time overheads directly, but it can still prevent a decrease in compute capacity by allowing LC tasks to steal extra cores from BE tasks.

The KSCHED kernel module accelerates scheduling operations such as waking tasks and collecting interference metrics. It does so by amortizing the cost of sending interrupts, offloading scheduling work from the scheduler core to the tasks' cores, and providing a non-blocking API that allows the scheduler core to handle many inflight operations at once. These techniques eliminate scheduling bottlenecks, allowing Caladan to react quickly while scaling to many cores and tasks, even under heavy interference.

To the best of our knowledge, Caladan is the first system that can maintain both strict performance isolation and high CPU utilization under frequently changing interference and load. To achieve these benefits, Caladan imposes two new requirements on applications: the adoption of a custom runtime system for scheduling and the need for LC tasks to expose their internal concurrency (§8). In exchange, Caladan is able to converge to the right resource configuration 500,000× faster than the typical speed reported for Parties, a state-of-the-art resource partitioning system [12]. We show that this speedup yields an 11,000× reduction in tail latency when colocating memcached with a BE task that relies on garbage collection. Moreover, we show that Caladan is highly general, scaling to multiple tasks and maintaining the same benefits while colocating a diverse set of workloads (memcached, an in-memory database, a flash storage service, an x264 video encoder, a garbage collector, etc.). Caladan is available at https://github.com/shenango/caladan.

2 Motivation

In this section, we demonstrate how performance can degrade when interference is not quickly mitigated. Many workloads exhibit phased behavior, drastically changing the types and quantities of resources they use at sub-second timescales. Examples include compression, compilation, Spark compute jobs, and garbage collectors [49, 59]. The request rates issued to tasks can also change rapidly, with bursts occurring over microsecond timescales [5, 27, 42]; these bursts in load can cause bursts of resource usage. In both cases, abrupt changes

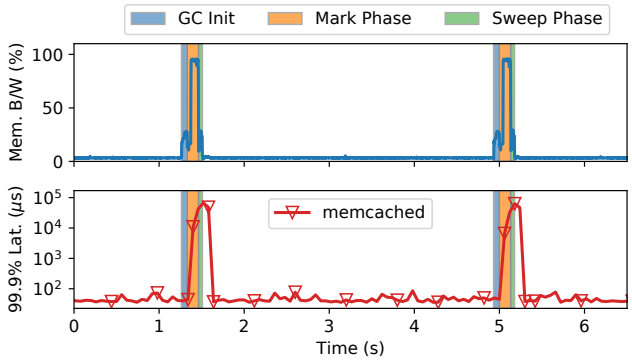


Figure 1: Periodic GC in a background task (shaded regions) increases memory bandwidth usage (top), causing severe latency spikes for a colocated memcached instance (bottom). Note the log-scaled y-axis in the bottom graph.

in resource usage can abruptly increase interference. This degrades request service times and causes request queues to grow when the rate of arriving requests exceeds the rate at which a task can process them.

To better understand the challenges associated with time-varying interference, we consider what happens when we colocate an LC task, memcached [43], with a BE workload that exhibits phased behavior due to garbage collection. In this example, we use the Boehm GC (see §7), which employs the mark-sweep algorithm to reclaim dead heap objects [10]. We have observed similar problems with more sophisticated, incremental GCs, such as the Go Language Runtime [60].

In this experiment, we offer a fixed load to memcached and statically partition cores between the two tasks. memcached is given enough cores to keep its 99.9$^{\text{th}}$ percentile tail latency below 50 μs when run in isolation. As shown in Figure 1, this allocation is sufficient to protect tail latency when the GC is not running but it fails when the GC starts. The GC pauses normal execution of the BE task for 100–200 ms and scans the entire heap using all cores available to the BE task, which saturates memory bandwidth. During this brief period, each memcached request experiences a higher rate of cache misses and larger memory access latencies, causing the rate at which memcached can service requests to drop by about half and queues to build up. As a result, memcached's queueing delay increases at a rate of 5 μs every 10 μs, eventually reaching a tail latency that is 1000× higher than normal.

This example illustrates that fixed core partitioning is insufficient, and also indicates what core reallocation speed is necessary in order to effectively mitigate interference. If changes in interference can instantaneously reduce the request service rate by half, then in order to keep latencies from increasing by X, the CPU scheduler must detect and respond to interference within $2X$. Thus, preventing a latency increase of 50 μs requires reaction times within 100 μs. Unfortunately, existing systems are not designed to respond this quickly (§3), forcing datacenter operators to either tolerate severe tail latency spikes, or else isolate these tasks on different servers.

System	Decision Interval	Typical Convergence	Requires CAT	Supports HT
Heracles [38]	2–15 s	30 s	✓	✗
Parties [12]	500 ms	10–20 s	✓	✗
Caladan	10–20 μs	20 μs	✗	✓

Table 1: A comparison of Caladan to state-of-the-art systems that use partitioning to manage interference. Caladan can converge to the right resource configuration 500,000× faster.

3 Background

Throughout this paper, we discuss three forms of interference that can occur when sharing a CPU: hyperthreading interference, memory bandwidth interference, and LLC interference. Hyperthreading interference is usually present at a baseline level whenever tasks are running on *sibling* cores because the CPU divides certain physical core resources (e.g., the micro-op queue), but it can become more severe depending on whether shared resources (L1/L2 caches, prefetchers, execution units, TLBs, etc.) are contended. Memory bandwidth and LLC interference, on the other hand, can vary in intensity, but impact all cores that share the same physical CPU. As memory bandwidth usage increases, memory access latency slowly increases due to interference, until memory bandwidth becomes saturated; access latency then increases exponentially [62]. LLC interference is determined by the amount of cache each application uses: when demand exceeds capacity, LLC miss rates increase.

In this section, we discuss why existing systems are unable to manage abrupt changes in interference (§3.1) and explore the limitations imposed on them by the hardware extensions available in commercial CPUs (§3.2).

3.1 Existing Approaches to Interference

State-of-the-art systems such as Heracles [38] and Parties [12] handle interference by dynamically partitioning resources, such as cores and LLC partition sizes. However, both Heracles and Parties make decisions and converge to new resource allocations too slowly to manage bursty interference (Table 1). There are two main reasons. First, both systems detect interference using application-level tail latency measurements, which must be measured over hundreds of milliseconds in order to obtain stable results; the Parties authors found that shorter intervals produced "noisy and unstable results" [12]. Second, both systems make incremental adjustments to resource allocations, gradually converging to a configuration that can meet latency objectives. These systems lack the ability to identify the source of interference (application and contended resource) directly, so convergence can involve significant trial-and-error as different resources are throttled, requiring seconds to converge to a new resource allocation. During the adjustment period, latency often continues to suffer because the LC task must wait to be given enough resources to reduce its queueing delay buildup.

Thus, both Heracles and Parties take at least 50× as long to adapt to changes in interference as the duration of a GC cycle in our example. As a result, operators must make tradeoffs based on tunable parameters: either tail latency tolerances (e.g., 99.9$^{\text{th}}$ percentile tail latency) can be set higher, causing the GC interference to be tolerated without resource reallocations, or they can be set lower, causing the GC workload to be throttled continuously. Because the GC workload causes minimal interference during the majority of its execution (while not collecting garbage), faster reaction times are needed to keep cores busy without compromising tail latency.

In addition to convergence speed, existing systems suffer from scalability limitations. For example, a typical datacenter server must handle several LC and BE tasks simultaneously [66, 71], but Heracles is limited to only a single LC task (and many BE tasks). Parties can support multiple LC and BE tasks, but because it can only guess at which task is causing interference, its convergence time increases with each additional task.

The hardware mechanisms on which these systems rely also impose limitations. For example, hyperthreads lack control over resource partitioning, so Heracles and Parties turn them off entirely. Using both hyperthreads on a core simultaneously can yield up to 30% higher throughput than using a single hyperthread [40, 44, 45, 54], so this lowers system throughput significantly. Furthermore, the available hardware partitioning mechanisms that can be controlled constrain both reaction speeds and scalability. We discuss this problem next.

3.2 Limitations of Hardware Extensions

Intel has added several extensions to its server CPUs that are designed to partition and monitor the LLC and memory bandwidth. These extensions are optimized for scenarios where resource demand changes slowly, but as shown in our study of the GC workload, this assumption does not always hold. To better understand these limitations, we discuss each component in more detail.

The most commonly used extension is CAT, a technology that divides portions of the LLC between tasks to increase performance determinism [21]. CAT's way-based hardware implementation suffers from two limitations. First, changes to the partition configuration can take considerable time to have an effect; Intel cautions that "a reduction in the performance of [CAT] may result if [tasks] are migrated frequently" [26, sec. 17.19.4.2]. Appropriately sizing a partition, however, is challenging under time-varying demand because it must be large enough to accommodate peak usage. Second, CAT must divide a finite number of set-associative ways between partitions, reducing associativity within each partition. Unfortunately, performance can degrade significantly as associativity decreases [56, 67]; KPart avoids this by grouping complementary tasks together in the same partition, but it relies on frequent online profiling to identify groupings, resulting in high tail latency [18].

Another extension called Memory Bandwidth Allocation (MBA) applies a per-core rate limiter to DRAM accesses to throttle bandwidth consumption. MBA is necessary for systems that statically assign cores because it is the only method they can use to limit bandwidth consumption. Unfortunately, it is at odds with our goal of achieving high CPU utilization: a core that is heavily rate-limited by MBA will spend the majority of its time stalling. Instead, we found it is more efficient to allocate fewer cores, achieving the same throughput for a task, but with higher per-core utilization.

Finally, configuring partitioning mechanisms effectively requires the attribution of resource usage to specific tasks. To help with this goal, Intel introduced Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM) [72]. Unfortunately, these mechanisms are unable to detect changes in system conditions quickly. For example, when monitoring a streaming task with CMT, it takes 112 ms for its cache occupancy measurement to stabilize [21]. Similarly, we discovered experimentally that MBM requires milliseconds to accurately estimate memory bandwidth usage.

4 Challenges and Approach

Our overarching goal is to maintain performance isolation while maximizing CPU utilization. Achieving this goal is difficult because managing changes in interference requires microsecond-scale reaction times. Partitioning resources in hardware is too slow for these timescales (§3.2), so Caladan's approach is to instead manage interference by controlling how cores are allocated to tasks. Prior systems have adjusted cores as part of their strategy for managing interference [12, 28, 38, 70], but Caladan is the first system to rely exclusively on core allocation to manage multiple forms of interference. To mitigate interference quickly enough, we had to overcome two key challenges:

1. **Sensitivity:** For fast and targeted reactions, Caladan requires control signals that can identify the presence of interference and its source—task and contended resource—within microseconds. Commonly used performance metrics like CPI [71] or tail latency [12, 38] (as well as hardware mechanisms like MBM and CMT) are too noisy to be useful over short timescales. Metrics like queueing delay [8, 42, 47, 68] can be measured over microsecond timescales, but cannot identify the source of interference, only that a task's performance is degrading.
2. **Scalability:** Existing systems depend heavily on the Linux Kernel in order to gather control signals and adjust resource allocations (e.g., using `sched_setaffinity()` to adjust core allocations) [8, 12, 20, 38, 47, 52, 68]. Unfortunately, Linux adds overhead to these operations, and these overheads increase in the presence of interference and as the number of cores and tasks increase.

We address the challenge of sensitivity by carefully selecting control signals that enable fast detection of interference and by dedicating a core to monitor these signals and take action to mitigate interference as it arises. We address the challenge of scalability with a Linux Kernel module named KSCHED. We describe these in more detail below.

Caladan's Actions to Mitigate Interference

	Impact of Interference	
Contended Resource	↑ Service Times	↓ Compute Capacity
Hyperthreads	idle sibling core	add victim cores
Memory Bandwidth	throttle antagonist	add victim cores
LLC	none	add victim cores

Table 2: When a resource (left) becomes contended, Caladan takes action to avoid increased service times (middle). When this is insufficient to maintain compute capacity, Caladan takes additional action (right).

4.1 Caladan's Approach

Caladan dedicates a single core, called the *scheduler*, to continuously poll and gather a set of *control signals* over microsecond timescales. The scheduler uses these signals to detect interference and then reacts by adjusting core allocations. The scheduler is designed to manage several forms of interference (§3), using control signals tailored to each. For hyperthreads, we assume interference is always present when both siblings are active (because some physical core resources are partitioned) and focus on reducing interference for the requests that will impact tail latency—that is, the longest running requests [70]. We measure request processing times to identify these requests. For memory bandwidth, we measure global memory bandwidth usage to detect DRAM saturation and measure per-core LLC miss rates to attribute usage to a specific task. For cases like the LLC where we cannot directly measure or infer interference, we can still measure a key side effect of interference: increased queueing delays, caused by reductions in compute capacity. By focusing on interference-driven control signals, Caladan can detect problems before quality of service is degraded.

Table 2 summarizes the actions Caladan takes to mitigate interference. We first try to prevent service time increases by reducing interference directly. For example, Caladan reduces hyperthreading interference by controlling which logical cores (hyperthreads) may be used, idling a logical core when its sibling exceeds a request processing time threshold. In addition, it reduces memory bandwidth interference by limiting how many cores each task may use; this is effective because reducing the number of cores allocated to a task reduces its memory bandwidth usage. However, reducing LLC interference is more difficult: the magnitude of LLC interference is determined primarily by how much LLC capacity a task uses, but reducing a task's number of cores reduces its LLC access rate rather than its LLC capacity. Therefore, Caladan compensates for LLC interference—and any remaining hyperthreading and memory bandwidth interference—by granting extra cores to victim tasks, allowing them to recoup

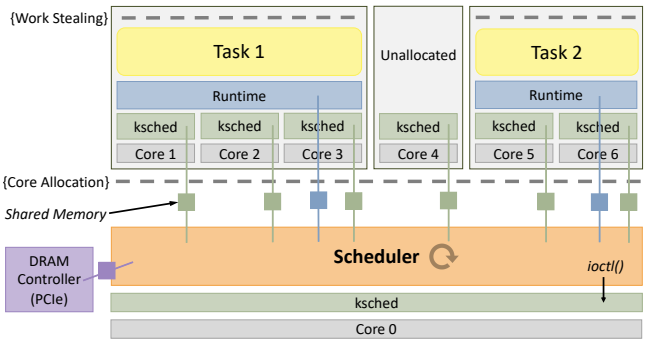


Figure 2: Caladan's system architecture. Caladan relies on a scheduler core to gather control signals from shared memory regions (provided by KSCHED, runtimes, and the DRAM controller). It uses these control signals to adjust core allocations via KSCHED.

the compute capacity lost to interference. Although this cannot fully protect service times, it can prevent queueing delays.

Finally, Caladan introduces a Linux Kernel module called KSCHED. KSCHED performs scheduling functions across many cores at once in a matter of microseconds, even in the presence of interference. KSCHED achieves these goals with three main techniques: (1) it runs on all cores managed by Caladan and shifts scheduling work away from the scheduler core to cores running tasks; (2) it leverages hardware support for multicast interprocessor interrupts (IPIs) to amortize the cost of initiating operations on many cores simultaneously; and (3) it provides a fully asynchronous scheduler interface so that the scheduler can initiate operations on remote cores and perform other work while waiting for them to complete.

5 Design

5.1 Overview

Figure 2 presents the key components of Caladan and the shared memory regions between them. Caladan shares some architectural and implementation building blocks with Shenango [47]: each application is linked with a runtime system, and a dedicated scheduler core (run with root privileges) busy polls shared memory regions to gather control signals and make core allocations. Both systems are designed to interoperate in a normal Linux environment, potentially managing a subset of available cores.

Despite these commonalities, Caladan adopts a radically different approach to scheduling and relies on different scheduling mechanisms. Shenango uses queueing delay as its only control signal to manage changes in load; Caladan uses multiple control signals to manage several types of interference as well as changes in load. Moreover, Shenango's scheduler core combines network processing with CPU scheduling; Caladan's scheduler core is only responsible for CPU scheduling, eliminating packet processing bottlenecks (§6). Finally, Shenango relies on standard Linux system calls to allocate cores, limiting its scalability; Caladan uses KSCHED to more efficiently perform its scheduling functions, including preempting tasks, assigning cores to tasks, detecting when tasks have yielded voluntarily, and reading performance counters from remote cores.

Caladan's runtimes share many properties with those of Shenango. Applications managed by Caladan run inside normal Linux processes, which we refer to as *tasks*. Within each task, the runtime provides "green" threads (light-weight, user-level threads) and kernel-bypass I/O (networking and storage). Runtimes use work stealing to balance load across the cores that are allocated to them—a best practice for minimizing tail latency [51]—and yield cores when they run out of work to steal. Handling threading and I/O in userspace makes managing interference easier in two ways. First, by performing all processing inside the task that needs it, we can better manage the resource contention it generates. By contrast, the Linux Kernel handles I/O on behalf of its tasks, making it difficult to attribute resource usage or interference to a specific task. Second, we can easily instrument the runtime system to export the right per-task control signals (discussed further in §5.2).

Provisioning cores: Users provision each task with a discrete number of *guaranteed cores* (zero or more) that are always available when needed. They can also allocate tasks additional *burstable cores* beyond the number guaranteed, allowing them to make use of any idle capacity. Additionally, each task is designated as LC or BE. BE tasks operate at a lower priority: they are only allocated burstable cores when LC tasks do not need them, they are always provisioned zero guaranteed cores, and they are throttled as needed to manage interference.

In some configurations, it may not be possible to manage interference without harming the performance of LC tasks. To prevent these cases, we recommend a configuration that leaves a small number of cores that are not guaranteed to any task, providing enough slack to manage interference. Caladan can also detect when provisioning constraints prevent it from mitigating interference. As a last resort, this information could be reported back to the cluster scheduler so that it could migrate tasks to other machines. A rich body of prior work has explored adding similar types of interference coordination, as well as identifying complementary workloads, at the cluster scheduler layer [12, 15, 16, 41, 69, 71].

5.2 The Caladan Scheduler

Figure 3 shows the scheduler's key components, the control signals they each use, and their interactions. Separate controller modules detect memory bandwidth and hyperthreading interference, each placing constraints on how cores can be allocated and revoking cores as necessary. The memory bandwidth controller restricts how many cores can be assigned to a task, while the hyperthread controller bans cores within sibling pairs. A top-level core allocator incorporates these restrictions and decides when to grant additional cores to tasks. It tries to minimize queueing delay (to manage changes in load and any unmitigated interference), allocating cores to tasks in a way that respects constraints from the controllers and each

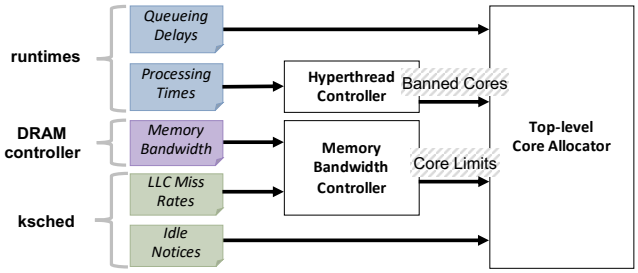


Figure 3: The flow of information through Caladan's scheduler. Control signals flow from runtimes, the DRAM controller, and KSCHED to the controllers and top-level allocator. The hyperthread and memory bandwidth controllers impose constraints on which and how many cores the top-level allocator may grant.

task's resource configuration (the number of guaranteed cores, BE vs. LC, etc.). The controllers and core allocator run once every 10 μs on the scheduler's dedicated core. Because the scheduler can reallocate cores so quickly, it is possible to allocate fractional cores to tasks on average over time (e.g., when less than a full core is needed to accommodate load). This is made efficient through KSCHED's scheduling optimizations (§5.3).

The scheduler gathers control signals from three sources. First, runtimes provide information about request processing times and about queueing delays. Second, the DRAM controller provides information about global memory bandwidth usage. Third, KSCHED provides information about per-core LLC miss rates and notifies the scheduler core when a task has yielded voluntarily. We now discuss each component in more detail. We present each algorithm as synchronous code for clarity, but to handle many tasks concurrently without delaying the scheduler, all code is asynchronous in practice. Each algorithm relies on one tunable parameter; these are described in more detail in Appendix A.

5.2.1 The Top-level Core Allocator

The goal of the top-level core allocator is to grant more cores to tasks that are experiencing queueing delays, whether these delays are due to lingering interference (as shown in the rightmost column of Table 2) or due to changes in load. Algorithm 1 shows its basic operation. The core allocator periodically checks the queueing delay of each task, and, when permitted by the memory bandwidth controller, tries to add cores to the tasks that have delays above a configurable per-task threshold (`THRESH_QD`). Queueing can occur in each runtime core's green thread runqueue, network ingress queue, storage completion queue, and timer heap. Each queued element contains a timestamp of its arrival time, and all queues are placed in shared memory. `QueueingDelay()` computes the delay for each core by summing the delays experienced by the oldest element in each of its queues. It then reports the maximum delay observed across the task's cores.

When a task's delay exceeds its `THRESH_QD`, the allocator

```
1  while True:
2      for each task T:
3          if QueueingDelay(T) < THRESH_QD[T]:
4              continue;
5          if T is limited by BW controller:
6              continue;
7          // try to allocate a core
8          for each core C:
9              if C is banned by HT controller:
10                 continue;
11             if task_on_core[C] has priority over T:
12                 continue;
13             score[C] = CalculateScore(C, T);
14         find core C with highest score;
15         allocate C to T (if found);
16     sleep(10 µs);
```

Algorithm 1: The top-level core allocator.

loops over all cores, checking which cores are allowed by the hyperthread controller and checking which tasks are running on each core. An idle core can be allocated to any task, but a busy core can only be preempted if the core provisioning configuration allows it. For example, if an LC task is only using guaranteed cores, it cannot be preempted by another task. Moreover, a BE task can never preempt an LC task.

Finally, `CalculateScore()` assigns a score to each core, and the core allocator picks the allowed core with the highest score (if one is found). Our scoring function is based on three factors (in order of priority). First, we prefer sibling pairs that are both idle because they have no hyperthreading interference. Second, we prefer hyperthread pairings between *different* tasks because hyperthreading is most efficient when tasks have different performance bottlenecks [31, 45]. Finally, we optimize for temporal locality: Caladan keeps track of the time each task last used each core, and gives the most recent timestamp the highest score. Timestamps are shared between hyperthread siblings, reflecting their shared cache resources.

The core allocator also receives notifications from KSCHED whenever a runtime yields a core voluntarily (not shown in Algorithm 1). When this happens, it updates the `task_on_core` array and immediately tries to grant the core to another task, reducing the cycles the core spends idling.

5.2.2 The Memory Bandwidth Controller

Algorithm 2 shows our memory bandwidth controller. Our aim is to use the majority of available memory bandwidth while avoiding saturation. The memory bandwidth controller periodically polls the DRAM controller's global memory bandwidth usage counter, calculating the access rate since the last polling interval, and triggers when it crosses a saturation threshold (`THRESH_BW`). It then attributes memory bandwidth usage to a specific task by relying on KSCHED to efficiently sample LLC misses from the performance monitoring unit (PMU) [25] of each scheduled core. We found that LLC misses are a good indicator of overall memory bandwidth

```
1  while True:
2      if GlobalMemBandwidth() < THRESH_BW:
3          increment the core limit on the most limited task;
4          sleep(10 µs);
5          continue;
6      for each core C:
7          start[C] = ReadLLCMisses(C);
8      sleep(10 µs);
9      for each core C:
10         end = ReadLLCMisses(C);
11         misses[task_on_core[C]] += end - start[C];
12     find task T that is BE and has the highest misses;
13     decrement task T's core limit and revoke a core;
```

Algorithm 2: The memory bandwidth controller.

```
1  while True:
2      for each core C:
3          T = task_on_core[C]
4          if T is LC and now - GetRequestStartTime(C) ≥
5            THRESH_HT[T]:
6              ban sibling of C;
7          else:
8              unban sibling of C;
9      sleep(10 µs);
```

Algorithm 3: The hyperthread controller.

usage, with the exception that they exclude non-temporal memory accesses, which don't allocate lines in the cache. Fortunately, these are rarely used, but we recommend they be counted in future CPUs. Waiting 10 μs between samples is enough to accurately estimate LLC misses. The bandwidth controller revokes one core from the worst offending task every time it runs until memory bandwidth is no longer saturated. When one task is throttled, another task (that consumes less memory bandwidth) can still use the throttled core.

While Algorithm 2 summarizes this controller's basic behavior, we had to take extra steps to improve its accuracy. First, because `ReadLLCMisses()` initiates PMU counter reads with IPIs (see §5.3), there can be timing skew. Therefore, KSCHED includes the local timestamp counter (TSC), which is stable across cores, when it stores PMU results. This allows us to calculate an LLC miss rate instead of a raw miss count. Second, we discard samples from tasks that have yielded or have been preempted during the measurement interval.

5.2.3 The Hyperthread Controller

Caladan's hyperthread controller detects hyperthread interference and then bans use of the sibling hyperthread until the current request completes (Algorithm 3). Runtimes place timestamps in shared memory to indicate when each hyperthread begins handling a green thread. The hyperthread controller then uses `GetRequestStartTime()` to retrieve these timestamps and check if the current thread has been running for more than a per-task processing time threshold (`THRESH_HT`).

When the threshold has been exceeded, the controller bans use of the sibling hyperthread via KSCHED. The sibling's runtime receives a request from KSCHED to preempt the core and places the current green thread back into its runqueue. The top-level core allocator can detect this as an increase in queueing delay and add back a different (not banned) core. Then KSCHED places the sibling in the shallow `C1` idle state using the `mwait` instruction; `mwait` parks the local hyperthread and reallocates shared physical core resources to the sibling, increasing its performance.

Caladan's hyperthread controller benefits from global knowledge. First, it will only ban a sibling that is handling an LC task if that LC task can be allocated another core, to avoid degrading throughput under high load. Second, if there are not enough cores available, it will prioritize speeding up the green threads that have spent the most time processing a request, keeping tail latency as low as available compute capacity permits. The hyperthread controller can also unban cores, respecting the same priority, when the top-level core allocator needs to allocate a guaranteed core, but none are available due to bans.

Caladan's approach to managing hyperthread interference was inspired by Elfen Scheduling [70]. Our policy for identifying interference is similar to Elfen's refresh budget policy, and both use `mwait` to idle hyperthreads. However, Caladan's approach differs in two key ways. First, Elfen relies on trusted BE tasks to measure interference and yield voluntarily, while Caladan's scheduler makes and enforces these decisions, leveraging the benefits of global knowledge. Second, Elfen can only support pinning one LC task and one BE task to each hyperthread pair. Instead, we allow any pairing (even self pairings) and can handle interference between LC tasks. This enables significantly higher throughput because all logical cores are available for use by any task (§7.3).

5.2.4 An Example: Reacting to Garbage Collection

As an example, we explain how Caladan's scheduler responds when a GC cycle begins, causing memory bandwidth interference for an LC task (the workload depicted in Figure 1). As soon as global memory bandwidth usage exceeds `THRESH_BW`, the memory bandwidth controller will revoke cores from the GC task, revoking one core every 10 μs until total memory bandwidth usage falls below `THRESH_BW` (Algorithm 2). In the meantime, the LC task may suffer from interference, increasing its queueing delay. This will cause the top-level core allocator to grant it additional cores, beginning with any idle cores, but preempting additional cores from the GC task if necessary. It will add one core every 10 μs until the LC task's queueing delay falls below its `THRESH_QD` again (Algorithm 1). Once the GC interference has been successfully mitigated, the LC task will yield the extra cores.

5.3 KSCHED: Fast and Scalable Scheduling

KSCHED's goal is to efficiently expose control over CPU scheduling to the userspace scheduler core. A scheduler core

that relies on the current Linux Kernel system call interface is subject to its limitations; KSCHED must overcome these. First, Linux system calls, like `sched_set_affinity()`, perform computationally expensive work (e.g., locking runqueues) on the core that calls them. Second, Linux system calls block and reschedule while waiting for their operation to complete, preventing the scheduler core from performing other work. Third, Linux system calls can only perform one operation at a time, squandering any opportunity to amortize costs across multiple operations and cores. Finally, cores may only directly read their own performance counters and Linux provides no efficient mechanism to query those on other cores.

KSCHED adopts a radically different approach from Linux's existing mechanisms, supporting direct communication between the scheduler core and kernel code running on other cores via per-core, shared-memory regions. The scheduler core writes commands into these regions and then uses an `ioctl()` to *kick* the remote cores by sending them IPIs. KSCHED then executes the commands (in kernelspace on the remote cores) and writes back results.

KSCHED supports three commands: waking tasks (potentially preempting the current task), idling cores, and reading performance counters. Before preempting a task or idling a core, KSCHED delivers a signal to the runtime to give it a few microseconds to yield cleanly, saving the current green thread's register state and placing it back in the runqueue. Then, to wake a new task on a core, KSCHED locks the task's affinity so that Linux cannot migrate it to another core and calls into the Linux scheduler. To idle a core instead, KSCHED calls `mwait`. Finally, KSCHED can sample any performance counter on any core, and includes the TSC in the response.

When the scheduler kicks a core, the IPI handler immediately processes any pending commands. Commands can also be processed without IPIs by cores that are idle through efficient polling. To achieve this, KSCHED bypasses the standard Linux idle handler, setting a flag that notifies the scheduler core that the current task has yielded voluntarily. KSCHED then checks for new commands; if none are available, it runs the `monitor` instruction, telling the core to watch the cache line containing the shared region. Finally, it parks the core with the `mwait` instruction, placing it in the shallow `C1` idle state. `mwait` monitors cache coherence messages and immediately resumes execution when the shared region is written to by the scheduler core.

One of the most expensive operations that both Linux and KSCHED must perform is sending IPIs. When there are multiple operations, KSCHED leverages the multicast capability of the interrupt controller to send multiple IPIs at once, significantly amortizing costs. To facilitate this, the scheduler core writes all pending operations to shared memory and then passes a list of cores to kick to an `ioctl()` that initiates IPIs. In addition, all of KSCHED's commands are issued asynchronously, so that the scheduler core can perform other work while waiting for them to complete. Finally, KSCHED performs expensive operations such as sending signals and affinitizing tasks to cores on the targeted cores rather than on the scheduler core. In combination, these three properties allow KSCHED to perform scheduling operations with low overhead, enabling Caladan to support high rates of core reallocation and performance counter sampling even with many concurrent tasks (§7.3).

6 Implementation

Caladan is derived from the open-source release of Shenango [61], but we implemented a completely new scheduler and the KSCHED kernel module, which are 3,524 LOC and 533 LOC, respectively. Shenango was a good starting point for our system because of its feature-rich runtime with support for green threads and TCP/IP networking. Moreover, Shenango's runtime is already designed to handle signals to cleanly preempt cores [47].

We modified Shenango's runtime in two important ways. First, Shenango relies on its scheduler core to forward packets in software to the appropriate runtime over shared memory queues. Instead, we linked the `libibverbs` library directly into each runtime, providing fast, kernel-bypass access to networking. This implementation strategy allowed us to completely eliminate the packet forwarding bottlenecks imposed by Shenango and also reduced our scheduler core's exposure to interference, by reducing its memory and computational footprint. Our scheduler core measures packet queueing delay by mapping the NIC's RX descriptor queues (for each task) over shared memory and accessing the packet arrival timestamps encoded in the descriptors by the NIC. Second, we augmented the runtime with support for NVMe storage using Intel's SPDK library to bypass the kernel. These changes required us to add 2,943 new LOC to the runtime, primarily to add integration with `libibverbs` and SPDK.

To support idling in KSCHED, each per-core shared memory region uses a single cache line (64 bytes) because `mwait` can only monitor regions of this size. We packed these cache lines into a contiguous array so that our scheduler core could take advantage of hardware prefetching to speed up polling. KSCHED allows the scheduler core to control which idle state `mwait` enters, but we have not yet explored power management. We also modified the Linux Kernel source to accelerate multicast IPIs; although the Linux Kernel provides an API called `smp_call_function_many()` that supports this feature, it imposes additional software overhead, especially under heavy memory bandwidth interference.

7 Evaluation

We evaluate Caladan by answering the following questions:

1. How does Caladan compare to previous systems (§7.1)?
2. Can Caladan colocate different tasks while maintaining low tail latency and high CPU utilization (§7.2)?

3. How do the individual components of Caladan's design enable it to perform well (§7.3)?

Experimental setup: We evaluate our system on a server with two 12 physical core (24 hyperthread) Xeon Broadwell CPUs and 64 GB of RAM running Ubuntu 18.04 with kernel 5.2.0 (modified to speed up multicast IPIs). We do not consider NUMA, and direct all interrupts, memory allocations, and threads to the first socket. The server is equipped with a 40 Gb/s ConnectX-5 Mellanox NIC and a 280 GB Intel Optane NVMe device capable of performing random reads at 550,000 IOPS. To generate load, we use a set of quad-core machines with 10 Gb/s ConnectX-3 Mellanox NICs connected to our server via a Mellanox SX1024 non-blocking switch. We tune the machines for low latency in accordance with recommended practices, disabling TurboBoost, CPU idle states, CPU frequency scaling, and transparent hugepages [37]. We also disable Meltdown [2] and MDS [24] mitigations, since these vulnerabilities have been fixed by Intel in recent CPUs. When evaluating Linux's performance, we run BE tasks with low-priority using `SCHED_IDLE` and use kernel version 5.4.0 to take advantage of recent improvements to `SCHED_IDLE`. We use *loadgen*, an open-loop load generator, to generate requests with Poisson arrivals over TCP connections [61]. Unless stated otherwise, we configure all Caladan experiments with 22 guaranteed cores for LC tasks, leaving one physical core for the scheduler.

Evaluated applications: We evaluate three LC tasks. First, *memcached* (v1.5.6) is a popular in-memory, key-value store that has been extensively studied [43]. We generate a mix of reads and writes based on Facebook's `USR` request distribution [6] (service times of about 1 μs). Second, *silo* is a state-of-the-art, in-memory, research database [64]. We feed it the TPC-C request pattern, which has high service time variability (20 μs median; 280 μs 99.9%-ile) [63]. Silo is only a library, so we integrated it with a server that can handle RPCs, performing one transaction per request. Finally, we built a new NVMe block storage server inspired by ReFlex [34], that we call *storage*. We added compression (using Snappy [1]) and encryption (using AES-NI [46]) to study the hyperthreading effects of RPC frameworks that rely on vector processing. We preload the SSD with XML-formatted data from Wikipedia [39], and issue requests for blocks of varying lengths (99% 4KB, 1% 44KB) to evaluate service time variability (35 μs and 250 μs for each respective size).

For BE tasks, we use workloads from the PARSEC benchmark suite [9]. In particular, we evaluate *x264*, an H.264/AVC video encoder, *swaptions*, a portfolio pricing tool, and *streamcluster*, an online clustering algorithm. We modified swaptions to use the Boehm garbage collector to allocate its memory objects [10], allowing us to study the interference caused by garbage collection; we call this version *swaptions-GC*. All three workloads exhibit phased behavior, changing their resource usage over regular intervals (some have much larger variance than others). Finally, we evaluate a synthetic antagonist that continuously reads and writes arrays of memory in two configurations: *stream-L2* displaces the L2 cache, while *stream-DRAM* displaces the LLC and consumes all available memory bandwidth.

All applications run in our modified Shenango runtime, which supports standard abstractions such as TCP sockets and the pthread interface (via a shim layer), making it relatively straightforward to port and develop applications (§8).

Parameter tuning: Caladan has three parameters that are user-tunable and can make tradeoffs between latency and CPU efficiency. Appendix A explains how to tune these parameters and shows how sensitive Caladan's performance is to particular choices of these parameters. In our evaluation, we tuned all three for low latency. First, we set `THRESH_QD` (the queueing delay threshold) to 10 μs for all tasks. Second, we set `THRESH_BW` (the memory bandwidth threshold) to 25 GB/s. Finally, we set a `THRESH_HT` (the processing time threshold) for each LC task (not supported for BE tasks). We set it to 25 μs for silo, 40 μs for storage, and infinite for memcached.

Comparison with Parties: Parties [12] is the most relevant prior work for mitigating interference. It builds upon Heracles [38] by adding support for multiple LC tasks. Ideally, we would compare directly to Parties, but its source code is not publicly available, and we were unable to obtain it from the Parties authors. Instead, we reimplemented Parties in accordance with the details described in its paper.

By implementing Parties ourselves, we were able to use the same runtime system for both Caladan and Parties, so they could benefit equally from kernel-bypass I/O, allowing us to evaluate only differences in scheduling policy. We did not implement some components in Parties that were not relevant to our experiments. Specifically, our workloads do not contend over disk, network, or memory capacity. Managing these resources is important but unrelated to our focus on CPU interference. Moreover, we did not include the CPU frequency scaling controller, as reducing energy consumption is outside the scope of our work. We did implement all of Parties' key mechanisms, including core allocation, CAT, and an external measurement client that samples tail latencies over 500 ms periods. We also invested considerable effort in tuning Parties' latency thresholds to yield the best possible performance.

Normally, Parties leaves hyperthreads disabled because it is unable to manage this form of interference, reducing its CPU throughput. Instead, we enabled hyperthreads with a policy that prefers self pairings. For specifically memcached—the workload we evaluated—this forms a complementary pairing that has minimal effect on latency, but allowed us to conduct a direct comparison with the same number of cores. The addition of kernel-bypass networking and hyperthread pairing enable our version of Parties to significantly outperform the reported performance of the original, so we refer to it as *Parties**.

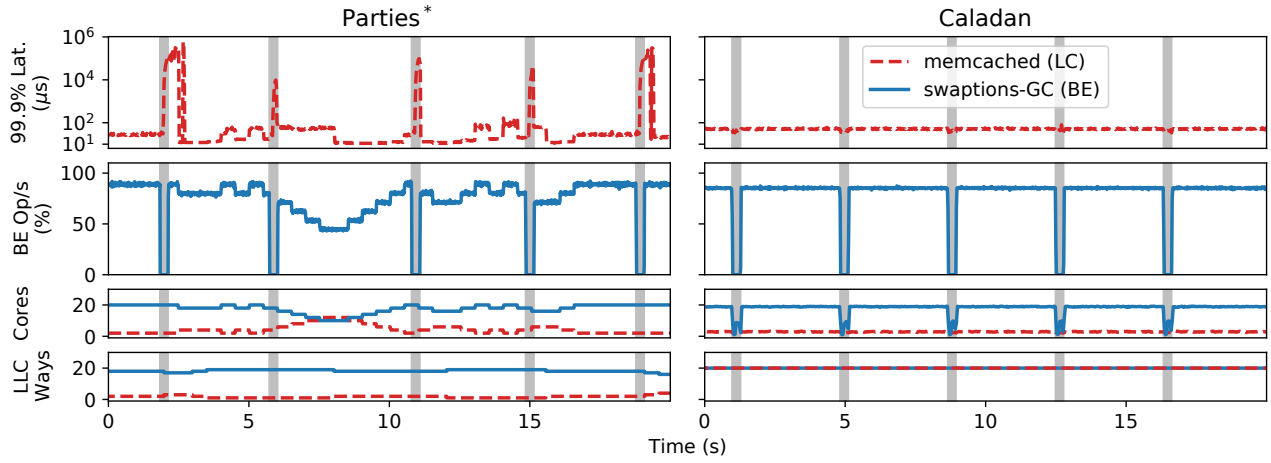


Figure 5: Timeseries of memcached colocated with a garbage-collected BE task for Parties* (left) and Caladan (right). Gray bars indicate GC cycles in the BE task. The Parties* resource controller algorithm is unable to provide performance isolation and high CPU utilization when tasks have dynamic resource demands, while Caladan maintains both. Top graphs have log-scaled y-axes.

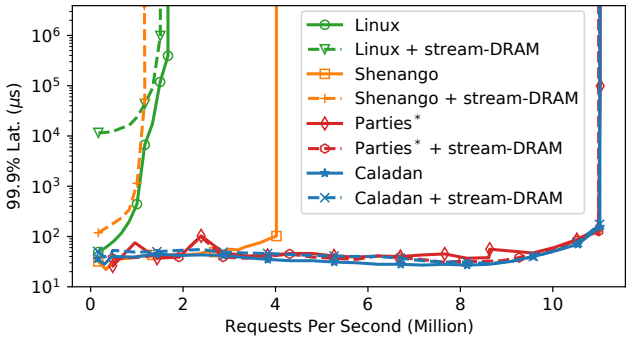


Figure 4: Constant memory bandwidth interference degrades memcached performance for Linux and Shenango, but Caladan and Parties* can mitigate it. Note the log-scaled y-axis.

7.1 Comparison to Other Systems

Constant interference: To demonstrate the necessity of managing interference, we first compare Parties* and Caladan to systems that do not explicitly manage interference. We evaluate a relatively less challenging scenario, where an LC task (memcached) is colocated with stream-DRAM, a BE task that generates constant memory bandwidth interference.

Figure 4 illustrates that, as expected, both Linux and Shenango suffer significant increases in tail latency in the presence of colocation, reaching tail latencies up to 235× and 6× higher than without interference, respectively. Shenango's throughput also decreases by 75% in the presence of interference, because its scheduler core becomes overloaded with packet processing, due to higher cache miss rates and memory access latency caused by stream-DRAM. In contrast, Caladan and Parties* are both able to maintain similar tail latency with and without interference, because they manage it explicitly. Both also achieve much higher throughput than Linux and Shenango because runtime cores send and receive packets directly using our runtime's kernel-bypass network stack (§6), preventing the Linux network stack or the scheduler core from becoming a bottleneck. While adapting Shenango to use our runtime's kernel-bypass network stack would eliminate this throughput bottleneck, it would not improve the tail latency of LC tasks suffering from interference.

Phased interference: We now focus on interference caused by phased behavior, a more difficult and realistic case that Caladan is designed to solve. We revisit the garbage collection experiment from Section §2, colocating an instance of memcached with swaptions-GC. We issue 800,000 requests per second to memcached for a period of 120 seconds and measure its tail latency over 20 ms windows. We show the first 20 seconds of the experiment in Figure 5, which we found to be representative of the behavior during the entire experiment.

Caladan throttles the BE task's cores as soon as each GC cycle starts, preventing latency spikes, and it gives back cores to the BE task as soon as the GC cycle ends, maintaining high BE throughput. Parties* attempts to find an allocation of cores and cache ways that minimizes latency and maximizes resources for the BE task, but it is unable to converge when resource demands are shifting at timescales much smaller than its 500 ms adjustment interval. Often Parties* grants additional cores in response to GC cycles, but these adjustments happen too slowly to prevent latency spikes. As a result, Parties* experiences 99.9% latency that is 11,000× higher than Caladan during GC cycles. In addition, Parties* also harms BE throughput, achieving an average of 5% less than Caladan because it punishes swaptions-GC by too much and for too long. These results show that faster reaction times are essential when handling tasks with phased behaviors.

7.2 Diverse Colocations

Two tasks: To understand if Caladan can maintain its benefits in diverse situations, we evaluate 15 colocations between pairs of LC and BE tasks with different resource usages, service

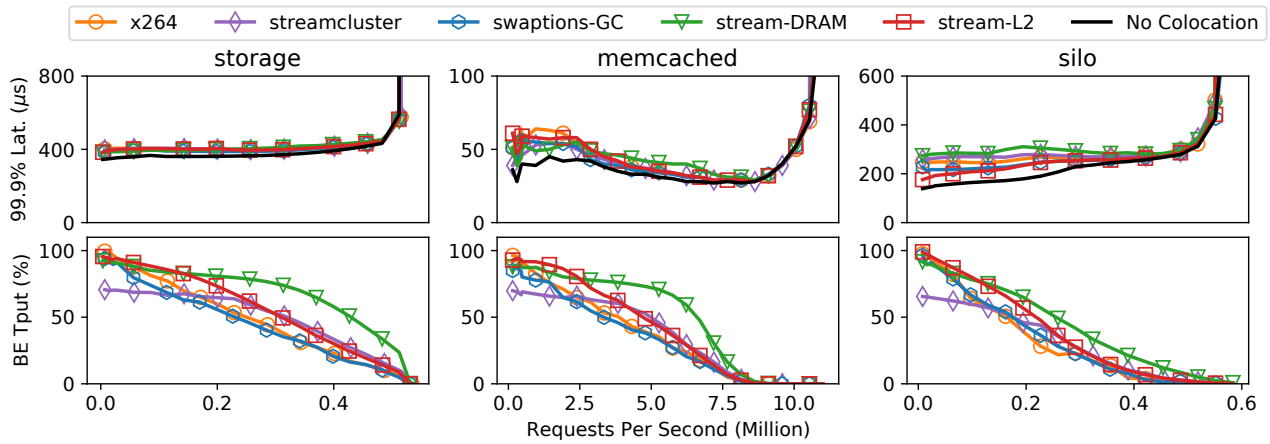


Figure 6: Caladan can colocate many combinations of LC and BE applications with only modest latency penalties for LC tasks (top), while maintaining excellent throughput for BE tasks (bottom).

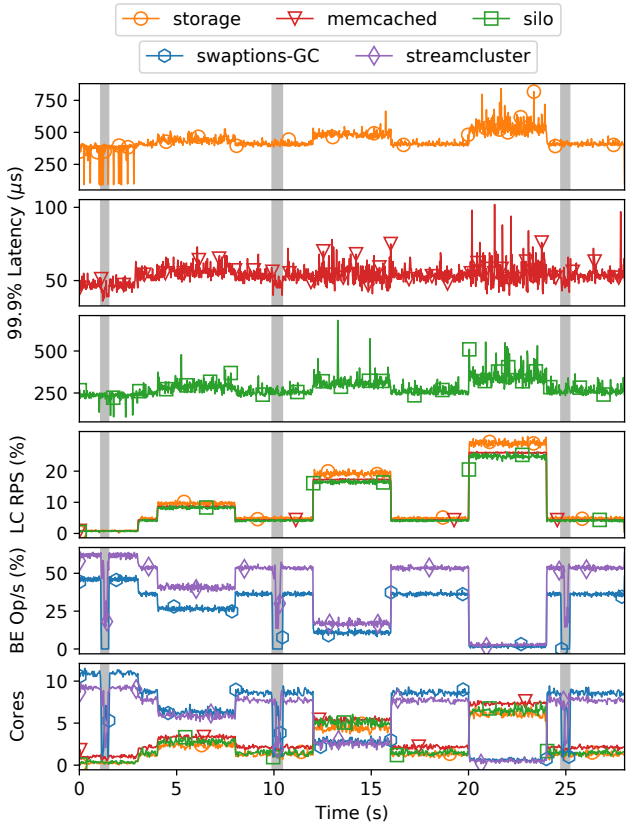


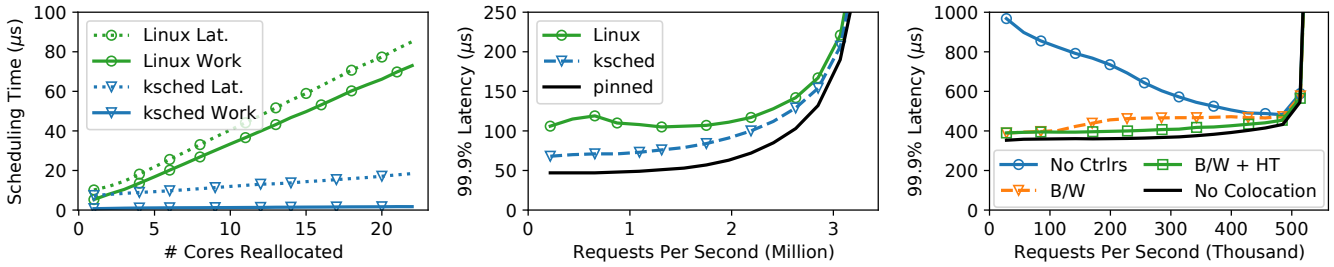
Figure 7: Caladan can colocate multiple LC and BE apps while providing performance isolation and high utilization. Gray bars indicate GC cycles in swaptions-GC.

time distributions, and throughputs. 9 out of 15 pairings include BE tasks with phased behaviors. We consider the impact on each LC task's tail latency and the amount of throughput the BE task can achieve by using burstable cores.

In this experiment (Figure 6), each data point represents a different fixed average load offered to an LC task (columns), while it is paired with a BE task (colors/linetypes). Caladan is highly effective at mitigating interference: storage and memcached achieve nearly the same tail latency as they do without colocation. Silo experiences a small increase in tail latency at low load because it is sensitive to LLC interference, leading to service time but not queueing delay increases. At higher load, silo generates self interference, so it experiences similar tail latency with and without colocation. Overall, Caladan can easily maintain microsecond-level tail latency under challenging colocation conditions.

At the same time, Caladan yields excellent BE task throughput. The exact BE throughput depends on the degree of resource contention with the LC task. For example, x264, swaptions-GC, and stream-L2 use less memory bandwidth (on average), so they can linearly trade CPU time with the LC task. Streamcluster and stream-DRAM both consume a larger amount of memory bandwidth, so they are throttled by our memory bandwidth controller. However, they also pair well with LC tasks as siblings (especially memcached) because they use different physical core resources. At higher LC load, these BE tasks are given fewer cores so they use less memory bandwidth and are then throttled less. Overall, BE throughput depends on the specific interactions between the BE and LC tasks, and varies with LC load. To the best of our knowledge, Caladan is the first system to achieve both microsecond-level LC tail latency and high BE throughput under such a broad range of conditions.

Many tasks: To demonstrate Caladan's ability to manage many tasks simultaneously, we colocate all 3 of the LC tasks along with swaptions-GC and streamcluster (each LC task is configured with 6 guaranteed cores). Figure 7 shows a 30-second trace from this experiment, during which the load of each of the LC apps changes multiple times (4th graph) and swaptions-GC performs garbage collection three times (gray bars). When load or interference changes, Caladan converges nearly instantly. When GC is not running, the combination of streamcluster and swaptions-GC does not saturate memory bandwidth. However, when GC begins, both tasks together saturate DRAM bandwidth and are throttled by Caladan. Cal-



(a) Scheduling Scalability **(b)** Aggregated Latency of 11 LC Apps **(c)** Contribution of Sub-controllers

Figure 8: (a) KSCHED dramatically increases scheduling scalability over Linux. (b) Aggregated latency for 11 LC apps scheduled using KSCHED vs. Linux mechanisms. (c) The impact of each Caladan sub-controller on tail latency for storage paired with stream-DRAM.

adan's fast reactions (up to 230,000 core reallocations per second) enable all three LC tasks to maintain low tail latency (top three graphs) throughout constantly shifting load and interference.

7.3 Microbenchmarks

KSCHED: To evaluate the benefits of KSCHED's faster scheduling operations, we run a simple microbenchmark where we continuously rotate tasks to different cores. To measure scalability, we migrate different numbers of tasks together in groups. We run the benchmark both with KSCHED and with a variant that uses standard Linux system calls such as `sched_setaffinity()`, `tgkill()`, and `eventfd()`.

Figure 8a shows both the scheduling work (time spent by the scheduler core) and the scheduling latency (time until the migration completes) per migration. Both metrics benefit tremendously from KSCHED's multicast IPIs, allowing it to amortize the cost of multiple simultaneous migrations. By contrast, Linux's system call interface suffers from overhead and because it cannot support batching; operations must be serialized, increasing scheduling work by 43× and scheduling latency by 5× when moving 22 tasks. In addition, KSCHED maintains low scheduling work even with many tasks by offloading expensive operations such as sending signals to remote cores.

We demonstrate the value of these improvements in an experiment with 11 synthetic LC tasks and 2 synthetic non-interfering BE tasks. The LC tasks have 5 μs average service times that are exponentially distributed and each is configured with 2 guaranteed cores. We compare against an earlier version of Caladan that employed the Linux scheduling mechanisms evaluated above. In Figure 8b, we show that Caladan is able to maintain much lower tail latency for the LC tasks (close to that of running with cores pinned). In this experiment, Caladan performs up to 560,000 core reallocations per second at its peak (at a load of 0.65 million RPS), while the version using Linux mechanisms bottlenecks at around 285,000 allocations per second. KSCHED provides similar benefits for sampling performance counters (not shown).

Controllers: We found that both the memory bandwidth and hyperthread controllers were necessary in order to ensure

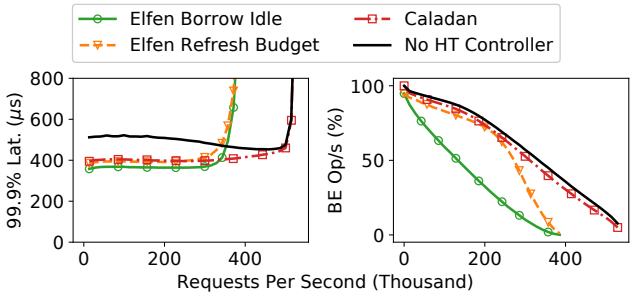


Figure 9: Caladan enables higher LC throughput than Elfen by allowing arbitrary tasks to co-run on a physical core (including the same LC task).

isolation across a variety of tasks and loads. To provide one concrete example, Figure 8c evaluates the contribution of each controller module to the storage LC task when colocated with the stream-DRAM BE task. At very low loads, the bandwidth controller is sufficient to provide low tail latency. This is because as Caladan revokes cores from the BE task, it leaves the hyperthread pair cores of the LC task idle, rendering the hyperthread controller unnecessary. However, at higher LC loads, both controllers are necessary in order for the storage task to achieve nearly the same tail latency as it would have without colocation.

Next we focus on the hyperthread controller and evaluate the benefits of allowing any two tasks to co-run on a physical core (e.g., two LC tasks or even two hyperthreads in the same task). Figure 9 compares Caladan to two modified versions of Caladan that implement Elfen's [70] scheduling policies, when colocating storage and stream-L2. Elfen's borrow idle policy disallows co-running, only allowing the BE to run on a physical core when it is not being used by the LC; this yields low tail latency for the LC task but also low BE throughput. Elfen's refresh budget policy, which Caladan generalizes (§5.2.3), yields higher BE throughput at the cost of a slight increase in tail latency, demonstrating the benefits of using both hyperthreads simultaneously. Caladan achieves 37% more LC throughput than Elfen by enabling the LC task to co-run with itself. Similarly, at low LC loads, Caladan is able to achieve 5% higher BE throughput than Elfen since BE tasks can use both hyperthread lanes. Finally, running Caladan with the hyperthread controller disabled yields slightly higher BE

throughput but at a cost of up to 117 μs higher tail latency, highlighting the need to explicitly manage hyperthread interference to achieve both high throughput and low tail latency.

8 Discussion

Compatibility: Caladan requires applications to use its runtime system because it depends on it to export control signals to the scheduler, and to rapidly map threads and packet processing work across a frequently changing set of available cores. Our runtime is not fully Linux compatible, but it provides a realistic, concurrent programming model (inherited from Shenango) that includes threads, mutexes, condition variables, and synchronous I/O [47]. Caladan also includes a partial compatibility layer for system libraries (e.g., libpthread) that can support PARSEC [9] without modifications, giving us some confidence our design is flexible enough to support unmodified Linux applications in the future. Applications that do not use our runtime can coexist on the same machine, but they must run on cores that are not managed by Caladan, and they cannot be throttled if they cause interference.

The more fundamental requirement for Caladan is the need for LC tasks to expose their internal concurrency to the runtime (e.g., by spawning green threads), potentially requiring changes to existing code. If there is insufficient concurrency, a task will be unable to benefit from additional cores, hindering Caladan's ability to manage shifts in load or interference. In general, we recommend that tasks expose concurrency by spawning either a thread per connection or a thread per request. For example, normally memcached multiplexes multiple TCP connections per thread, but we modified it to instead spawn a separate thread to handle each TCP connection.

On the other hand, Caladan can support BE tasks that do not expose their internal concurrency, as it can still throttle them if they cause too much interference. For example, if a BE task is single-threaded (i.e., has no concurrency), and it consumes too much memory bandwidth, Caladan will oscillate between giving it one and zero cores, effectively time multiplexing its memory bandwidth usage. However, BE tasks can optionally achieve higher performance by exposing their internal concurrency: load will be more evenly balanced and they will be able to take advantage of burstable cores.

Limitations: Our current implementation of Caladan has two limitations. First, it is unable to manage interference across NUMA nodes. NUMA introduces additional shared resources that are vulnerable to interference, including an inter-socket interconnect and separate memory controllers per node. Fortunately, high-precision performance counters are available for these resources, and we plan to explore NUMA-aware interference mitigation strategies in the future, such as revoking cores or migrating tasks between nodes. Second, our scheduling policies do not minimize the threat of transient execution attacks across hyperthread siblings [3, 11, 65]. Ideally, only mutually-trusting tasks should be allowed to run on sibling cores. At the time of writing, a similar capability is under development for the Linux Kernel [13].

Future work: One promising opportunity for future work is to incorporate hardware partitioning back into Caladan's design. For example, if a BE task uses high memory bandwidth and lacks temporal locality, many of the cache lines it occupies in the LLC will be wasted. Under these conditions, Caladan is still effective at preventing latency increases, but it must allocate extra cores to victim tasks. If future hardware partitioning mechanisms could be designed to accommodate frequently shifting LLC usage—or if static LLC usage could be identified and managed through existing mechanisms—CPU efficiency could be further improved.

9 Related Work

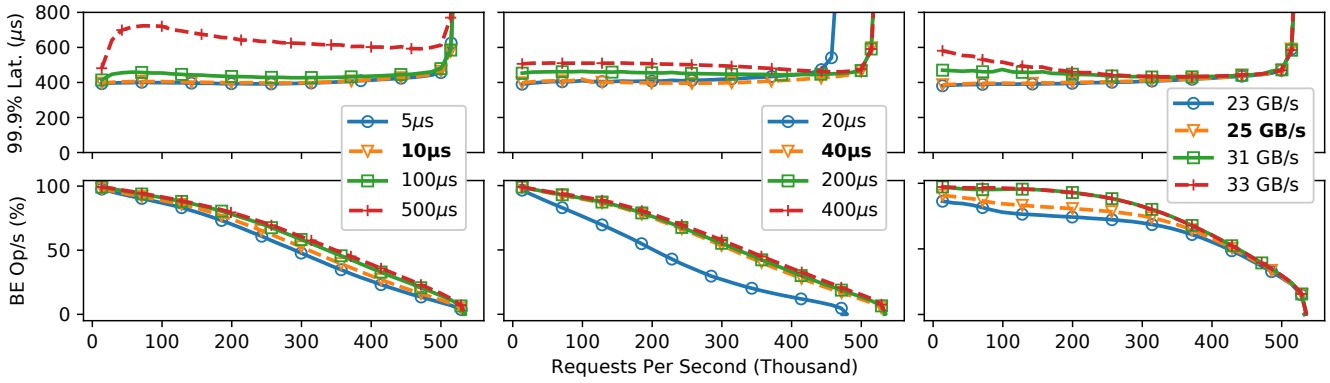
Interference management: Many prior systems manage interference between LC and BE tasks by statically partitioning resources [19, 28, 50, 62]. While this approach can reduce interference, it sacrifices CPU utilization because each task must be provisioned enough resources to accommodate peak load. Heracles [38], Parties [12], and PerfIso [27] instead adjust partitions dynamically. However, unlike Caladan, these systems cannot manage changes in interference while maintaining microsecond latency and high utilization.

Efforts to isolate the network [35, 36, 53] or storage [34] are complementary to Caladan. We do not currently focus on power management [32, 58] or TurboBoost [23], because we optimize for the setting in which all cores are fully utilized, but it should be possible to integrate power management with Caladan to improve its CPU efficiency at lower utilization.

User-level core allocators: To enable low latency in the face of fluctuating load, systems like IX [8], PerfIso [27], Shenango [47], and Arachne [52] introduce user-level core allocators that estimate load and reallocate cores to BE tasks when they are not needed by LC tasks. Similarly, TAS [33] and Snap [42] adjust cores in response to changes in packet processing load. Like these systems, Caladan manages changes in load through core allocations, but it goes a step further by using core allocation to manage interference too.

Scheduling optimizations: Shinjuku [30] proposes fine-grained preemption to reduce tail latency, using Dune [7] to provide fast, direct access to IPIs in userspace. KSCHED includes kernel optimizations that allow for similar performance when sending an IPI to a single core, but it speeds up IPIs over Shinjuku's reported speeds when sending more than one IPI at a time because of its multicast IPI optimization.

Dataplane systems: There has also been significant work on optimizing OS networking for throughput and latency [8, 29, 33, 48, 52, 55]. ZygOS proposes work stealing as a technique to reduce tail latency under variable service times [51]. Arachne [52] and Shenango [47] build a similar latency reduction strategy on top of green threads to improve programmability. Caladan builds upon all of these ideas to eliminate



(a) THRESH_QD **(b)** THRESH_HT **(c)** THRESH_BW

Figure 10: Parameter sensitivity for the storage LC task; the parameters used in our evaluation appear in bold. (a) THRESH_QD allows an operator to achieve better tail latencies at the expense of BE throughput. (b) THRESH_HT reins in the latency of long requests, but setting it too low reduces BE throughput. (c) THRESH_BW is set to avoid exponential increases in memory access latencies.

network processing and queueing bottlenecks, allowing it to manage interference unperturbed by software overheads or load imbalances.

10 Conclusion

This paper presented Caladan, an interference-aware CPU scheduler that significantly improves performance isolation while maintaining high CPU utilization. Caladan's effectiveness comes from its speed: by matching control signals and actions to the same timescale that interference affects performance, Caladan can mitigate interference before it can harm quality of service. Caladan relies on a carefully selected set of control signals to manage multiple forms of interference in a coordinated fashion, and combines a wide range of optimizations to rapidly gather control signals and make core allocations faster. These contributions allow Caladan to deliver microsecond-level tail latency and high CPU utilization while colocating multiple tasks with phased behaviors.

11 Acknowledgments

We thank our shepherd Kathryn S. McKinley, the anonymous reviewers, Frans Kaashoek, Malte Schwarzkopf, Akshay Narayan, and other members of PDOS for their useful feedback. We thank CloudLab [17] and Eitan Zahavi at Mellanox for providing equipment used to test and evaluate Caladan. This work was funded by the DARPA FastNICs program under contract #HR0011-20-C-0089, by a Facebook Research Award, and by a Google Faculty Award.

A Parameter Tuning and Sensitivity

In this Appendix, we describe how to set Caladan's three user-tunable parameters and show how sensitive Caladan's performance is to particular choices of these parameters. To illustrate the behavior of THRESH_QD and THRESH_HT, we colocate the storage workload with stream-L2. For THRESH_BW, we colocate the storage workload with stream-DRAM. In each case, we vary a single parameter, while fixing other parameters to the values used in our evaluation.

THRESH_QD represents the per-task queueing delay limit before the top-level core allocator tries to grant another core. As shown in Figure 10a, an operator can trade some LC tail latency for higher BE throughput using a value of THRESH_QD larger than Caladan's default 10 μs. For example, a THRESH_QD of 100 μs enables 7% more BE throughput at the cost of 54 μs higher LC tail latency for these workloads. We chose to optimize for tail latency, and found that values below 10 μs degraded BE throughput without further improving LC tail latency.

THRESH_HT places a worst-case limit on how long a request can be delayed by a task generating interference on its hyperthread sibling. If it is set too low (i.e., most request processing requires a dedicated physical core), BE throughput will suffer and LC latency will degrade at high load due to insufficient compute capacity. For a skewed service time distribution, like our storage workload, choosing a value above the median is a good heuristic. Figure 10b illustrates that setting THRESH_HT below the median of 35 μs significantly lowers BE throughput, while values that are slightly above the median yield increased BE throughput and good tail latency. For workloads with service times less than 5 μs (e.g., memcached), we recommend setting THRESH_HT to infinite because mwait requires a few microseconds to park a hyperthread.

Finally, THRESH_BW represents the global maximum allowed memory bandwidth usage before Caladan begins to throttle tasks. THRESH_BW should be set once per machine to a bandwidth just low enough to avoid the exponential increase in memory access latency that occurs close to memory bandwidth saturation. We use 25 GB/s for our machine (70–80% of its capacity), which keeps memory latency low for any access pattern. Figure 10c shows this setting trades a small amount of BE throughput in exchange for predictable latency.

B Artifact

Caladan's source code, ported applications, and experiment scripts can be found at https://github.com/shenango/caladan-all.

References

[1] Snappy. https://github.com/google/snappy.

[2] Intel analysis of speculative execution side channels. Technical report, January 2018.

[3] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri. Port contention for fun and profit. In *IEEE S&P*, 2019.

[4] I. Arapakis, X. Bai, and B. B. Cambazoglu. Impact of response latency on user behavior in web search. In *SIGIR*, 2014.

[5] D. Ardelean, A. Diwan, and C. Erdman. Performance analysis of cloud applications. In *NSDI*, 2018.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[7] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*, 2012.

[8] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *TOCS*, 2017.

[9] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[10] H. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *PLDI*, 1991.

[11] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[12] S. Chen, C. Delimitrou, and J. F. Martínez. PARTIES: QoS-aware resource partitioning for multiple interactive services. In *ASPLOS*, 2019.

[13] J. Corbet. Completing and merging core scheduling. https://lwn.net/Articles/820321/, Sept. 2020.

[14] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[15] C. Delimitrou and C. Kozyrakis. QoS-aware scheduling in heterogeneous datacenters with Paragon. *TOCS*, 2013.

[16] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *ASPLOS*, 2014.

[17] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *USENIX ATC*, 2019.

[18] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sánchez. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. In *HPCA*, 2018.

[19] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. In *SIGCOMM*, 2020.

[20] T. Harris, M. Maas, and V. J. Marathe. Callisto: co-scheduling parallel runtime systems. In *EuroSys*, 2014.

[21] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *HPCA*, 2016.

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[23] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. F. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *HPCA*, 2015.

[24] Intel. Microarchitectural data sampling. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-analysis-microarchitectural-data-sampling.

[25] Intel Corporation. *Intel 64 and IA-32 Architectures Performance Monitoring Events*, December 2017.

[26] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, April 2020.

[27] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. R. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. PerfIso: Performance isolation for commercial latency-sensitive services. In *USENIX ATC*, 2018.

[28] S. A. Javadi, A. Suresh, M. Wajahat, and A. Gandhi. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *SoCC*, 2019.

[29] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *NSDI*, 2014.

[30] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *NSDI*, 2019.

[31] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks. Profiling a warehouse-scale computer. *IEEE Micro*, 2016.

[32] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *MICRO*, 2015.

[33] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP acceleration as an OS service. In *EuroSys*, 2019.

[34] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash $\approx$ local flash. In *ASPLOS*, 2017.

[35] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM*, 2015.

[36] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster, et al. PicNIC: predictable virtualized NIC. In *SIGCOMM*. 2019.

[37] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *EuroSys*, 2014.

[38] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *ISCA*, 2015.

[39] M. Mahoney. Large text compression benchmark. http://www.mattmahoney.net/text/text.html, 2011.

[40] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[41] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.

[42] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. E. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.

[43] Memcached community. memcached – a distributed memory object caching system. https://memcached.org/.

[44] Michael Larabel. Intel hyper threading performance with a Core i7 on Ubuntu 18.04 LTS. https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=4, 2018.

[45] J. Nakajima and V. Pallipadi. Enhancements for hyper-threading technology in the operating system: Seeking the optimal scheduling. In *WIESS*, 2002.

[46] OpenSSL. OpenSSL cryptography and SSL/TLS toolkit. https://openssl.org/.

[47] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[48] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *TOCS*, 2015.

[49] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, 2017.

[50] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *OSDI*, 2014.

[51] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *SOSP*, 2017.

[52] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. K. Ousterhout. Arachne: Core-aware thread management. In *OSDI*, 2018.

[53] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: scalable NIC for end-host rate limiting. In *NSDI*, 2014.

[54] S. Ren, Y. He, S. Elnikety, and K. S. McKinley. Exploiting processor heterogeneity in interactive services. In *ICAC*, 2013.

[55] L. Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX ATC*, 2012.

[56] D. Sánchez and C. Kozyrakis. Scalable and efficient fine-grained cache partitioning with Vantage. *IEEE Micro*, 2012.

[57] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[58] E. Sharafzadeh, S. A. S. Kohroudi, E. Asyabi, and M. Sharifi. Yawn: A CPU idle-state governor for data-center applications. In *APSys*, 2019.

[59] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.

[60] The Go Community. The go programming language. https://golang.org.

[61] The Shenango Authors. Shenango's open-source release. https://github.com/shenango/shenango.

[62] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. J. Argyraki, S. Ratnasamy, and S. Shenker. ResQ: Enabling SLOs in network function virtualization. In *NSDI*, 2018.

[63] TPC. TPC-C benchmark. http://www.tpc.org/tpcc/.

[64] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[65] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: rogue in-flight data load. In *IEEE S&P*, 2019.

[66] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[67] R. Wang and L. Chen. Futility scaling: High-associativity cache partitioning. In *MICRO*, 2014.

[68] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[69] H. Yang, A. D. Breslow, J. Mars, and L. Tang. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *ISCA*, 2013.

[70] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *USENIX ATC*, 2016.

[71] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.

[72] L. Zhao, R. R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *PACT*, 2007.

[73] H. Zhu and M. Erez. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. In *ASPLOS*, 2016.