Peafowl: In-application CPU Scheduling to Reduce Power Consumption of In-memory Key-Value Stores

Esmail Asyabi
Boston University
easyabi@bu.edu

Azer Bestavros
Boston University
best@bu.edu

Erfan Sharafzadeh
Johns Hopkins University
erfan@cs.jhu.edu

Timothy Zhu
The Pennsylvania State University
tuz68@psu.edu

ABSTRACT

The traffic load sent to key-value (KV) stores varies over long timescales of hours to short timescales of a few microseconds. Long-term variations present the opportunity to save power during low or medium periods of utilization. Several techniques exist to save power in servers, including feedback-based controllers that right-size the number of allocated CPU cores, dynamic voltage and frequency scaling (DVFS), and c-state (idle-state) mechanisms. In this paper, we demonstrate that existing power saving techniques are not effective for KV stores. This is because the high rate of traffic even under low load prevents the system from entering low power states for extended periods of time. To achieve power savings, we must unbalance the load among the CPU cores so that some of them can enter low power states during periods of low load. We accomplish this by introducing the notion of in-application CPU scheduling. Instead of relying on the kernel to schedule threads, we pin threads to bypass the kernel CPU scheduler and then perform the scheduling within the KV store application. Our design, Peafowl, is a KV store that features an in-application CPU scheduler that monitors the load to learn the workload characteristics and then scales the number of active CPU cores when the load drops, leading to notable power savings during low or medium periods of utilization. Our experiments demonstrate that Peafowl uses up to 40-54% lower power than state of the art approaches such as Rubik and μDPM.

SoCC '20, October 19–21, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421298

CCS CONCEPTS

• **Software and its engineering** → *Power management.*

ACM Reference Format:
Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, and Timothy Zhu. 2020. Peafowl: In-application CPU Scheduling to Reduce Power Consumption of In-memory Key-Value Stores. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421298

1 INTRODUCTION

In-memory Key-Value (KV) stores are non-persistent storage backbones for an ever-growing number of large-scale applications, ranging from graph processing in social media to event log processing in cybersecurity [53, 62], retail shopping carts, machine learning parameter servers [34], sequences in distributed synchronization [25], and application data caching [32]. Many of these applications exhibit a high fan-out pattern, where responding to a single request requires responses from hundreds of KV store services. Hence, the tail latency of KV stores (slowest reply) often impacts the overall performance [10, 13, 24, 42]. To meet strict tail latency goals, service providers budget individual cache node compute capacity for double or even triple normal peak request rates to allow for traffic spikes [4, 8, 15, 40, 56]. Cache nodes, therefore, notably contribute to the cost and energy consumption of data centers [18].

The traffic load sent to KV stores often varies both over long timescales (e.g., minutes to hours) and short timescales (e.g., microseconds to milliseconds) [1, 4, 36, 42, 58]. For example, Facebook's ETC workload follows a diurnal pattern with 2× load variations over the course of a day [4]. On the other hand, Google's Gmail workload exhibits micro bursts that result in sudden CPU usage spikes in the span of microseconds [42]. Long-term variations present an opportunity to leverage elastic KV stores to right-size the amount of resources allocated to KV stores. However, it is still an open challenge to build KV stores to elastically scale to short-term bursts in workloads. Bursty workloads cause short term spikes and queueing delays that can significantly increase tail latency [8, 27]. The goal of this work is to build an elastic KV store that can effectively deal with short-term variability to meet microsecond-scale tail-latency bounds while gradually adjusting to long-term variability to achieve energy-proportionality [27, 36].

One approach to building energy-proportional KV stores is to use feedback-based controllers that monitor and measure the load, compare it to predefined thresholds, and scale the number of active cores to save power during periods of low or medium utilization [36, 64]. Feedback-based controllers can adapt to diurnal variations; however, they are too slow (operating at second-scale intervals) to cope with short-term burstiness, lengthening tail latency [27]. Additionally, they rely on the operating system to schedule KV store threads once the number of active cores is determined. OS CPU schedulers (e.g., Linux CFS scheduler [43]), however, operate at millisecond timescales, aggravating microsecond-scale tail latency [3, 6, 31, 42, 47]. Another approach is to exploit the latency slack (i.e., latency goal minus current latency) and slow down [27] or delay request processing [8] through DVFS and sleep states to save power while executing requests just in time. We will show that the short service times and high arrival rates of KV stores do not provide much opportunity for power saving with these approaches.

In this paper, we present Peafowl, an elastic in-memory KV store designed to exploit load variability to save power while maintaining low *microsecond-scale tail latency.* Peafowl gradually adapts to long-term variability to save power and quickly copes with short-term variability to avoid tail latency problems. The key insight behind Peafowl's design is to handle the scheduling in the application rather than the operating system. OS schedulers are designed to support general use cases that can tolerate millisecond-scale scheduling quantum slices, but they can introduce significant tail latency issues for low latency applications. OS scheduling can present even greater tail latency problems when the number of CPU cores is elastically scaled to save power. Peafowl pins threads to CPU cores and performs all of the power-aware scheduling within the application using in-application knowledge to characterize the workload and rightsize the number of active CPU cores in response to load.

This paper makes the following contributions:

- We show that current power-aware mechanisms are ineffective when it comes to the short service time and high arrival rate of KV stores.
- We demonstrate the effectiveness of in-application scheduling for maintaining microsecond-scale tail latency and leveraging in-application knowledge.
- We have implemented a prototype of Peafowl and extensively evaluated it to demonstrate its effectiveness for real-world workload traces. The results demonstrate that Peafowl is able to significantly save power up to 36% while keeping tail latency at microsecond-scale.
- Our implementation of Peafowl is publicly available at https://github.com/showanasyabi/peafowl-kvs.

2 ELASTIC KV STORES

In this section, we study the essential properties of an elastic KV store as a solution to achieve energy proportionality. We then discuss why existing solutions are not able to address the hard problem of energy proportionality in in-memory KV stores.

2.1 Elastic KV store requirements

Energy proportionality. Several studies have reported that KV store workloads operate frequently at low or medium utilization due to the diurnal/periodic nature of user traffic [4, 18, 66]. Therefore, elastic KV stores can exploit load variability to enforce energy proportionality by scaling the processing capacity to match the available work [36], leading to a significant reduction of power consumption during low or medium utilization periods. Given the widespread adoption of in-memory KV stores, low power consumption of cache nodes during off-peak periods raises CPU efficiency in data centers, where a even small improvement in CPU efficiency saves millions of dollars and reduces data-center footprints and environmental impacts [36, 42].

Microsecond-scale tail latency. Many applications that exhibit a high fan-out pattern use KV stores as caches [13]. These applications can issue tens to hundreds of sub-requests to satisfy a single user request [10, 11, 19, 36, 45, 57, 65]. The slowest response time to these sub-requests determines the overall response time. For these applications, therefore, the overall performance is determined by the tail of the latency distribution (e.g., 99th percentile) rather than average latency [8, 13, 21, 27, 36, 42], leading to many efforts to keep tail latency of KV stores at microsecond scale [7, 13, 45]. We argue that any solution for designing elastic KV stores must not come at the expense of sacrificing tail latency.

Ease of deployment, Generality. Data centers such as Google's compute cluster host several types of applications to raise the utilization [23, 42]. Therefore, solutions that rely on significant modifications to the OS or device drivers are less practical as they transform general-purpose machines to single-purpose ones, which not only waste resources but also require substantial maintenance and verification efforts [37]. More importantly, we argue that any suggested solution must be readily deployable in data centers. Some existing

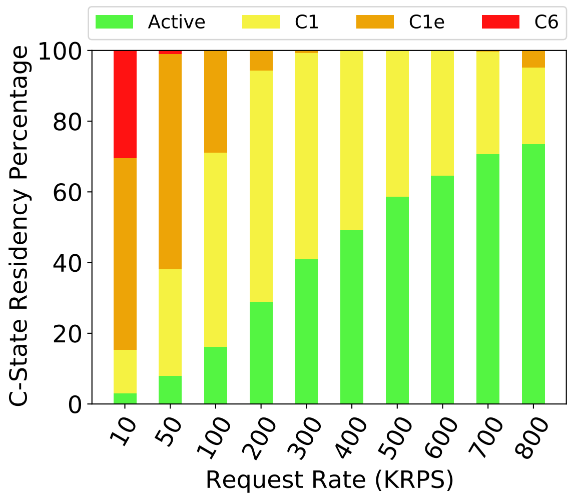

Figure 1: Fraction of time spent in different idle states. CPU cores cannot go to the deepest idle-state (C6) even when the load is as low as 100K (10K per each core) requests per second (RPS).

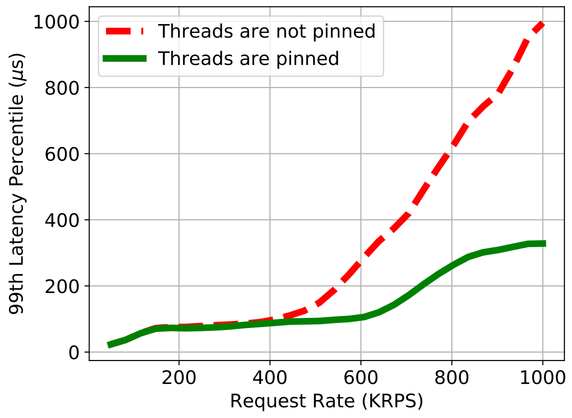

Figure 2: The impact of OS scheduler on KV store tail latency. When KV store threads are not pinned, the OS CPU scheduler notably impacts tail latency.

solutions have built on the assumption that the hardware offers particular features (e.g., per core DVFS [27] or resource disaggregation [41]) that are not currently offered by all hardware vendors, making these solutions less practical.

2.2 Existing systems

Unfortunately, existing solutions do not satisfy the combination of requirements outlined in Section 2.1. They fall into the following categories:

Idle states. CPUs feature several power saving modes called c-states (idle-states) to save power during idle periods [8, 9, 39]. Intel Xeon 6130 CPUs, for example, feature C0 (shallowest), C1, C1e, and C6 (deepest) states. C0 is the operational mode when the CPU executes instructions and no power is saved. As the sleep mode gets deeper, more components are turned off and more power is saved; however, the CPU will require more time to wake up (i.e., return to C0 mode) [14, 17]. Intel Xeon 6130 CPUs require 133μs to return to C0 from C6. Since transitions to deep c-states are costly, a CPU core enters a particular c-state only when it predicts the next idle period is greater than a predefined threshold, known as the target residency time of the c-state [8]. In other words, the target residency time is the minimum idle period for a specific c-state to be profitable energy-wise [26]. For Intel Xeon 6130 CPUs, the target residency time for the C6 state is 600μs. Although c-states can notably save power, KV stores typically receive high arrival rates and short service times (e.g., 6 to 33 μs [8, 47]), which limit their effectiveness [8, 36]. This is because idle periods are fragmented into very short idle cycles, and hence CPUs cannot enter a deep enough sleep state (e.g., C6) to be effective. To show this phenomenon, we run a Memcached server with 10 worker threads and measure the c-state residency time of CPU cores under different request rates (see Section 4 for experimental setup details). Figure 1 shows the fraction of time spent in different c-states. As shown, CPU cores cannot go to deep idle states (e.g., C6) even when the request rate is as low as 100K (10K per each core) requests per second (RPS).

DVFS and Request delaying. Modern processors support multiple clock-frequency steps [12, 55]. One approach to save power is to exploit the latency slack (difference between the observed and the target tail latency) to slow down request processing by adjusting the CPU frequency. At every request arrival, when enough slack observed, the CPU frequency is decreased to save power while meeting tail latency constraints [27]. Although this approach is effective for requests with large service times (> 250μs) [29], the variable and short service times of KV stores do not provide enough opportunity to save power using this approach. This is because the service time of KV store requests cannot be extended enough to fill the latency slack due to limited DVFS frequency ranges.

Another approach is to exploit latency slack to delay request processing [8]. Request processing is completed just-in-time (before missing the deadline). This approach, in fact, consolidates idle periods to meet the target residency time of deep idle states (e.g., C6) more frequently. For this approach to be effective, the difference between the target tail latency and the tail response time (tail service time + tail queueing time) must be greater than the residency time of deep idle states (600μs for Intel Xeon 6130 CPUs), making this approach effective only when the latency constraints are more relaxed (e.g., $> 700\mu s$).

Feedback-based controllers. Feedback-based controllers periodically measure the incoming rate or the observed latency, compare them to predefined thresholds, and adjust the number of cores allocated to the KV store application [36, 64].

These solutions are straightforward, but they are unresponsive to short-term variability, leading to latency performance degradation [27].

Moreover, if they adjust the number of allocated cores in response to load variations, they will rely on the kernel scheduler to handle the KV store threads once the number of allocated CPU cores is adjusted. OS schedulers, however, are not designed for scheduling at the granularity of microseconds as time quanta are typically on the order of milliseconds [6, 42, 47]. Hence, they notably aggravate tail latency. Figure 2 shows the impact of the OS scheduler on tail latency. For this experiment, we run Memcached when its threads (workers) are pinned to distinct CPU cores (the impact of OS scheduler is eliminated) compared to when KV store threads are not pinned (OS is in charge of scheduling). As shown, the OS scheduler increases the tail latency by up to 230%, making solutions that rely on the OS scheduler less appealing in the presence of tight microsecond-scale tail latency goals. Several research works [31, 47], in fact, suggest pinning each KV store thread to a distinct core to eliminate the impact of kernel schedulers on tail latency.

3 PEAFOWL DESIGN AND IMPLEMENTATION

We introduce Peafowl, an elastic in-memory KV store to reduce power consumption while maintaining microsecond-scale tail latency. Peafowl achieves this by logically transferring the scheduling from the OS to the application. Peafowl addresses the requirements outlined in Section 2.1 as follows:

1) Energy proportionality. Peafowl's scheduler leverages in-application knowledge to characterize the current and peak load of the KV store. It carefully consolidates the load among fewer CPU cores to reduce the number of active cores during off-peak periods. Peafowl triggers the OS idle-state (c-state) governor to save power for inactive cores. This translates to notable power saving during low or medium periods of utilization.

2) Keeping latency at microsecond scale. Peafowl introduces the idea of performing scheduling within the KV store application to maintain microsecond-scale tail latencies. By managing the scheduling within the application, Peafowl eliminates the millisecond-scale tail latency impact of OS scheduling. Furthermore, in-application scheduling allows Peafowl to monitor load in fine-grained intervals and immediately distribute load when a core is overloaded to avoid worsening tail latency.

3) Generality and ease of deployment. Peafowl works with the Memcached protocol and is entirely implemented in user space. It can meet tail latency goals ranging from microseconds to milliseconds. Peafowl does not rely on third party monitoring services or feedback-based controllers. It does not require any OS modification or privileged components. It does not assume any hardware feature other than the c-state mechanism that is supported by almost all processors currently adopted in data centers, making Peafowl production-ready.

3.1 Peafowl Architecture

Figure 3 shows the architecture of Peafowl. Peafowl introduces the idea of transferring the scheduling from the OS to the application where there is more domain-specific knowledge and control. This is accomplished by creating a worker thread for all but one CPU core and pinning each worker thread to a different core. The last remaining core has a scheduler/management thread pinned to it for performing all the in-application scheduling. We dedicate a core to the scheduler to avoid potential interference, but as it is not on the critical path, we believe this may not be necessary.

Scheduling every single KV request (e.g., get, set, update) would introduce a significant scalability bottleneck, so the key idea to make Peafowl scalable is to treat connections as schedulable entities. Connections represent clients (e.g., web applications) that send a multitude of KV requests over the lifespan of the connection. Peafowl schedules connections by assigning them a current worker and a home worker. If a connection is currently served by its home worker, we call it a *resident connection*. If it is served by another worker, we call it a *guest connection*. Mirroring the behavior of Memcached, Peafowl dispatches newly created connections among workers in a round-robin fashion. It assigns this initial worker as the connection's home worker. Over time, Peafowl migrates connections between workers to consolidate connections onto a fewer number of active CPU cores during low load periods. Peafowl triggers the OS c-state governor to put the remaining inactive CPU cores in a deep-idle state, thereby saving power.

By performing scheduling within the application, Peafowl is able to take advantage of application-specific knowledge. For example, each worker monitors its load and communicates this to the scheduler. Peafowl also learns the peak load of the incoming workload to establish a baseline performance goal. That is, Peafowl ensures that the performance during low power periods with a few active CPU cores matches the performance during peak loads with all CPU cores active. Additionally, workers are able to independently respond to

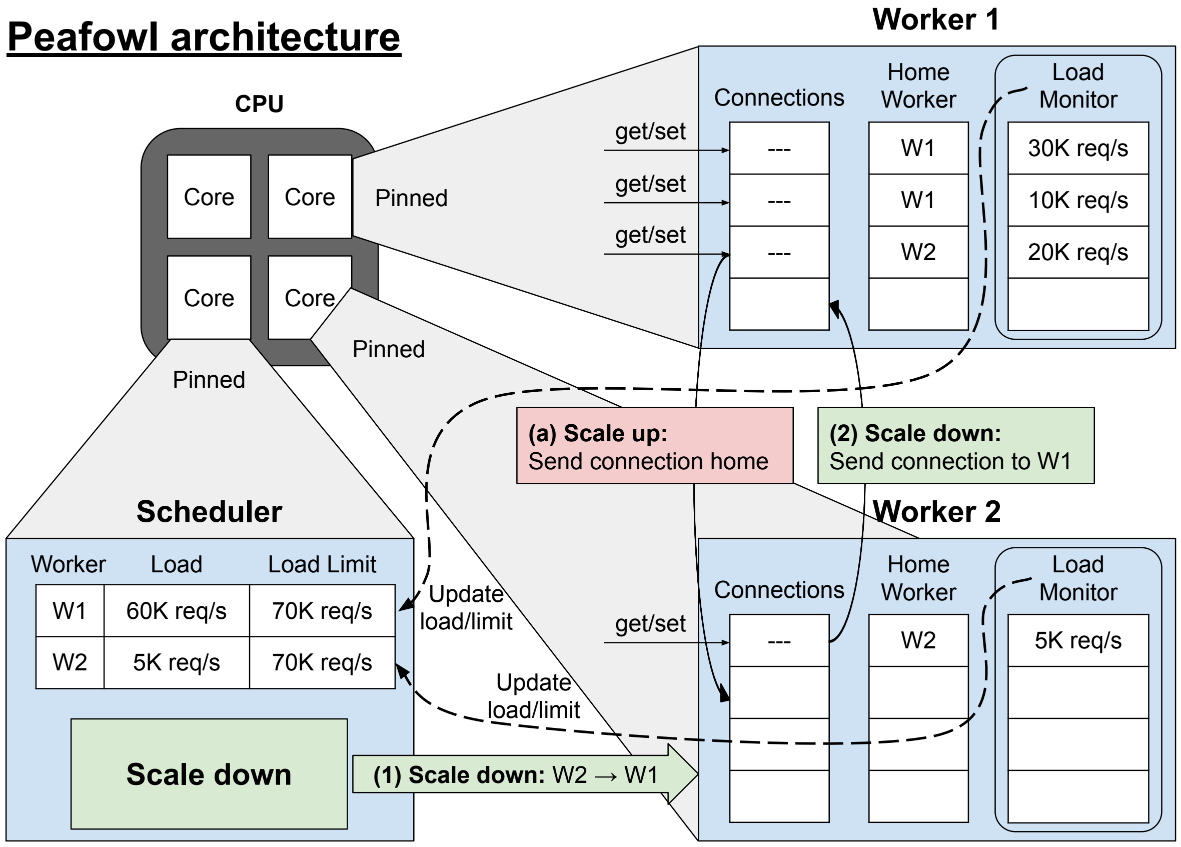

Figure 3: The architecture of Peafowl. Peafowl creates and pins one thread per CPU core. One thread functions as the scheduler. Other threads function as workers, monitoring user connections and accommodating their requests (e.g., get, set, update). Workers also monitor the load for their connections and periodically update the scheduler with their load. The scale down process is managed by the scheduler. (1) The scheduler instructs a worker (e.g., worker 2) to transfer a connection to another destination worker. (2) The worker sends a connection instance to the other worker (e.g., worker 1) using shared memory. The scale up process is not coordinated with the scheduler. (a) If a worker (e.g., worker 1) detects too high of a load, it sends a guest connection back to its home worker (e.g., worker 2).

load increases immediately without coordination with the scheduler by transferring guest connections back to their home worker. This conservative approach allows Peafowl to maintain low tail latencies on par with the performance using all CPU cores.

3.2 Energy-proportional Scheduling

Peafowl's scheduling policy consists of two parts: scale-down and scale-up. When load decreases, Peafowl's scale-down process will consolidate connections onto fewer CPU cores and put inactive cores in a deep idle state to save power. When load increases, Peafowl's scale-up process will expand to using more cores to avoid increasing tail latency. This is similar to autoscaling (e.g., AutoScale [15]), but in the context of CPU cores within a server rather than across servers. Another key difference between Peafowl and autoscaling web systems is that Peafowl is designed for KV store workloads with high request rates. This section describes how Peafowl accomplishes this with low overhead on the critical path.

3.2.1 Scale-Down Process. Peafowl scheduler is responsible for coordinating the scale-down process. Each worker corresponds to a CPU core. Workers periodically report their current load to the scheduler. When the scheduler sees that the total load can be consolidated onto fewer workers (i.e., cores), it initiates the scale-down process.

Peafowl uses a greedy algorithm for determining which worker to scale down. First, it marks the worker with the lowest load as the *scale-down worker*. Since it has the lowest load, it is easier to pack its connections among the other workers. Second, the scheduler selects among the remaining active workers (excluding the scale-down worker) the one with the lowest load as the *destination worker*. The scheduler signals the scale-down worker to transfer one connection to the destination worker. Upon completion, the scheduler will repeat the process with a potentially different destination worker to distribute the scale-down worker's connections across the remaining active workers. By gradually moving connections one at a time, we avoid drastic load changes at a worker. Peafowl is designed to be conservative to ensure low tail latency.

Once all the scale-down worker's connections are migrated, Peafowl enables the idle-states on the scale-down worker's core to put it into deep idle states, thereby saving power. We only enable the idle-states for inactive cores and disable the idle-states for active cores to avoid expensive idle-state wake-ups as reported in [31, 47].

3.2.2 Scale-up Process . When a worker's load approaches its load limit (learned by Peafowl's monitoring system in Section 3.3), the worker starts migrating some guest connections back to their home worker. As the home worker is predetermined, workers do not coordinate with the scheduler for performing the scale-up migration, leading to immediate scale-up. They independently send guest connections back home as soon as load is too high, thereby avoiding performance degradation.

To avoid the scale-up and scale-down process from competing with each other, Peafowl takes a conservative approach and allows the scale-up process to block the scale-down process. In the final step of the scale-up process, both the current and home worker signal the scheduler to avoid considering them as candidates for the scale-down or destination worker for a period of time. We find that Peafowl is not sensitive to the specific period of time, and we use 30 seconds as a conservative duration for blocking transfers to prevent excessive connection transfers during micro-bursts.

3.2.3 New Connections. Peafowl assigns newly created connections to workers in a round robin fashion, mirroring the behavior of Memcached. This initial worker is deemed the home worker of the connection, which identifies how to rearrange connections during the scale-up process. If the home worker is currently active, then the new connection would be sent to this worker.

However, if the home worker is currently inactive due to the scale-down process, Peafowl will send the new connection to least loaded active worker.

3.3 Monitoring

Peafowl monitors two key pieces of information: each connection's current load and the worker's load limit. Both of these are application-specific metrics, which necessitates Peafowl's in-application scheduling.

A connection's load is measured in terms of its request rate, which is calculated based on the amount of time in which a connection sends a fixed number of requests, n. That is, the request rate is calculated as n divided by the time period in which a user sends n requests in a connection. Thus, monitoring the load only introduces a request counter and timestamp information for each connection, which adds minimal request processing and memory overhead. The worker periodically sends load and load limit updates to the scheduler so that it can make the appropriate scheduling decisions for which workers should be consolidated at what point in time.

The load limit plays an important role in Peafowl's scheduling decisions for determining when it is possible to pack connections onto fewer active CPU cores. Rather than having users specify a load limit, Peafowl automatically learns the load limit based on the traffic it receives. The key insight is that we can use the default behavior without Peafowl as a baseline for acceptable load conditions. Without Peafowl, connections would all be served by the home worker, which was determined in a round robin fashion as per the default Memcached behavior. Thus, summing the load of the resident connections (i.e., connections served by the home worker) calculates the load of the original KV store without Peafowl. The load limit is then calculated by tracking the maximum total load of the worker's resident connections over time.

Using the maximum load across time could lead to an unrealistically high load limit due to outliers in the load measurement. To guard against this, Peafowl applies the Z-score technique for filtering out outliers [44]. The Z-score quantifies measurements in terms of number of standard deviations above/below the mean. That is, $Z-score = \frac{measurement-mean}{stddev}$. We quantify outliers as measurements with a Z-score greater than 2.5, which corresponds to measurements that are more than 2.5 standard deviations above the mean. To calculate the mean and standard deviation, we use an online algorithm [61], making our outlier detection algorithm O(1). In addition to the outlier filtering, Peafowl also provides a mechanism for users to manually reset the load limit to handle unusually high load periods that are not reflective of normal expected behavior.

3.4 Connection Transfer Implementation

Peafowl extends the current Memcached code base, which uses an event-driven architecture based on `Libevent`. In this architecture, `Libevent` monitors file descriptors (e.g.,

Algorithm 1: Worker

```
Function process_request(connection c):
    if c is a user connection then
        process_memcached_request(c)
        if connection_transfer== true then
            connection_transfer = false
            unregister_libevent(c)
            send c to destination_worker
            active_connections -= 1
            if active_connections == 0 then
                turn on idle-state mechanism
            end
        end
        if worker_load > load_limit and c is a guest connection then
            unregister_libevent(c)
            send c to home_worker(c)
            active_connections -= 1
            if active_connections == 0 then
                turn on idle-state mechanism
            end
            signal the scheduler
        end
    else
        // c is a communication channel
        cmd = extract message from c
        if cmd == "receive connection" then
            new_conn = receive connection
            register_libevent(new_conn)
            if active_connections == 0 then
                turn off idle-state mechanism
            end
            active_connections += 1
            signal the scheduler
        else if cmd == "transfer connection" then
            extract destination_worker from c
            connection_transfer = true
        end
    end
```

user connections) and waits for requests (e.g., get, set) from any of its connections. Peafowl builds on top of this by creating its communication channels using Linux pipes, which are file descriptors, and registering them with `Libevent`. Thus, Peafowl works seamlessly with Memcached without introducing additional synchronization mechanisms that may cause bottlenecks.

When requests or internal communication messages arrive, `Libevent` invokes a function, `process_request`, to process the requests/messages (Algorithm 1). The function first checks if a message is from a user connection. If so, it performs the user request (e.g., set, get). Then if the worker has been instructed by the scheduler to transfer a connection (scale-down) or if the worker's load is too high (scale-up), it unregisters the current connection and sends it to the destination worker (scale-down) or home worker (scale-up).

If a worker transfers all of its connections, it performs a system call to enable idle-states on its corresponding CPU.

When a worker receives a connection (due to scale-down, scale-up, or new connections), it registers the connection with `Libevent` and disables idle-states on its core if needed.

The overhead of transferring a connection is minimal since it only involves unregistering a connection from one worker's libevent and registering the connection on the destination worker's libevent. Our experiments show that the overhead is less than 30μs.

4 EVALUATION

We implement Peafowl in C as an extension of Memcached. Our evaluation of Peafowl addresses the following questions:

- How well does Peafowl perform when facing real-world workload patterns? (Section 4.1)
- How do Peafowl's internal mechanisms contribute to its observed performance? (Section 4.2)
- How does Peafowl compare to the state-of-the-art approaches? (Section 4.3)

Experimental setup We use one dual-socket server machine equipped with 12-core Intel Xeon 4116 CPUs running at 2.1 GHz and 32 GB of RAM. We generate load using one dual socket client machine equipped with 32-core Intel Xeon CPUs running at 2.6 GHz and 64 GB of RAM. These machines are configured with Intel x520 10GbE NICs. The server machine hosts Peafowl and Memcached. The client machine runs a modified version of Mutilate [30] as a load generator. We use the Ubuntu 18.04 distribution running Linux kernel version 4.11. To measure power consumption, we query MSR registers through the RAPL interface [36, 46].

4.1 Real-world Scenarios

In this section, we evaluate the effectiveness of Peafowl for real-world traces. We modify Mutilate to recreate three different workloads: (1) the ETC workload from Facebook [4], which exhibits a diurnal pattern, (2) a storage workload trace from Microsoft [18], which has a diurnal pattern over a wider range, and (3) a Google search trace reported in [36], which features sudden load shifts. We use 512 persistent Mutilate connections to generate load. Similar to the USR workload

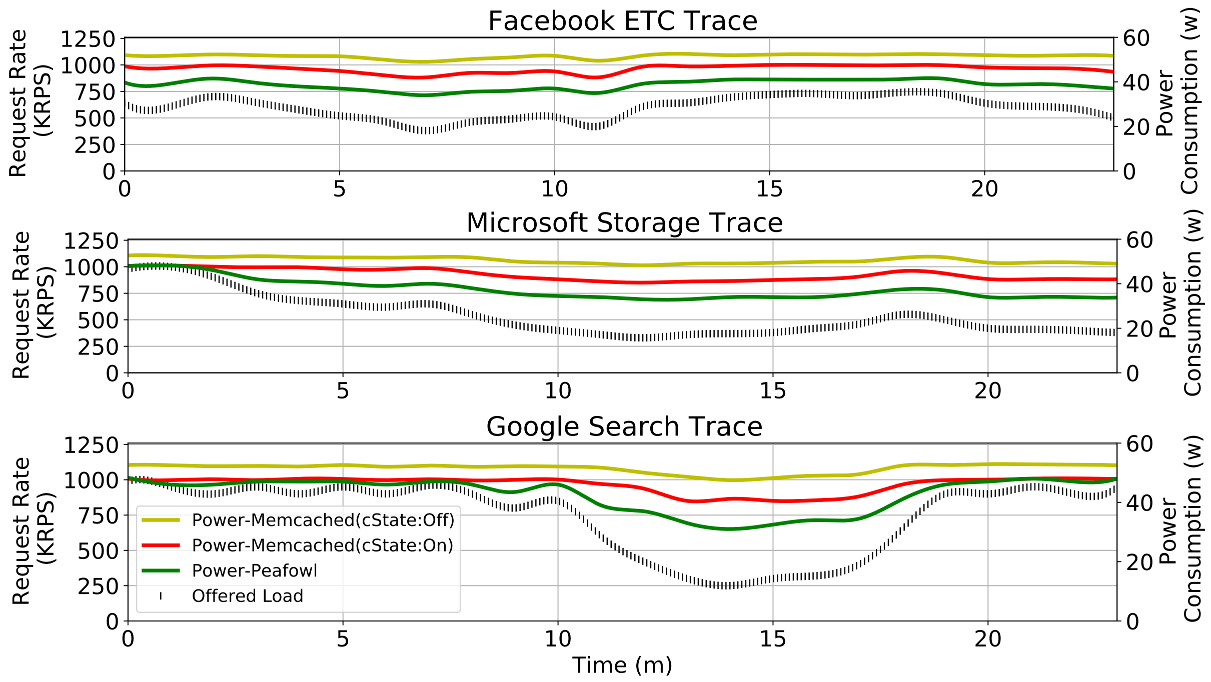

Figure 4: Power consumption of Memcached and Peafowl when facing real-world scenarios. Peafowl delivers higher energy-proportionality. When the load drops, the power consumption decreases, accordingly.

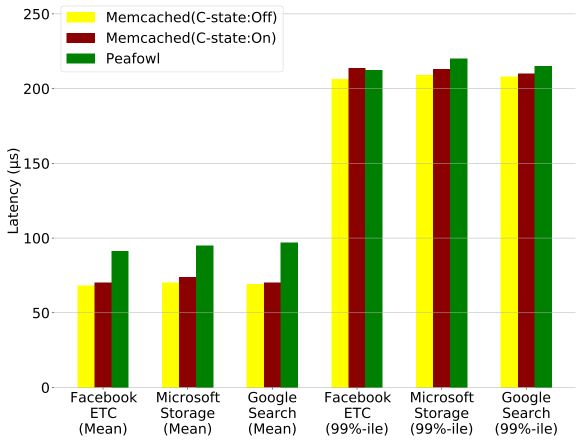

Figure 5: Average and tail latency of requests under Peafowl and Memcached. Both deliver similar tail latency; Memcached delivers lower average latency because unlike Peafowl, it always operates all workers.

reported by Facebook [4], requests follow a Poisson arrival process, consisting of 99.8% GET requests and 0.2% SET requests, where value sizes follow Generalized Pareto distribution. We run both Memcached and Peafowl with 10 worker threads. As noted before, Peafowl pins workers to distinct CPU cores as it employs in-application scheduling. To ensure a fair comparison, we pin Memcached threads to distinct CPU cores.

Figure 4 shows the power consumption of (1) Peafowl, (2) Memcached when idle-states are enabled, and (3) Memcached when the idle-states are disabled. Compared to Memcached when idle-states are enabled, Peafowl reduces the power consumption by up to 36%.

This is because Peafowl identifies off-peak periods and consolidates the load onto fewer active cores. In the Google trace, for example, the load at first is 1000K requests per second (RPS). At this point, Peafowl operates all 10 workers. At around minute 15, the load drops to 230KRPS. At this point, Peafowl operates with three workers. Meanwhile, the corresponding CPU cores of the remaining seven workers save power by mostly being in C6, the deepest idle state. In Section 4.3, we will elaborate on why c-state governors such as Menu are not effective in KV stores.

Figure 5 shows the observed average and 99^{th} percentile tail latency for all scenarios. While Memcached and Peafowl deliver the same tail performance, Memcached yields a lower average latency. This is because at any given time Memcached operates on greater or equal number of CPU cores compared to Peafowl.

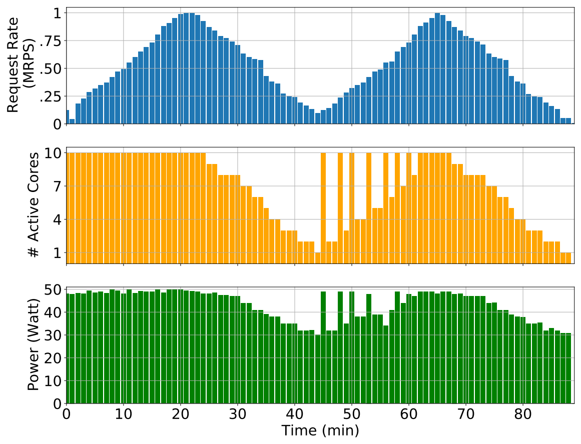

Figure 6: Scale-up and scale-down process in Peafowl. The number of active workers/cores corresponds to the current load. The scale down process achieves energy proportionality. The scale up process is conservative as it reactivates all cores for a short time when the load increases to ensure that tail latency is low.

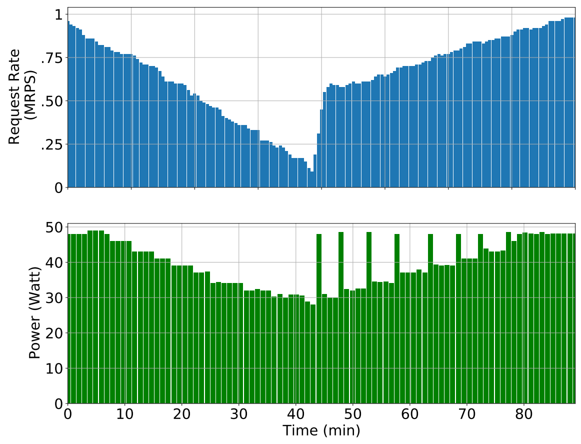

Figure 7: When an extreme load shift from 50KRPS to 650KRPS happens, Peafowl operates all cores to avoid severe tail latency degradation.

4.2 Microbenchmarks

This section evaluates Peafowl's internal mechanisms that are key to its performance. We use the Mutilate benchmark to generate a synthetic workload, shown in Figure 6. As shown, the load sent to the KV store starts from 50KRPS, gradually increases to one million RPS, gradually decreases back to 50KRPS, and then repeats the process again. This load pattern shows how Peafowl handles the scale-up and scale-down process, and also illustrates how Peafowl learns the load limit.

Scale-down process. Figure 6 also shows how the number of active cores changes when the load varies. Initially, Peafowl starts with 10 workers and assumes that the starting request rate is the load limit. However, as the load increases, Peafowl learns that supporting up to one million RPS is an acceptable peak load for all 10 workers, which corresponds to a peak load of 100KRPS for each worker. Although Peafowl does not save power during this initial period when learning the load limit, this is only a one-time effect.

Then when the load drops between 20-45min, Peafowl's scheduler packs the load into a reduced number of workers. For example, when the load is under 100KRPS (1/10 of the peak load), Peafowl operates one worker (1/10th of initial active workers).

Scale-up process. Once the experiment gets to the point where only one worker is operating, the experiment starts increasing the load. At this time, the load of the only operating worker approaches its load limit. Consequently, it starts sending guest connections to their home workers, which leads to the situation when all workers are operating again. At this point, Peafowl waits for a window of time for things to settle. It then starts consolidating the load (200KRPS), which fits on two workers. As the load increases in the workload, the scale-up and scale-down process repeats. Although our scale-up process is not as energy proportional as the scale-down process, this conservative approach is important for avoiding tail latency degradation during sudden bursts.

Sudden bursts. To show how Peafowl performs when sudden bursts happen, we use a synthetic workload shown in Figure 7. We offer a baseline load of one million RPS. The load gradually decreases to 50KRPS, where Peafowl consolidates the load onto one worker. This is followed by an instantaneous increase to 650KRPS. When the sudden burst occurs, all workers immediately become activated. In this experiment, the average latency is 110μs and the 99^{th} percentile tail latency is 250μs, which is in roughly the same microsecond scale range as a baseline Memcached setup (with 78μs average latency and 210μs tail latency). Thus, Peafowl reacts quickly so that it is able to maintain micro-second scale tail latency, even when handling an extreme load increase from 50KRPS to 650KRPS.

4.3 Peafowl compared to state-of-the-art

Lack of hardware control prevents approaches like Rubik and μDPM to be faithfully implemented and adopted in off-the-shelf servers. For example, μDPM requires offloading the prediction calculations and per-request scheduling to smartNICs. Therefore, they report simulation results. In this section, we simulate Rubik, μDPM, an optimal idle-state governor with oracle foreknowledge, and Peafowl. We develop a discrete event queueing simulator where we implement

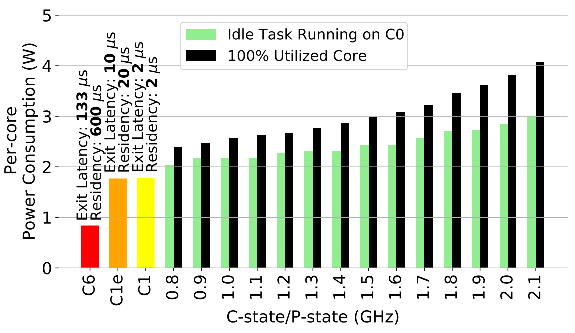

Figure 8: Idle-states available in our testbed and their respective exit latency, target residency, and per-core p-state power consumption when the core is in C0 and when the core is executing a CPU-intensive task.

Peafowl and the state-of-the-art policies. In our simulation, we use the common queueing arrival process known as a Poisson process (i.e., exponential distribution interarrival time) and vary the request rate over time. Service times are modeled by an exponential distribution with an average service time of $10\mu s$[1]. To model the c-state transitions, we set the transition latency, the target residency, and the power consumption according to Figure 8, which matches our testbed. We simulate DVFS with a CPU frequency range from 0.8 to 2.1GHz and DVFS transition latency of $10\mu s$. We also measure DVFS power consumption for each frequency in our server and use these parameters in our simulation. We simulate power consumption by quantifying the weighted sum of idle and busy periods with the corresponding energy consumption, factoring in both the c-states and DVFS as appropriate for each system.

4.3.1 *Evaluated Systems.*

- **Rubik** [27] saves power by exploiting existing latency slack and slowing down request processing through DVFS. Rubik adjusts frequency upon arrival and completion of each request. Their simulation results indicate that Rubik has better power consumption and tail latency than approaches relying on feedback-based controllers (e.g., Carb [64] and Pegasus [36]) since Rubik is responsive enough to cope with short term variations.
- **μDPM** [8] extends idle period lengths by consolidating idle periods through delayed request processing. That is, it exploits latency slack by delaying the processing of requests to let the CPU be idle for longer time periods while executing requests just in time. Their simulation results illustrate that their technique is more energy efficient than existing techniques such as Rubik [27], SleepScale [35] and DynSleep [9].
- **Idle-state Governors** (e.g., Menu [60] and recently TEO [52]) combine several factors (e.g., device and timer interrupts) as a heuristic to predict the upcoming idle duration and consequently select the idle mode that fits best. Studies [22, 52, 54] show that Menu frequently makes inaccurate predictions, missing the opportunity to save power or hurting latency. As an upper bound on the idle-state governor approach, we simulate a Clairvoyant Idle-State Governor (CISG) that precisely predicts the future events and always chooses the correct c-state. Since network events cannot be determined in advance, clairvoyant idle-state governing is impractical. However, our goal is to show that even an optimal clairvoyant idle state governor does not work well for KV stores because of the high request rates. Rather, techniques such as Peafowl that shifts around load is needed.

To evaluate these approaches, we initially start with an 800KRPS request rate and gradually decrease the rate to 80KRPS in 80KRPS steps. We measure the power consumption and 99^{th} percentile tail latency at each step.

Figure 9 shows the power consumption and Figure 10 shows the observed tail latency of different systems when the target latency ranged from $300\mu s$ (stringent target latency) to $900\mu s$ (relaxed target latency). Peafowl saves significantly more power than Rubik, μDPM, and CISG.

As pointed out in the μDPM paper, Rubik suffers from frequent DVFS transitions especially at higher request rates, which affect both power and latency. Additionally, Rubik does not effectively utilize c-states in KV store workloads since the frequent request arrivals prevent it from spending significant time in the deepest c-state (C6). μDPM addresses these problems by avoiding frequent DVFS transitions and immediately transitioning to C6 when idle. μDPM maximizes the idle time while ensuring predicted latencies are below the target latency. However, since C6 has a long exit latency, μDPM does not significantly save power under tight target latencies (e.g., $300\mu s$). By contrast, Peafowl is able to pack connections onto fewer cores to allow some cores to remain in C6 for extended periods of time, saving significant power[2].

Both μDPM and Rubik rely upon latency predictions, which are not always perfect. In some cases, this results in tail latencies exceeding the target latency. In other cases where μDPM overpredicts latency estimates, it results in lower tail latencies and lower power savings (see higher request rates

[1]To quantify the average service time, as suggested in [20], we increase the request rate until the completion rate levels off at some value. At this point, the average service time is equal to 1 / completion_rate.

[2]Peafowl's power consumption in Figure 9 is linear due to the resolution of sampling, but is expected to have a more staircase-like pattern as cores are put to sleep.

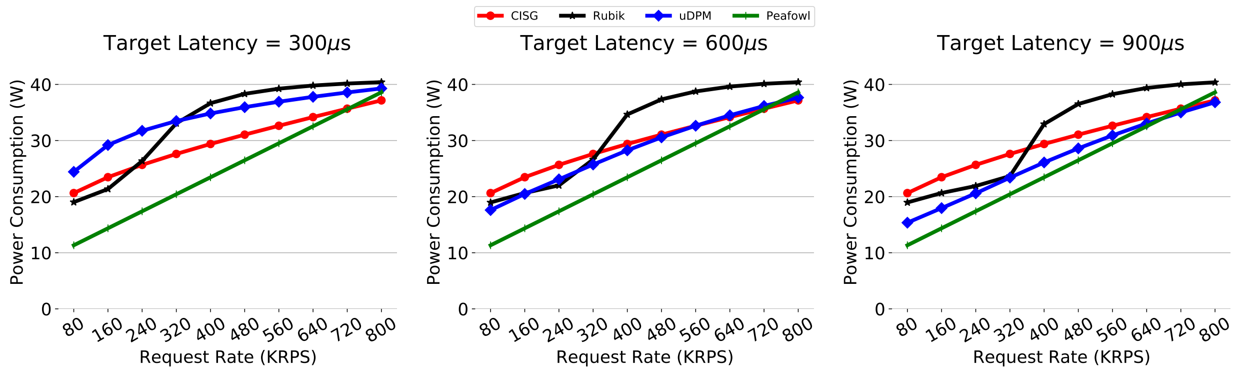

Figure 9: Power consumption of state-of-the-art systems under different target latencies. Peafowl saves more power than prior approaches by identifying off-peak periods and right-sizing the number of active workers.

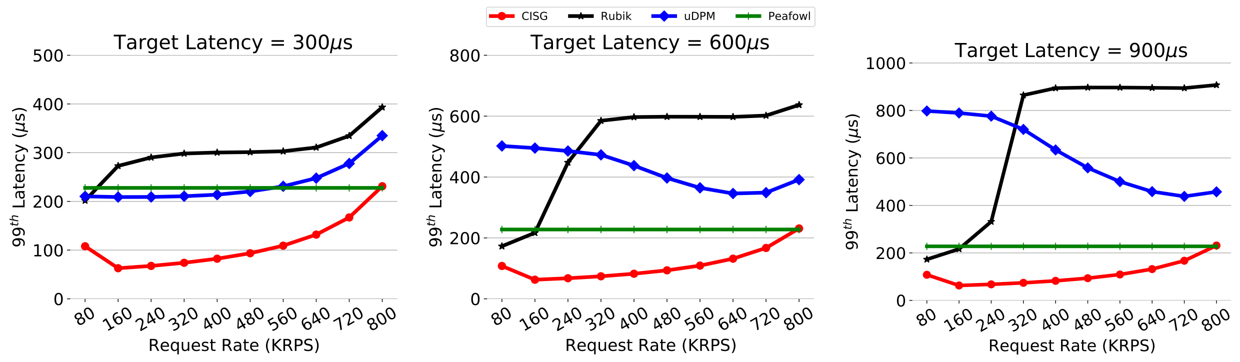

Figure 10: 99th percentile tail latency comparison of state-of-the-art systems under different target latencies. Mispredictions in μDPM and Rubik cause high tail latency at high request rates.

under Figure 10's 600μs and 900μs target latency graphs). By contrast, Peafowl is not explicitly designed for latency targets and achieves power savings even without delaying or slowing down requests. It is possible to adjust the DVFS settings in conjunction with Peafowl to save additional power (at the expense of additional latency), and we leave this to future work (Section 6).

As demonstrated in Figure 1, high arrival rates of KV store workloads fragment the idle periods, resulting in short idle cycles. This makes idle-state governors ineffective since they cannot find enough opportunities to go to deep sleep states. Our simulation results for CISG demonstrate that even the optimal clairvoyant idle-state governor is not able to significantly save power for KV stores due to the short service times and high arrival rates that are common in KV stores.

Peafowl consolidates the workload to fewer cores when the load drops and saves power by letting the idle cores go to deep idle states. As shown in Figure 9, when the load drops to 80KRPS, Peafowl saves 40%, 54%, and 45% over Rubik, μDPM and CISG respectively.

5 RELATED WORK

Feedback based approaches. Pegasus [36] and Carb [64] leverage feedback-based controllers for saving power. Pegasus periodically measures tail latency and adjusts the frequency every few seconds to avoid tail latency violations. Pegasus adapts to long-term variations, but is unresponsive to short term variability. Carb is a feedback-based controller that determines the number of operating cores allocated to the KV store, turning off inactive cores. Even if approaches like Carb operate at zero overhead, they rely on the OS scheduler to handle KV store threads. As shown in Figure 2 and described in multiple papers, OS schedulers can significantly impact tail latency [31, 33, 47]. Peafowl does not rely on slow feedback-driven controllers. It is responsive to sudden load changes as it monitors the load inside the KV store. Further, Peafowl worker threads are pinned to distinct cores to reduce the OS impact on tail latency, and Peafowl's in-application scheduler takes the job of managing the load among the worker threads.

DVFS and request delaying. In systems with latency goals, multiple existing techniques exploit the latency slack (i.e., latency goal minus current latency) to reduce power by slowing down request processing through DVFS [8, 21, 27–29, 35, 36, 59], or delaying request processing through sleep states [8, 9, 38, 39] while meeting tail latency constraints. Adrenaline [21] boosts long requests which are more likely to contribute to the tail latency. It senses the load at minute-scale intervals, which makes it unable to respond to sudden load changes. Rubik [27] outperforms the existing DVFS-based approaches such as Adrenaline [21], Pegasus [36] and TimeTrader [59] by using fast, per-core DVFS that quickly adapts to short-term variability. Upon arrival and completion of each request, it adjusts the frequency when there is a gap between the target latency and the observed latency. However, the limited frequency range of DVFS does not allow Rubik to slow down request processing enough to fill latency slack in low load scenarios, leading to suboptimal power savings, especially when the target tail latency is relaxed. Moreover, Rubik's aggressive state transition at every request arrival is costly in terms of energy and transition latency [8]. DVFS based approaches are effective only when the request execution time is long enough (e.g., $> 250\mu s$) [29].

μDPM outperforms approaches that are based on deep sleep states such as PowerNap [38], DreamWeaver [39], and DynSleep [9] and also DVFS-based approaches such as Rubik by leveraging a hybrid approach that uses both sleep states and DVFS mechanisms. μDPM augments Rubik with a request delaying approach. If there is enough latency slack, μDPM delays the request processing to enable idle periods where deep sleep states can be utilized. μDPM, however, is effective only when the target latency is relaxed and the arrival rate is low. Otherwise, it is unable to extend idle periods enough to save power. Furthermore, this approach requires heavy computations to estimate the response time and scheduling of each request. By contrast, Peafowl only needs to monitor the load and pack the load onto fewer CPU cores when the load drops.

In-application Scheduling. Several works demonstrate that data-centers can benefit from more focus on per-node efficiency as opposed to the more common focus on scalability to a large number of nodes [2, 3, 7, 42, 48, 49, 63]. Rhoden et al. [49] believe that traditional OSs are ill-suited for data center applications. They propose Akaros, an OS built on the philosophy of transparency, where the OS exposes as much information (e.g., CPU core information) as possible to data center applications and giving them APIs to enable application-level management to achieve more efficiency [16]. Following this path, several systems have been

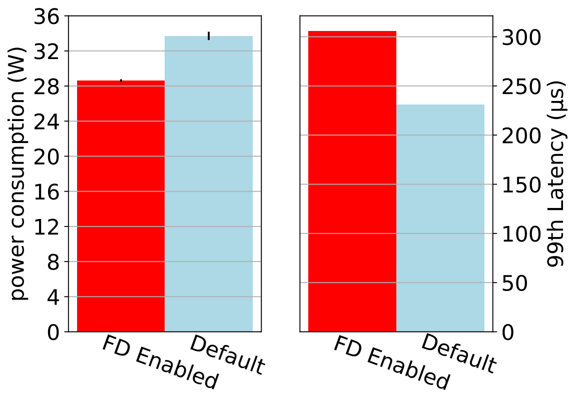

Figure 11: Potential reduction of power consumption by steering the NIC interrupts to active CPU cores. This comes at the cost of higher request tail latency.

designed where applications can contribute to resource management. Arachne [47], for example, uses a central arbiter to allocate cores between applications, and applications control the placement of their threads on those cores, which can raise the throughput of thread-driven applications with extremely short-lived threads. In a similar spirit, Shenango [42] raises CPU efficiency while delivering high throughput by using high-performance network stacks and a privileged component called IOKernel that reallocates cores across applications at fine time scales. ZygOS [45] achieves a notable throughput improvement by using kernel-bypass networking and adding work stealing to improve load balancing within an application. While these works are special-purpose mechanisms to deliver high throughput and low latency by leveraging thread-driven architectures, third party runtimes and high-performance network stacks, Peafowl introduces an application-level scheduler to save power in event driven in-memory KV stores while not relying on modified operating systems, privileged components, third-party runtimes, or special hardware.

6 DISCUSSION AND FUTURE WORK

Receive-Side Scaling (RSS). Peafowl saves power by decreasing the number of active CPU cores when load drops. However, inactive cores are not completely idle as Receive-Side Scaling (RSS) in the network interface still distributes the incoming traffic between all CPU cores. The general issue with RSS, apart from its inability to evenly distribute flows with variable load characteristics [5, 51], is its blindness to OS thread scheduling [50]. Even with RSS, Peafowl is able to significantly save power, but even greater power savings can be achieved through finer-grained control over RSS. Specifically, Intel's Flow Director (FD) can be used to provide the NIC with flow classification rules using the application knowledge for directing network interrupts to the appropriate cores. We built a prototype of Peafowl with FD to pack the interrupts only into active cores. However on the packet transmission side, there needs to be a similar interface limiting the number of cores responsible for packet transmission. Otherwise, the TX interrupts will still wake up the inactive cores, causing unnecessary idle-state wake-ups. As there is no current user-space interface for controlling the transmission interrupts, we defer implementing this enhancement to a future work.

Figure 11 shows the achievable power reductions when using FD on the receive side and a similar flow classification technology on the transmission side ($\approx$ 18%). In this preliminary experiment, we test a low load scenario where Peafowl consolidates all the connections on a single core. At this point, we measure the tail latency and power consumption when RSS distributes the NIC interrupts among all cores, compared to when NIC interrupts are routed into active cores only. Figure 11 shows that the lower power consumption comes at the cost of an increase in the tail latency due to only having a single core for packet processing on both the RX and TX sides, a tradeoff that must be decided according to the performance requirements. In addition, we note that in the future implementations of Peafowl that use kernel-bypass network stacks, hardware flow classification would be unnecessary as poll-mode drivers typically enable a single NIC receive queue [42].

Memcached Load Imbalance. Load imbalance in Memcached is a well known problem where some workers may receive a lot more traffic due to connections with high request rates. Specifically, Memcached's round-robin allocation of connections to workers can lead to persistent imbalance under skewed workloads, with some workers hosting fat connections (connections with high arrival rates) while other workers are serving thin/idle connections [45]. This imbalance can impact both latency and throughput [47]. With in-application scheduling, it is possible to leverage Peafowl's knowledge of connections rates to relocate connections so as to reduce persistent imbalance among workers, which we leave to future work.

Latency Targets. Peafowl is designed for ease of deployment where users do not need to provide latency targets and there aren't complex parameters to tune. Peafowl learns the peak load and matches the performance of a native Memcached deployment at peak load. That is, we assume operators have provisioned the system to have acceptable performance at peak load and use this as our baseline performance target rather than having users specify targets. Section 3.3 discusses outlier detection to handle spikes in the peak load learning.

For comparing with Rubik and μDPM, we introduce latency targets in the evaluation, but Peafowl doesn't use this information as it learns the performance goal based on the system load. It is possible to adapt Peafowl to handle latency targets by adjusting the DVFS configuration to meet the latency target at peak load, which we leave to future work. If anything, this would result in Peafowl achieving even greater power savings than demonstrated in our results.

Limitations and Assumptions. Peafowl is designed for interactive services and is most beneficial for workloads that serve high request rates (e.g., KV stores). Our implementation can generalize to libevent-based multi-threaded applications, but we do not think our implementation will apply to other non-event based application architectures. However, the core ideas in Peafowl (e.g., in-application scheduling, packing connections together to allow cores to go to deep idle states) are generally applicable to saving power. We assume workloads are run on dedicated servers where we have control over the power configuration settings. Applying our approach in multi-tenant scenarios (e.g., to use idle cores for other low priority tasks) is left to future work since Peafowl assumes that reclaiming resources is fast (10s-100s of microseconds).

7 CONCLUSION

This paper presents Peafowl, an event driven key-value store that saves power while keeping tail latency at microsecond scale. In Peafowl, the KV store itself is in charge of scheduling. The Peafowl scheduler monitors the load, learns the peak load, and when the load drops, it consolidates the load among a fewer number of CPU cores. The remaining idle worker threads turn on idle-states for their corresponding CPU cores to save power.

We compare Peafowl to state-of-the-art approaches and show that even with the high arrival rates and short service times of KV stores, there is a lot of room for power savings via Peafowl's approach. Compared to its closest competitors, Peafowl achieves up to 40-54% lower power consumption. Peafowl does not rely on any third party applications, runtimes, custom hardware, kernel modules, or any changes to the underlying OS, and therefore can be readily deployed on the top of off-the-shelf OSs and commodity hardware.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation awards (NSF CISE-1414119, CISE-1717858, and CISE-1931714). We thank our shepherd David Lo and the anonymous reviewers who provided excellent feedback. We are grateful to Vasiliki (Vasia) Kalavri, Vijay Chidambaram, and Manos Athanassoulis for their useful comments on earlier drafts of this work.

REFERENCES

[1] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*.

[2] E. Asyabi, S. SanaeeKohroudi, M. Sharifi, and A. Bestavros. 2018. TerrierTail: Mitigating Tail Latency of Cloud Virtual Machines. *IEEE Transactions on Parallel and Distributed Systems* 29, 10 (2018), 2346–2359.

[3] Esmail Asyabi, Erfan Sharafzadeh, SeyedAlireza SanaeeKohroudi, and Mohsen Sharifi. 2019. CTS: An operating system CPU scheduler to mitigate tail latency for latency-sensitive multi-threaded applications. *J. Parallel and Distrib. Comput.* 133 (2019), 232 – 243.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 12)* (2012).

[5] Tom Barbette, Georgios P Katsikas, Gerald Q Maguire, Jr., and Dejan Kostić. 2019. RSS++: Load and State-aware Receive Side Scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT '19)*.

[6] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[8] C. Chou, L. N. Bhuyan, and D. Wong. 2019. μDPM: Dynamic Power Management for the Microsecond Era. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[9] Chih-Hsun Chou, Daniel Wong, and Laxmi N. Bhuyan. 2016. DynSleep: Fine-grained Power Management for a Latency-Critical Data Center Application. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED '16)*.

[10] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013).

[11] Christina Delimitrou and Christos Kozyrakis. 2018. Amdahl's Law for Tail Latency. *Commun. ACM* 61, 8 (2018), 65–72.

[12] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. 2012. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*.

[13] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[14] L. Duan, D. Zhan, and J. Hohnerlein. 2015. Optimizing Cloud Data Center Energy Efficiency via Dynamic Prediction of CPU Idle Intervals. In *IEEE 8th International Conference on Cloud Computing*.

[15] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4 (2012), 14:1–14:26.

[16] Jana Giceva, Gerd Zellweger, Gustavo Alonso, and Timothy Roscoe. 2016. Customized OS support for data-processing. In *DaMoN '16*.

[17] Vishal Gupta, Paul Brett, David Koufaty, Dheeraj Reddy, Scott Hahn, Karsten Schwan, and Ganapati Srinivasa. 2012. The Forgotten 'Uncore': On the Energy-Efficiency of Heterogeneous Cores. In *USENIX Annual Technical Conference (USENIX ATC 12)*.

[18] U. U. Hafeez, M. Wajahat, and A. Gandhi. 2018. ElMem: Towards an Elastic Memcached System. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*.

[19] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[20] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press.

[21] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.

[22] Thomas Ilsche, Marcus Hähnel, Robert Schöne, Mario Bielert, and Daniel Hackenberg. 2018. Powernightmares: The Challenge of Efficiently Using Sleep States on Multi-core Systems. In *Euro-Par 2017: Parallel Processing Workshops*, Dora B. Heras, Luc Bougé, Gabriele Mencagli, Emmanuel Jeannot, Rizos Sakellariou, Rosa M. Badia, Jorge G. Barbosa, Laura Ricci, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer (Eds.).

[23] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

[24] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*.

[26] S. Kanev, K. Hazelwood, G. Wei, and D. Brooks. 2014. Tradeoffs between power management and tail latency in warehouse-scale applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*.

[27] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. 2015. Rubik: Fast analytical power management for latency-critical systems. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 15)*.

[28] David H. K. Kim, Connor Imes, and Henry Hoffmann. 2015. Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics. In *Proceedings of the 2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA '15)*.

[29] Mustafa Korkmaz, Martin Karsten, Kenneth Salem, and Semih Salihoglu. 2018. Workload-Aware CPU Performance Scaling for Transactional Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*.

[30] Jacob Leverich. 2014. Mutilate: High-Performance Memcached Load Generator.

[31] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*.

[32] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[33] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*.

[34] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[35] Yanpei Liu, Stark C. Draper, and Nam Sung Kim. 2014. SleepScale: Runtime Joint Speed Scaling and Sleep States Management for Power Efficient Data Centers. *SIGARCH Comput. Archit. News* 42, 3 (June 2014).

[36] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards Energy Proportionality for Large-scale Latency-critical Workloads. *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA 2014)* (2014).

[37] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. 2019. Snap: A Microkernel Approach to Host Networking *(SOSP '19)*.

[38] David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2009. PowerNap: Eliminating Server Idle Power. *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009).

[39] David Meisner and Thomas F. Wenisch. 2012. DreamWeaver: Architectural Support for Deep Sleep. *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 12)* (2012).

[40] Minh Nguyen, Zhongwei Li, Feng Duan, Hao Che, and Hong Jiang. 2016. The Tail at Scale: How to Predict It?. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

[41] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

[42] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[43] Chandandeep Singh Pabla. 2009. Completely Fair Scheduler. *Linux Journal* 2009, 184 (2009).

[44] R. Peck, C. Olsen, and J.L. Devore. 2011. *Introduction to Statistics and Data Analysis*. Cengage Learning.

[45] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 17)*.

[46] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. 2015. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*.

[47] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[48] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. 2011. TritonSort: A Balanced Large-scale Sorting System. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*.

[49] Barret Rhoden, Kevin Klues, David Zhu, and Eric Brewer. 2011. Improving Per-node Efficiency in the Datacenter with New OS Abstractions. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*.

[50] Clayne B Robison. 2017. Intel Ethernet Flow Director. https://software.intel.com/content/www/us/en/develop/articles/setting-up-intel-ethernet-flow-director.html. Accessed: 2020-5-22.

[51] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. 2019. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019 (APNet '19)*.

[52] Marta Rybczyńska. 2019. Improving idle behavior in tickless systems. https://lwn.net/Articles/775618/. Accessed: 2019-11-18.

[53] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*.

[54] Erfan Sharafzadeh, Seyed Alireza Sanaee Kohroudi, Esmail Asyabi, and Mohsen Sharifi. 2019. Yawn: A CPU Idle-State Governor for Datacenter Applications. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19)*.

[55] V. Spiliopoulos, A. Sembrant, and S. Kaxiras. 2012. Power-Sleuth: A Tool for Investigating Your Program's Power Behavior. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*.

[56] Akshitha Sriraman and Thomas F Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[57] A. Sriraman and T. F. Wenisch. 2018. Suite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*.

[58] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*.

[59] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. 2015. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*.

[60] Arjan van de Ven. [n.d.]. cpuidle: A new variant of the menu governor to boost IO performance. https://lwn.net/Articles/352180/. Accessed: 2018-5-25.

[61] BP Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics* 4, 3 (1962), 419–420.

[62] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. Tux2: Distributed Graph Computation for Machine Learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[63] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. 2013. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*.

[64] X. Zhan, R. Azimi, S. Kanev, D. Brooks, and S. Reda. 2017. CARB: A C-State Power Management Arbiter for Latency-Critical Workloads. *IEEE Computer Architecture Letters* 16, 1 (2017), 6–9.

[65] Y. Zhang, D. Meisner, J. Mars, and L. Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*.

[66] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A Kozuch. 2012. Saving Cash by Using Less Cache. In *4th Usenix workshop on hot topics in cloud computing (HotCloud '12)*.