Q-Zilla: A Scheduling Framework and Core Microarchitecture for Tail-tolerant Microservices

Amirhossein Mirhosseini
University of Michigan
miramir@umich.edu

Brendan L. West
University of Michigan
westbl@umich.edu

Geoffrey W. Blake
Amazon Web Services
blakgeof@amazon.com

Thomas F. Wenisch
University of Michigan
twenisch@umich.edu

ABSTRACT

Managing tail latency is a primary challenge in designing large-scale Internet services. Queuing is a major contributor to end-to-end tail latency, wherein nominal tasks are enqueued behind rare, long ones, due to Head-of-Line (HoL) blocking. In this paper, we introduce *Q-Zilla*, a scheduling framework to tackle tail latency from a queuing perspective, and CoreZilla, a microarchitectural instantiation of our framework. On the algorthmic front, we first propose *Server-Queue Decoupled Size-Interval Task Assignment (SQD-SITA)*, an efficient scheduling algorithm to minimize tail latency for high-disparity service distributions. SQD-SITA is inspired by an earlier algorithm, SITA, which explicitly seeks to address HoL blocking by providing an "express-lane" for short tasks, protecting them from queuing behind rare, long ones. But, SITA requires prior knowledge of task lengths to steer them into their corresponding lane, which is impractical. Furthermore, SITA may underperform an $M/G/k$ system when some lanes become underutilized. In contrast, SQD-SITA uses incremental preemption to avoid the need for a priori task-size information, and dynamically reallocates servers to lanes to increase server utilization with no performance penalty. We then introduce *Interruptible SQD-SITA*, which further improves tail latency at the cost of additional preemptions. Finally, we describe and evaluate *CoreZilla*, wherein a multi-threaded core efficiently implements ISQD-SITA in a software-transparent manner at minimal cost. Our evaluation demonstrates that CoreZilla improves tail latency over a conventional SMT core with 2, 4, and 8 contexts by 2.25×, 3.23×, and 4.38×, on average, respectively.

1. INTRODUCTION

Modern user-facing cloud services (e.g., web search, social media) must meet stringent Service Level Objectives (SLOs) to ensure responsiveness to millions of daily users [1, 2]. Often expressed in terms of (e.g., 99th percentile) tail latency, SLOs target the latency of the slowest requests, and thus bound the slowest interaction a user may have with the service. The "tail at scale" effect [3] makes tail-tolerant computing even more challenging—such services typically communicate via fan-out patterns wherein datasets are "sharded" across numerous "leaf" servers and their responses are aggregated before responding to the user. As such, the end-to-end latency is often dictated by the slowest leaves.

Two effects can lead to high tail latencies. First, applications' service time distributions often include rare cases that take much longer (10×-100× or more) than the mean [4]. Such tasks may require extraordinary processing time and/or trigger unusual code paths [5, 6, 7]. In other cases, system effects, such as from garbage collection [8, 3], memory management activities [9, 10], virtualization [11], network stack impediments [12, 13, 14], or co-runner application interference [15, 16, 17] may delay tasks.

Queuing effects are a second key contributor to end-to-end tail latency [18]. Queuing, where some requests must wait for others, arises at many system layers [3, 19]. Whereas queuing can affect average performance, its effect on tail latency may be devastating. For stable performance, systems must be engineered to ensure overall request arrival rate is below the aggregate system capacity (service rate). However, as both rates fluctuate, arrivals may briefly outstrip service capacity, causing requests to queue. Queuing delay is most apparent under high service time variability and/or high system load. Under high-disparity service distributions, many requests become delayed by an exceptionally slow one that stalls a server/core—a phenomenon called Head-of-Line (HoL) blocking. These delayed requests account for the bulk of the latency distribution tail under moderate-to-high loads [20].

In this paper, we introduce *Q-Zilla* as an algorithmic framework to tackle the problem of tail latency from a queuing perspective. In Q-Zilla, we make two distinct contributions: First, we propose *Server-Queue Decoupled Size-Interval Task Assignment (SQD-SITA)* as an efficient scheduling algorithm for high-disparity service distributions to minimize tail latency. SQD-SITA is inspired by an earlier algorithm, SITA [21, 22], which seeks explicitly to address HoL blocking by providing an "express-lane" for short tasks, protecting them from queuing behind rare, long ones. However, SITA requires prior knowledge of tasks lengths to steer them into their corresponding lane—an impractical assumption. Furthermore, whereas SITA is generally effective at reducing queuing delay and tail latency, it can fall short of the performance of a single-queue $M/G/k$[1] system when some lanes become underutilized. To overcome these challenges, SQD-SITA uses incremental preemption to avoid the need for a priori task-size information, and dynamically reallocates servers to lanes to boost server utilization. SQD-SITA never falls short of $M/G/k$ performance. We further introduce an enhanced variant of SQD-SITA, called Interruptible SQD-SITA (ISQD-SITA), which maximizes server utilization and further improves tail latency at the cost of additional preemptions.

Second, as an example realization of the Q-Zilla framework, we propose *CoreZilla*, a microarchitecture to minimize the tail latency of μs-scale microservices. Modern internet services use distributed microservice architectures, wherein a complex application is decomposed into numerous discrete microservices that interact over high-performance data center networks using remote procedure calls (RPCs) [25, 26, 27]. Many cloud service companies, including Amazon [28], LinkedIn [29], Netflix [30], and SoundCloud [31] have adopted microservice-based architectures. Example microservices include content caching [32, 33], protocol routing [34, 35], key-value lookup [36, 37], query rewriting [38], or

[1]Kendall's Notation: A/S/k [A/S: arrival/service distribution, k: number of servers, M: Markovian/memoryless (exponential) distribution, G: general distribution]—because requests usually originate from many independent sources (e.g., distinct users), they typically mimic Poisson (memoryless) arrivals [23, 24].

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00026

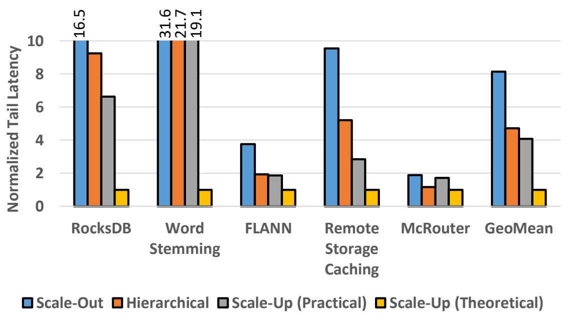

Figure 1: Normalized 99th percentile tail latency of different queuing organizations (16 dual-threaded cores).

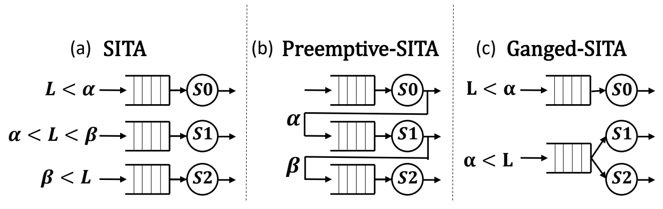

Figure 2: (a) Size-Interval Task Assignment (SITA); (b) SITA with incremental preemption (Preemptive-SITA); and (c) SITA with Server Ganging (Ganged-SITA). α and β represent cutoff points. L refers to task lengths.

other steps performed across various application tiers [39].

Managing tail latency is inherently more difficult for microservices, as individual RPCs/tasks are often only a few microseconds [40, 41]. Due to these short task lengths, it is often prohibitive to implement a "scale-up" queuing organization, wherein a single task queue is shared among all cores, as this organization leads to high contention on the shared queue—all cores must synchronize frequently to retrieve new tasks and the excessive synchronization costs may outweigh the benefits of sharing the task queue across cores. Nonetheless, such systems can adopt a hierarchical queuing scheme, wherein each core maintains a distinct queue that is shared only among hardware threads running on that core, achieving strong cache affinity for the local task queue.

Our proposal, CoreZilla, implements a hierarchical scheduling algorithm across hardware contexts in a Simultaneous Multithreading (SMT) core. It incorporates an automatic load adaptation scheme that dynamically tunes the number of physical contexts and schedules virtual contexts on them using ISQD-SITA. CoreZilla minimizes queuing delay and tail latency at each core, obviating the need for a cross-core scale-up queuing architecture and its associated synchronization and cache coherence overheads. Our evaluation demonstrates that CoreZilla improves tail latency over a conventional SMT core by 2.25×, 3.23×, 4.38× with 2, 4, 8 contexts, on average, respectively. We further compare CoreZilla to a hypothetical 32-core scale-up system with idealized (zero-overhead) synchronization. CoreZilla with 8 contexts still outperforms the idealized scale-up design by 12%, due to superior task scheduling.

2. BACKGROUND AND MOTIVATION

2.1 Queuing Organizations

Prior work has considered two different approaches to compose multiple servers: scale-out and scale-up [20]. In the scale-out model ($k - M/G/1$), a dispatcher balances incoming tasks among separate request queues dedicated to each server. In the scale-up model, servers instead fetch tasks from a single, shared queue. In principle, in terms of average and tail response time, the scale-up ($M/G/k$) organization always outperforms the scale-out organization. In the scale-up organization, no server will idle if there are tasks waiting in the central queue. However, in scale-out systems, a server with an empty queue will remain idle even while others have outstanding tasks. Furthermore, for scale-out systems, when a task takes longer than average all the tasks behind it suffer from HoL blocking. In contrast, in the scale-up model, tasks may be serviced by any server; stalling at one server has less impact on the system-wide instantaneous service rate.

Figure 1 reports the normalized 99th percentile tail latency of five different microservices measured at 70% load, on a Xeon processor with 16 cores and Hyperthreading. More details about the microservices are presented in Section 6. Different bars report the tail latency under (1) a scale-out queuing organization, wherein each core has a distinct task queue, (2) a hierarchical queuing organization, wherein the two hyperthreads of each core share a single task queue, (3) a scale-up organization, wherein a single task queue is shared among all cores, and (4) a theoretical scale-up model, wherein the costs of the synchronization and cache coherence are neglected (not measured on real hardware).

As shown in Figure 1, whereas a scale-up organization can theoretically result in 8.3× lower tail latency, on average, compared to a scale-out organization, it can only reduce the tail latency by 1.93× in practice due to communication and synchronization costs of the shared queue. However, a hierarchical approach only achieves 16% higher tail latency than a practical scale-up organization because (1) sharing the queue across hyperthreads within a core minimizes the synchronization/coherence overheads [42], and (2) as observed in prior work [20], only a small degree of concurrency is sufficient to eliminate the HoL blocking and allow the nominal tasks to drain past the rare, long ones. Interestingly, for microservices like McRouter, which do not exhibit heavy-tailed service distributions, the hierarchical approach results in lower tail latency than a practical scale-up organization.

Nonetheless, there is a still a $\sim 4\times$ gap between the hierarchical approach and the theoretical scale-up organization (with no synchronization or cache coherence overheads). Our goal is to design scheduling algorithms that can be implemented across hardware threads within a single physical core, to bridge the gap between theoretical and practical (hierarchical) queueing schemes.

2.2 SITA Scheduling

In an $M/G/k$ system with high service-time variability, especially with moderate-to-high load, it is probable that all servers become occupied by long tasks. In these cases, short tasks become enqueued behind long ones and suffer substantially from HoL blocking, increasing tail latency. The Size-Interval Task-Assignment (SITA) [21, 22] scheduling policy explicitly addresses this problem by providing an "express-lane" for the short tasks, protecting them from rare, long ones. Unlike $M/G/k$, SITA considers multiple servers with dedicated queues for each, similar to scale-out ($k - M/G/1$) systems. However, in contrast to scale-out systems, SITA assigns cutoff points to task-size intervals and steers tasks into queues based on the interval to which their size belongs. As an example, Figure 2(a) illustrates a SITA-scheduled system with three servers and cutoff points α and β. By providing an express-lane for tasks that are shorter than α, SITA prevents them from being enqueued behind long ones, to reduce tail latency under high service-time variability.

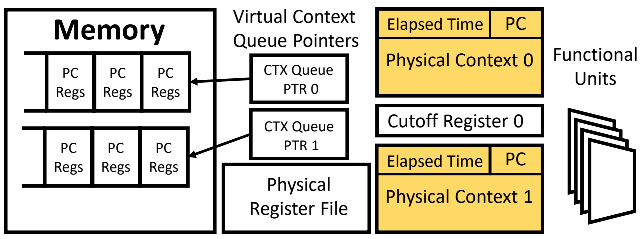

Figure 3: A two-way Express-Lane SMT (ESMT) core.

There are various approaches for tuning SITA cutoff points, such as equalizing the load across all servers [22]. However, to minimize the end-to-end response time, cutoff points must often be set in a way that intentionally unbalances load to favor either short or long tasks [21]. Finding the minimal cutoff points for SITA is, to date, an open problem and is usually done via empirical search, especially if the number of servers is small [24].

Although SITA is often effective at reducing tail latency and queuing delay under most high-variability service distributions, it suffers from two shortcomings: First, whereas $M/G/k$ fully utilizes all servers, SITA fails to do so, as it pre-assigns tasks to servers while ignoring their load, similar to scale-out systems; while there might be outstanding tasks at one queue, other servers may be idle waiting for new tasks to arrive. Second, SITA requires task sizes to be known in advance, which is an impractical assumption. We propose SQD-SITA to address these shortcomings.

2.3 Express-Lane SMT

An Express-lane SMT (ESMT) core [43], shown in Figure 3, comprises two physical contexts and a fixed number (e.g., 32) of virtual contexts, organized in two context queues in dedicated memory. The ESMT core may fetch and issue instructions only from the two physical contexts. Virtual contexts must be swapped into a physical context before they may execute. The ESMT core datapath resembles an SMT core with two hardware threads, and has similar area costs and clock frequency. Similar to existing SMT cores, the physical register file holds the architectural register values of all physical contexts and additional registers that enable register renaming and out-of-order execution.

When the first physical context reaches a preset service cutoff, it stops fetching/dispatching new instructions and drains the in-flight instructions from the re-order buffer. Once all in-flight instructions are drained, the context only contains architectural registers (all temporary physical registers are released). Then, its architectural state (program counter, registers, etc.) is swapped by the virtual context at the head of the first context queue and the preempted virtual context is placed at the end of the second context queue.

Swapping the virtual contexts into and out of the core is performed via microcode operations using the *Firmware Context Switching* (FCS) mechanism [44, 40]. FCS behaves as an additional instruction sequence for swapping threads, much like that done in software by the operating system, and therefore does not impose any additional requirements on the number of register file ports; microcode r-save/r-restore operations access the register file like typical load/store instructions. However, because FCS does not incur user/kernel mode transitions or switch address spaces, it is considerably faster than software context switches; while software context switches require 5-20μs [45, 46], a typical FCS can usually be performed within only 300ns [44]. Nonetheless, each virtual context is slowed down by at least a microsecond when it is preempted and moved to the second context queue due to indirect caching effects (i.e., cold misses when the task is resumed) [40]. This is not a significant problem in ESMT as the number of preemptions per virtual context is at most one.

ESMT allocates an idle virtual context to each incoming task. By having two context queues, ESMT provides an "express lane" for nominal tasks, protecting them from HoL blocking behind rare, long ones, thereby reducing queuing delay and tail latency—ESMT implements a preemptive variant of the SITA scheduling policy (which we will formally define in Section 3) with only two lanes.

A major drawback of ESMT, and SITA in general, is underutilization of physical contexts (servers in SITA) as they are statically mapped to queues. As we will show in Section 7, ESMT bridges the small gap between a hierarchical queuing scheme and a practical implementation of a cross-core scale-up queuing organization. However, due to said underutilization, it still significantly falls short of a theoretical scale-up queuing organization. Our goal is to design scheduling mechanisms that can be used in ESMT-like core microarchitectures to avoid underutilization while minimizing preemptions, closing the gap between hierarchical designs and theoretical scale-up organizations.

3. SERVER-QUEUE DECOUPLED SITA

We propose Server-Queue Decoupled (SQD)-SITA—a preemption-based variant of SITA that improves server utilization by dynamically reallocating servers to queues, which prevents servers from idling while tasks wait in another queue. We also introduce an enhanced variant of SQD-SITA, called Interruptible SQD-SITA (ISQD-SITA), which maximizes server utilization and further improves tail latency but may result in additional preemptions. Note that the (I)SQD-SITA scheduling algorithm may be implemented on different substrates, including multicore processors. However, in this paper, we propose an ESMT-based implementation of these algorithms, called CoreZilla, which is well-suited for modern μs-scale microservices. We construct the SQD-SITA and ISQD-SITA scheduling algorithms in three steps:

3.1 Adding Preemption and Ganging to SITA

We begin our development of SQD-SITA by enhancing SITA with *incremental preemption* and *server ganging*.

Incremental preemption. Whereas SITA statically assigns tasks to lanes based on their length, SQD-SITA incrementally preempts and migrates them to the end of the next queue as they reach a pre-determined service time cut-off. We call a SITA variant that also performs incremental preemption *preemptive-SITA*, as shown in Figure 2(b), and compare against it in our evaluation. The incremental preemption approach of preemptive-SITA is similar to the approaches used in ESMT [43] and some software frameworks [5, 47]. Unlike SITA, preemptive-SITA (and SQD-SITA) does not require prior knowledge of task lengths.

Server ganging. Server ganging (also called server pooling) is the practice of merging multiple scale-out queues into a single scale-up one by allowing multiple servers to share a single queue [20]. The original SITA algorithm was designed for task allocation in data center clusters, where a "server" represents a physical machine [21]. In such deployments, each server is associated with a distinct queue. However, SQD-SITA is intended primarily for scheduling tasks on cores/threads within a single machine. Therefore, it is possible to consolidate multiple queues and have fewer queues (and hence, cutoffs) than servers. Our key observation, which we will quantitatively explain in Section 7, is that only a few cutoffs are typically sufficient for SITA to achieve optimal isolation of long and short tasks; having a distinct cutoff per server leads to unnecessary load-imbalance, increasing queuing delay and tail latency. As a result, to construct the SQD-SITA algorithm, we start from a (preemptive) SITA variant where the number of queues is less than or equal to the number of servers, allowing a queue to be serviced by more than one server, as the first step towards server-queue decoupling. We call this variant Ganged-SITA, as shown in Figure 2(c), and compare against it in our evaluation.

3.2 Server-Queue Decoupling

The key feature of SQD-SITA is that it dynamically reallocates servers to queues to improve utilization and tail latency. We start from a preemptive-SITA system with server ganging as a strawman and look for scenarios where changing the assignment of servers to queues improves utilization without impacting performance. To this end, we derive upper and lower bounds on the number of servers that can be assigned to service tasks from each queue, and an algorithm to assign servers to queues in a way such that these bounds are met.

细节 All tasks enter the system at queue 0 and, when they reach the predetermined service cutoff, are preempted and enqueued in queue 1, and so on. Thus, each successive queue contains longer tasks. Whereas in a conventional queuing system, servers are assigned to particular queues, in SQD-SITA, we conceptually associate lanes with each queue, and servers join lanes to accept tasks from a particular queue. A server is only preempted when its task finishes or reaches the cutoff point where it should advance to the next lane (i.e., task preemptions are only triggered by timers, not any external events, such as new task arrivals); we will relax this assumption later when we discuss ISQD-SITA. When a task is immediately reassigned to the server it was running on before being preempted, the preemption is elided.

Reservations and starvation. In SQD-SITA, we conceive a system with N servers and M queues/lanes, numbered 0 to N-1, and 0 to M-1, respectively, where $M <= N$. We associate each lane with a positive number of *reservations*, where the sum of the number of reservations in all M lanes is equal to N. Thus, for example, if the number of servers and lanes is equal ($M = N$), each lane has only one reservation. The number of reservations specifies the minimum number of servers that must be available to serve tasks in a lane. We say that a lane *starves* if it has fewer tasks in service than its reservations while tasks wait in its queue. SQD-SITA's goal is to maximize server utilization while avoiding starvation. That is, we allow a lane to be assigned more servers than its reservations only if we can guarantee no other lane will starve.

Upper-bound criterion. To ensure a lane is assigned servers beyond its reservation only if we provably avoid starvation, we define an *upper bound criterion* to limit the maximum number of servers that may be assigned to each lane. The upper bound criterion assures that, if new tasks arrive, servers will be available at lower-numbered lanes so those lanes do not starve. The upper-bound criterion is expressed in Equation 1. For any given lane, the number of servers that may be allocated to this and all higher-numbered lanes, in total, is at most the cumulative reservations of these lanes. At a high level, Equation 1 ensures that, for any k, lanes 0 to k can always accommodate, in total, at least as many tasks as their
解释
cumulative reservations. To understand the purpose of this criterion, consider an extreme case where all the lanes 0 to k are empty. Even so, a burst of tasks might arrive and quickly flow into these lanes before existing (long) tasks running in higher-numbered lanes finish, starving low-numbered lanes. In the special case where the number of lanes and servers are equal (i.e., each lane has a single reservation), Equation 1 simplifies to Equation 2. In this case, only one server may service lane N-1 (longest tasks), at most two in lanes N-2 or higher, 3 in lanes N-3 or higher, and so on.

$$\forall\, 0 \le m \le M-1 \,:\, \sum_{i=m}^{M-1} servers(lane_i) \le \sum_{i=m}^{M-1} reservations(lane_i) \qquad (1)$$

$$\forall\, 0 \le m \le M-1 \,:\, \sum_{i=m}^{M-1} servers(lane_i) \le M-m \quad (M=N) \qquad (2)$$

Lower-bound criterion. Whereas the upper-bound criterion is necessary to avoid starvation, it is not sufficient. We must also introduce a *lower-bound criterion* on the allocation of servers to lanes, denoted in Equation 3. At a high level, the lower-bound criterion ensures that a lane will receive at least as many servers as the sum of its reservation and the unused reservations of higher-numbered lanes. The key intuition underlying this criterion is that tasks that reach a cutoff and must advance to the next lane *can take their server with them* to satisfy the next lane's reservation but this must not cause their previous lane to starve. The lower bound criterion ensures this does not happen. We will provide an example later to illustrate this case.

$$\forall\, 0 \le m \le M-1 \,:\, servers(lane_m) \ge \qquad (3)$$

$$min(tasks(lane_m), \sum_{i=m}^{M-1} reservations(lane_i) - \sum_{i=m+1}^{M-1} servers(lane_i))$$

Interestingly, we show that, to maximize utilization (while ensuring no lane starves), each lane must be allocated exactly as many servers as specified by the lower-bound criterion: the upper bound criterion (Equation 1) can also be written as Equation 4 by deriving the maximum number of servers that may be allocated to each lane from Equation 1. Equation 4 shows that this maximum number equals the second operand of the *min()* function in Equation 3. When this second operand is the smaller, the upper- and lower-bound criteria match. Conversely, if the first is the smaller (the lane has fewer tasks than reservations), allocating additional servers to the lane would leave those servers idle. As a result, in SQD-SITA, each lane (except lane 0) is allocated exactly as many servers as specified by the lower bound criterion (Equation 3), maximizing server utilization and ensuring no lane starves. "Extra" servers beyond those required to satisfy the lower-bound criterion wait at lane 0 in anticipation of newly arriving tasks.

$$\forall\, 0 \le m \le M-1 \,: \qquad (4)$$

$$servers(lane_m) \le \sum_{i=m}^{M-1} reservations(lane_i) - \sum_{i=m+1}^{M-1} servers(lane_i)$$

SQD-SITA algorithm. To ensure lanes are allocated servers to match the lower-bound criterion, SQD-SITA adopts the following algorithm: when a server becomes idle, it joins the *highest-numbered* lane where the lower-bound criterion (Equation 3) is not already met. Stated differently: when a server becomes idle, it joins the *highest-numbered* lane with a *non-empty queue*, where allocating one more server would not violate the upper-bound criterion (Equation 1 or 4).

Example. We illustrate SQD-SITA's operation in Figure 4. In this simple example, the number of servers and lanes are both three, and hence, each lane has a single server reservation. Initially, all servers accept tasks from lane 0. When a server becomes idle (i.e., its task finishes or reaches the cutoff point and is therefore preempted and advances to the next queue), the server joins a lane to accept a new task from the head of the corresponding queue according to the SQD-SITA procedure. Suppose three tasks (red, orange, yellow) arrive at queue 0; all three servers (S0-S2) accept tasks from lane 0 and all three tasks enter service (Figure 4 - Step 1). When the red task has been serviced for a time equal to the first cutoff point, it is preempted and advances to queue 1. The newly idle server, S0, then scans the queues to seek the eldest waiting

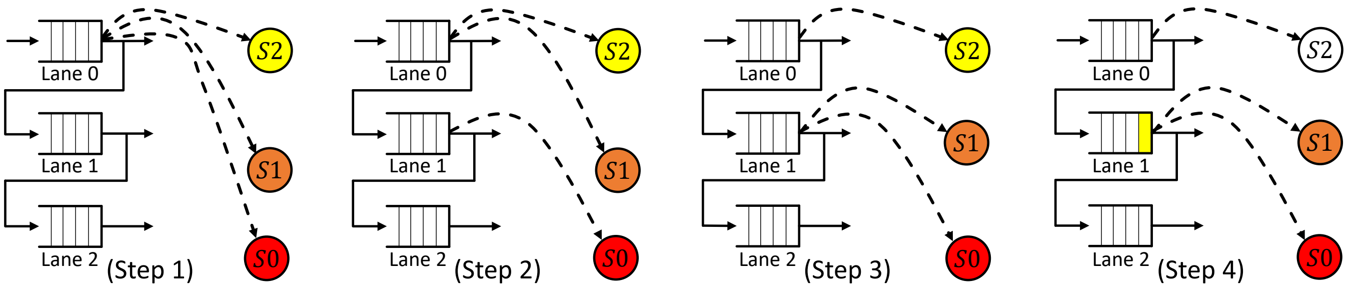

Figure 4: (Step 1) The initial configuration of an SQD-SITA system with three lanes and three servers, which are all initially allocated to lane 0, and (Step 2)/(Step 3)/(Step 4) when the first/second/third task reaches the first cutoff point.

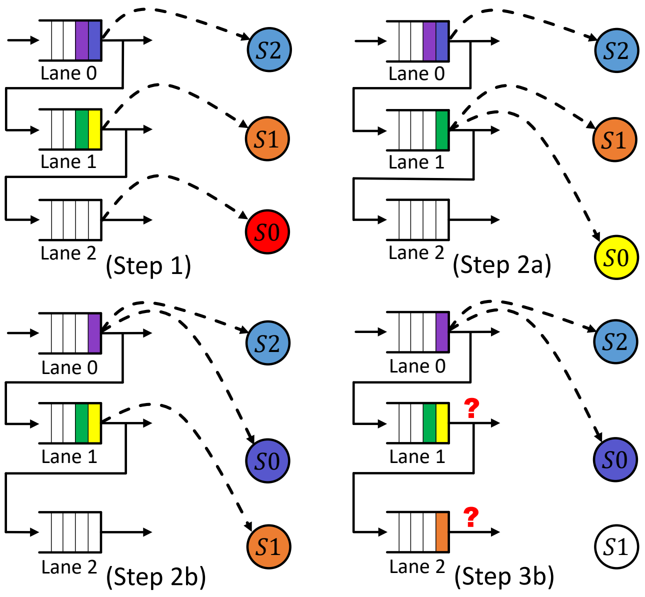

Figure 5: (Step 1) A valid configuration where each lane has at least one task and is allocated a server; (Step 2a) S0 follows the SQD-SITA procedure and joins lane 1 after finishing its task at lane 2; (Step 2b) S0 is allocated to lane 0 instead of lane 1 to prioritize short tasks; and (Step 3b) either lane 1 or lane 2 starves due to shortage of servers, resulted by the lower-bound violation in Step 2b.

task while respecting the upper-bound criterion. In this case, S0 joins lane 1, as shown in Figure 4 (Step 2), and resumes servicing the task that it had previously served (the preemption is elided). Now suppose the orange task running on S1 also reaches the cutoff point; it is also preempted and migrated to lane 1. Again, the newly idle server, S1, scans the queues, finds work in lane 1 (the just-preempted orange task), and resumes serving the task (Figure 4 - Step 3). Note that this configuration does not the violate upper-bound criterion, which allows at most two servers to be allocated to lanes 1 and 2 in total. However, subsequently, when the yellow task also reaches the cutoff point, although the task migrates to lane 1, server S2 may not join lane 1 and resume serving the task, as the resulting lane assignment would violate the upper-bound criterion. Hence, the yellow task is preempted and appended to queue 1, yielding the state in Figure 4 (Step 4), wherein server S2 remains idle at lane 0 (in anticipation of new arrivals) despite the yellow task waiting at lane 1.

Avoiding starvation. It may at first seem counter-intuitive that idle SQD-SITA servers scan lanes from highest to lowest to seek the eldest waiting task, as this appears to favor longer tasks over shorter ones. This policy ensures the lower-bound criterion is met at all lanes, while the the upper-bound criterion prevents too many servers from joining high-numbered lanes. We present another example to illustrate how this procedure avoids starvation. Consider the system in Figure 5, again with three servers and three lanes. In the scenario in Figure 5 (Step 1), servers S0, S1, and S2 are serving red, orange, and blue tasks and are assigned to lanes 2, 1, and 0, respectively. The indigo and violet tasks wait in the queue at lane 0 while the yellow and green tasks wait in the queue at lane 1. Lane 2's queue is empty. When S0's task completes, it scans lanes to seek the eldest waiting task, finding work (the yellow task) at lane 1 (Figure 5 - Step 2a). Were it instead assigned to lane 0 to prioritize shorter tasks (Figure 5 - Step 2b), when S1's orange task reaches the cutoff point, is preempted, and advances to lane 2 (Figure 5 - Step 3b), one of the two lanes 1 and 2 has to starve as they both have tasks but there is only one server S1 available to be assigned. This example shows that greedy prioritization of short-task lanes over long-task ones can lead to starvation; the lower-bound criterion (violated in Step 2b) ensures this may not occur.

Corollaries. We note two additional provable properties of SQD-SITA: (1) no task experiences longer response time under SQD-SITA than it can experience with (server-ganged) preemptive-SITA[2], and (2) SQD-SITA minimizes preemptions as a server always follows its task when the task advances across lanes, to avoid preemption, unless doing so violates the upper-bound criterion[3].

3.3 Interruptible SQD-SITA

SQD-SITA limits the number of servers assigned to each lane, according to the upper-bound criterion, to guarantee that no task experiences a longer response time than under preemptive-SITA. As a result, whereas SQD-SITA improves server utilization over both SITA and preemptive-SITA, it fails to achieve the optimal utilization, wherein servers never remain idle when tasks are queued at some lane (i.e., SQD-SITA is not work-conserving)—SQD-SITA intentionally idles servers in anticipation of new task arrivals. In this section, we propose Interruptible SQD-SITA (ISQD-SITA), which seeks to maximize server utilization by allowing servers to join lanes in violation of the upper-bound criterion *if and only if* they would otherwise remain idle.

However, to avoid starvation (assure each lane can accommodate at least as many tasks as its reservations), ISQD-SITA requires an additional preemption mechanism, which allows new arrivals to

[2]Tasks enter lanes in FIFO order and no lane ever starves.

[3]By induction, any elder tasks are either being served, or cannot be currently served due to the upper-bound criterion (Equation 1).

preempt running tasks (in contrast to SQD-SITA, wherein tasks are only preempted at cut-off points). When a new task arrives, if no idle server waits at lane 0, ISQD-SITA scans lanes from highest to lowest to check if the upper-bound criterion has been violated in any lane. If so, it preempts the youngest running task in that lane and allocates the preempted server to the arriving task in lane 0.

ISQD-SITA algorithm. Algorithm 1 shows the high-level procedure that SQD-SITA and ISQD-SITA follow when a server is preempted. The highlighted parts are exclusive to ISQD-SITA. The procedure has two phases. In phase 1—shared between SQD-SITA and ISQD-SITA—an idle server joins the highest-numbered non-empty queue that has capacity for an additional server without violating the upper-bound criterion. Under SQD-SITA, if no such queue is found, the server will idle, waiting for new work to arrive at lane 0. Under ISQD-SITA, the server instead picks the head of the lowest-numbered non-empty queue (phase 2), in violation of the upper-bound criterion. If a new task arrives, some server must be preempted immediately to ensure the lower-bound criterion is still met. Note that even though ISQD-SITA permits upper-bound violations, it never violates the lower-bound criterion. As a result, external preemptions (i.e., non–timer-based preemptions) only occur upon new task arrivals.

Corollaries. As shown in Algorithm 1, while phase 1 scans the lanes from highest to lowest, phase 2 scans them from lowest to highest. This has two advantages: (1) it prioritizes shorter tasks, and (2) it guarantees that no tasks wait in a lane numbered lower than the one with the upper-bound violation[4]. The latter property has two useful implications: First, it retains the guarantee that no lane starves since the system behaves the same as SQD-SITA in all lanes numbered above that where the violation occurred (those lanes meet both the lower-bound and the upper-bound criteria). The yellow highlight in Algorithm 1 is an optimization that exploits this property to stop scanning lanes in phase 1 if an upper-bound violation is detected (as the rest of the lanes have empty queues). Second, (ignoring preemption cost) it ensures that no task experiences higher response time under ISQD-SITA than SQD-SITA, again since all lanes numbered higher than the violating lane behave as in SQD-SITA and no task can be waiting in lower-numbered lanes. Therefore, if detecting new arrivals and preempting existing tasks is inexpensive, ISQD-SITA should be preferred over SQD-SITA.

4. CORE-ZILLA MICROARCHITECTURE

In this section, we describe the CoreZilla microarchitecture, which extends the ESMT design and employs ISQD-SITA to schedule tasks to hardware threads. CoreZilla enables a hierarchical queuing organization, wherein each physical core has a dedicated task queue (as in scale-out designs) to avoid the synchronization and cache coherence overheads of sharing a queue across all cores in scale-up systems. However, each core's task queue is shared among its hardware threads to minimize queuing delay and tail latency at the core, obviating the need for cross-core scale-up solutions. CoreZilla facilitates software-transparent preemptive scheduling within the core to eliminate HoL blocking and minimize tail latency. In addition to ISQD-SITA scheduling, CoreZilla dynamically tunes the number of active hardware threads, based on the system load, to yield optimal tail latency. In the following subsections, we describe the two key components of CoreZilla: *Hierarchical Scheduling* and *Automatic Load Adaptation*.

[4]By induction: the first time an upper-bound violation occurs, the property holds; after that, if a task is enqueued in a lower-numbered lane than the violating lane, that task would be selected by phase 2 of the algorithm, so the property continues to hold.

Algorithm 1: SQD-SITA and ISQD-SITA pseudocodes
(Red highlight: only in ISQD-SITA — Yellow highlight: optimization)

```
event server S is preempted
    server_count = 0                                        // phase 1
    reservation_count = 0
    for i = M-1 to 0 do
        server_count += lanes[i].num_servers()
        reservation_count += lanes[i].num_reservations()
        if lanes[i].has_waiting_tasks() and server_count < reservation_count
          then
            lanes[i].allocate_server(S)
            return
        end
        else if server_count > reservation_count then
            break
        end

    end
    for i = 0 to M-1 do                                     // phase 2

        if lanes[i].has_waiting_tasks() then
            lanes[i].allocate_server(S)
            return
        end
    end
    lanes[0].allocate_server(S)                         // last resort

end
event task arrival
    if lanes[0].has_idle_servers() then
        return
    end
    server_count = 0
    reservation_count = 0
    for i = M-1 to 0 do
        server_count += lanes[i].num_servers()
        reservation_count += lanes[i].num_reservations()
        if server_count > reservation_count then
            S = lanes[i].youngest_running_task()
            preempt(S)
            lanes[0].allocate_server(S)
            return
        end
    end
end
```

4.1 Hierarchical Scheduling

As shown in Figure 3, a strawman ESMT core is composed of two physical contexts and a fixed number of virtual contexts that are organized in two context queues in dedicated memory. CoreZilla extends ESMT to have a tunable number of physical contexts (e.g., 2-8), and virtual contexts (e.g., 32). Thus, rather than only two context queues, in CoreZilla we provision a number of context queues that can be configured to be less than or equal to the number of physical contexts. These context queues each correspond to an ISQD-SITA queue and maintain the backlog of virtual contexts ready for execution in a particular ISQD-SITA lane. Each physical context represents an ISQD-SITA server.

The CoreZilla scheduling hardware manages the assignment of physical contexts (ISQD-SITA servers) to context queues (ISQD-SITA lanes) in accordance with constraints outlined in Section 3. When a physical context becomes idle (due to task completion or preemption), the hardware scheduler selects the next virtual context from the head of a context queue by scanning for non-empty queues starting with the highest numbered lane (eldest tasks), based on the procedure explained in Algorithm 1. The scheduling hardware further tracks the execution time for each virtual context so that it can determine when the virtual context reaches the execution time limit imposed by the next ISQD-SITA scheduling cutoff. When this cutoff is reached, the task is preempted and the virtual context is descheduled and appended to the end of the context queue for the next ISQD-SITA lane. Tasks in the highest lane have no cutoff and will execute to completion.

CoreZilla provides a task-based software model where a single worker thread is pinned to each virtual context. The worker threads retrieve tasks from a single shared software task queue, as in an $M/G/k$ system, and manage both task queue synchronization and the CoreZilla scheduling hardware transparently to the executing tasks. Worker threads run the procedure shown in Algorithm 2. Each virtual context has an associated *elapsed time* that is maintained with the context and tracks how long the virtual context has been scheduled on a physical context since it began a new task. The elapsed time implicitly maps the context to an ISQD-SITA lane, based on where it falls relative to the ISQD-SITA cutoffs. An idle thread retrieves a task from the software task queue and resets the elapsed execution time for the virtual context to zero. The task then begins execution. When the elapsed execution time reaches the next ISQD-SITA cutoff, the context is preempted and appended to the context queue for the next ISQD-SITA lane. As noted in Section 3, we optimize for the special case where the next ISQD-SITA queue is empty and elide the context switch if the physical context would immediately reschedule the same virtual context. The central idea of our approach is to map a task-based software model to a thread-based execution model that allows the hardware to schedule among a fixed number of threads (virtual contexts) while managing a potentially unbounded number of tasks.

The scheduling hardware tracks the elapsed time for all physical contexts, and therefore also tracks the assignment of physical contexts to lanes, which can be inferred from the elapsed times and the cutoffs. Time can be maintained using any convenient monotonic counter (e.g., Intel's timestamp counter). Cutoffs are specified in a set of special registers and are set by the task framework based on prior knowledge or runtime monitoring of the service distribution. To enable ISQD-SITA, we add a mechanism that interrupts execution upon a write to a monitored memory location, to be able to detect new arrivals (highlighted line; only needed for ISQD-SITA). This scheme is similar to existing memory monitoring mechanisms, such as `mwait` in Intel processors [48], which detect changes to a memory location by tracking coherence invalidation messages.

Algorithm 2: High-level procedure of the worker threads

```
while true do
    while task == nil do
        reset_elapsed_time()
        task = dequeue(task_queue)
    end
    async_monitor(task_queue)
    run(task)
    task = nil
end
```

Figure 6 illustrates the microarchitecture of a 4-way CoreZilla, with four physical contexts and three context queues (the number of reservations for lane 0 is two). Whereas there is fixed mapping between physical contexts and context queues in ESMT—leading to underutilization of physical contexts—in CoreZilla, the mapping changes dynamically, improving throughput and efficiency.

4.2 Automatic Load Adaptation

CoreZilla's two-level thread-context mechanisms enable hardware scheduling to implement ISQD-SITA. We add an additional mechanism to tune the number of active physical contexts to best serve the current load and service characteristics.

Some prior works [16, 49] advocate reducing or disabling hardware multithreading to avoid interference, which can exacerbate tail latency. Others [50, 51, 52] advocate hardware multithreading to improve core utilization and reduce Total Cost of Ownership (TCO). The guidance from these studies is in conflict.

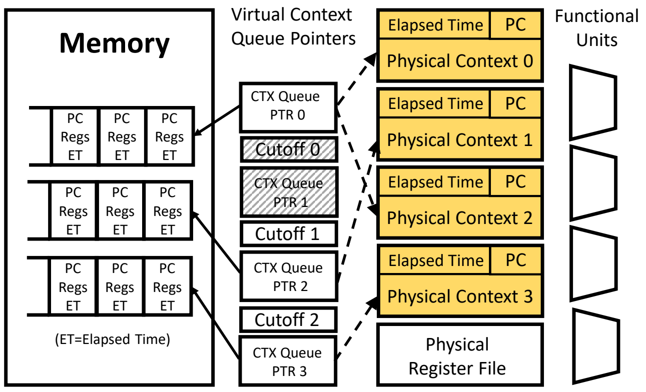

Figure 6: A 4-way CoreZilla with three context queues.

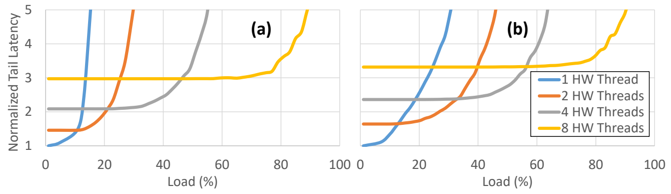

Figure 7: Normalized 99th percentile latency at different loads for (a) Word Stemming and (b) McRouter.

We observe that the right number of hardware threads to balance tail latency and utilization depends on the workload's service time distribution and system load. When load is low, and end-to-end response time is dominated by service (rather than queuing) time, it is better to disable additional hardware threads, allowing a single thread to enjoy higher execution bandwidth and run faster without interference. However, as load increases, additional threads enable higher instruction throughput, which results in higher overall service rate at the core and reduced queuing time. Furthermore, as the service time distribution grows more heavy-tailed, additional threads become critical to minimize HoL blocking and excessive queuing delays—high disparity service distributions are common in microservices [7, 53, 14, 26].

We illustrate these effects with an example. Figure 7(a) and (b) present end-to-end tail latency at different loads for Word Stemming and McRouter microservices (described in detail later in Section 6). As these figures show, fewer threads yield lower tail latency at low load, as each thread executes faster and queuing is rare at low load. However, as load increases, the better instruction throughput enabled by additional threads results in substantially lower queuing delay and tail latency. Furthermore, we observe that the break-even points for McRouter (b), which has a low-disparity service distribution, occur at higher relative loads compared to the break-even points of word stemming (a), which is a heavy-tailed microservice; word stemming requires more threads at lower load than McRouter to prevent HoL blocking.

To exploit this trade-off, CoreZilla incorporates an automatic load adaptation mechanism that dynamically tunes the number of physical contexts to minimize tail latency. CoreZilla's load adaptation comprises offline profiling and online adaptation phases. The offline profiling phase constructs a profile of tail latency vs. load across thread counts for a particular workload like those shown in

Figure 7. The critical break-even points (crossings) in the load curves are then recorded in a lookup table.

During execution, the instantaneous arrival rate (the rate tasks are added to SQD-SITA lane 0) is monitored over 5-millisecond-scale windows to estimate load. Then, the lookup table is consulted to determine how many physical contexts to activate. A 5-ms window is long enough relative to the μs-scale service times of microservices to yield accurate load measurement, but also short enough to capture transient load changes as load fluctuations usually occur at least at the granularity of 10s of milliseconds [5, 25].

5. DISCUSSION

Finding cutoffs. Finding SQD-SITA (and SITA) cutoffs is a non-trivial problem and is usually performed by empirical search. When the number of lanes is only two, cutoffs can be found by quantizing the service time distribution and linearly searching the entire space. However, with a larger number of servers/lanes, the search space grows combinatorially. Furthermore, our algorithm for finding cutoffs must consider server-ganged variants of SITA and SQD-SITA algorithms, where the lane count may be smaller than the server count. Hence, even the lane and reservation counts may not be fixed parameters when searching for the optimal cutoffs.

We propose a hill climbing-based heuristic for finding cutoffs. We empirically find the search space to be convex and the cutoffs to be near high quantiles of the service time distribution. We initialize the search with all reservations in the final lane (i.e., no cutoffs). We then consider moving one reservation to a new lane, searching for a tail-latency-minimizing cutoff value over descending quantiles of the service time distribution. We then iterate, considering (1) moving an additional reservation to the most recent lane, or (2) adding a new lane with a lower cutoff. The algorithm halts when neither moving a reservation nor adding a lane improves tail latency. Search time scales with the granularity of the search over quantiles and is independent of the number of servers/lanes. Hence, it is scalable to many servers. We performed point validations against exhaustive searches for a 4-server system and were not able to improve over this heuristic.

Comparison with SRPT and PS. Shortest Remaining Processing Time (SRPT) and Processor Sharing (PS) are other scheduling algorithms tailored for high-disparity service distributions. SRPT preempts a running task upon a new arrival and gives the execution resources to the new task if it is shorter than the remaining portion of the currently running task. While SRPT is proven to be asymptotically optimal [54, 55] (at most a constant factor worse than the optimal) for heavy-tailed service distributions, it can cause long tasks to starve and yield unpredictable results for tail latency, especially at high loads. Furthermore, it is a size-based algorithm and is only applicable when task sizes are known or can be predicted accurately in advance [56] (much like SITA).

PS fairly shares execution capacity across all tasks by time sharing servers at small scheduling quanta. While PS is not size-based and does not starve long tasks, as in SRPT, it entails frequent preemptions (proportional to the task sizes), which hurt performance and may be impractical. We will show in Section 7 that (I)SQD-SITA outperforms both SRPT and PS, given enough servers, by isolating short tasks from long ones while respecting FIFO ordering of task arrivals.

Finite virtual contexts. Because CoreZilla supports only a finite number of virtual contexts, it is possible for all of them to be assigned tasks longer than the first (I)SQD-SITA cutoff (i.e., if 32 incomplete tasks all execute past the first cutoff). This scenario leads to a violation of both upper and lower-bound criteria, as there is no context able to execute newly arriving tasks in lane 0. Once any task completes execution, the hardware scheduler resumes obeying the (I)SQD-SITA constraints. The number of virtual contexts should be provisioned such that the probability that all virtual contexts are occupied by tasks longer than the first cutoff is negligible. This probability vanishes rapidly as the number of virtual contexts grows; 32 virtual contexts is more than sufficient.

Table 1: Microarchitecture details

Core	4-wide issue OoO, 192-entry ROB/PRF, 48-entry LQ, 32-entry SQ
SMT	ICOUNT [59] Fetch, up to 8 physical contexts, 32 virtual contexts
L1 cache	Private 64KB I/D, 64B lines, 2-way SA
LLC	1 MB per core, 64B lines, 8-way SA
Memory	50 ns access latency

Hardware costs and scalability. CoreZilla builds upon the ESMT hardware substrate, and from a hardware point of view, only extends it to have more than two physical contexts. The only hardware extension an N-way ESMT/CoreZilla requires over an N-way SMT core are N registers to hold virtual context queue pointers, N-1 registers to hold service cutoff points, N-1 timers/counters to measure elapsed times, and three registers to hold load break-even points. All of these structures in total add negligible area/power overheads to the datapath as they are small compared to the core's physical register file.Furthermore, all of these structures scale linearly with respect to the number of SMT execution lanes. Therefore, the main scalability bottleneck to add lanes is the SMT mechanism itself, because having more execution lanes complicates both the core frontend and backend, and particularly, requires a larger register file to at least contain all the architectural registers of all physical contexts. Consequently, no commercial system supports more than 8 SMT threads; we have also only considered 2, 4, and 8 threads, which are the options available in IBM Power 8/9 microarchitectures that currently support the largest number of SMT threads. We have modeled these additional structures in McPAT [57] and found the area and power overheads of CoreZilla to be within 2% and 3% of a baseline SMT core, respectively. There is no accurate way to measure the clock frequency impact, except via RTL-level implementation and synthesis. However, we do not expect the additional control logic to be on the critical path and both the original Firmware Context Switching (FCS) [44] and subsequent designs employing this mechanism [40] report negligible cycle-time impact.

6. EVALUATION METHODOLOGY

To evaluate SQD-SITA, we employ Stochastic Queuing Simulation (SQS), based on the BigHouse methodology and simulator [23]. We simulate until we achieve 95% confidence intervals of 5% error in reported results. We find cutoffs based on the heuristic explained in Section 5. We measure service time distributions of five microservices and feed their latency distribution histograms to our SQS framework. To accurately model the cost of preemption and context switches in CoreZilla and its alternatives, we model their hardware in the gem5 simulator [58] and include the preemption/restart latencies in our SQS experiments, following prior work [40]. We consider 2,4, and 8 SMT lanes (physical contexts) to model the available options for the number of hardware threads in IBM Power8/9 processors.

We use the following microservices for our evaluation:

- **FLANN:** we use a microservice benchmark based on FLANN [60], an open-source library for performing fast approximate nearest neighbor searches in high-dimensional spaces. FLANN uses Locality Sensitive Hashing (LSH) to perform k-nearest neighbor identification—a critical microservice employed in content-based similarity search. We use Google's Open Images dataset [61]. We consider variants of FLANN with (1) 20-bit, and (2) 12-bit LSH key sizes.

- **RocksDB:** We use RocksDB [62], a popular and widely deployed in-memory key-value store developed by Facebook. We use an open-source Twitter dataset [50] and RocksDB's default load generator with two different configurations, (1) where 90% of requests are GETS and 10% are SCANs, and (2) where 99% of requests are GETS and 1% are SCANs. SCAN requests scan 5000 keys and take approximately 50× longer than GETs.
- **Word Stemming (WS):** Stemming is a normalization process that reduces words to their root, employed in various cloud services, such as web search. We employ a word stemming microservice based on Oleander's implementation [63] of the Porter stemming algorithm [64, 65]. It is a high-disparity microservice as it hard-codes all stemming paths (prefixes, suffixes, etc.) into the control-flow and the length of paths for different words might be substantially different. Our queries include words from Wikipedia Redux [66].
- **Remote Storage Caching (RSC):** We implement a remote storage caching microservice as a simplified variant of existing host-side Flash caches [67, 68, 69, 70]. Our RSC microservice maps linear block addresses of a remote storage system to a local low-latency SSD using Cuckoo hashing [71]. We only consider read transactions. The three outcomes of a lookup query are that it might be a hit in the local memory, hit in local SSD, or a miss.
- **McRouter:** We employ a consistent hashing microservice, based on Facebook's McRouter [34, 35], to route Key-Value (KV) operations to 100 leaf servers via a consistent hash function. We generate key-value lookup queries from an open-source Twitter dataset [50].

7. RESULTS

7.1 SQD-SITA performance analysis

We first study alternative queuing organizations to gain insight into how each organization behaves in principle. In this section, we neglect the costs of preemption, and also consider scheduling algorithms that require task lengths to be known a priori, which is impractical for systems like CoreZilla. Figure 8 reports normalized 99th percentile tail latencies achieved under various queuing organizations, including $M/G/k$, SITA, Ganged SITA (G-SITA), Preemptive Ganged SITA (PG-SITA), SQD-SITA, ISQD-SITA, Processor Sharing (PS), and Shortest Remaining Processing Time (SRPT). For PS, we consider a 2μs scheduling quantum. We consider 2, 4, and 8 servers, which correspond to (a), (b), and (c), respectively. For each number of servers in each workload, we set the offered load to the break-even point where our load adaptation system selects that configuration (e.g., break-even points in Figure 7).

As Figure 8 shows, ISQD-SITA improves tail latency by 2.28×, 3.39×, 4.76× over an M/G/k system with, 2, 4, and 8 servers, on average, respectively. Furthermore, while (I)SQD-SITA consistently improves tail latency over PG-SITA, there are a few cases where it falls short of the tail latency achieved by (G-)SITA. These cases arise because non-preemptive SITA is a size-based algorithm, which can exploit its prior knowledge of task sizes. In addition, note the impact of server ganging—with 4 and 8 servers, SITA

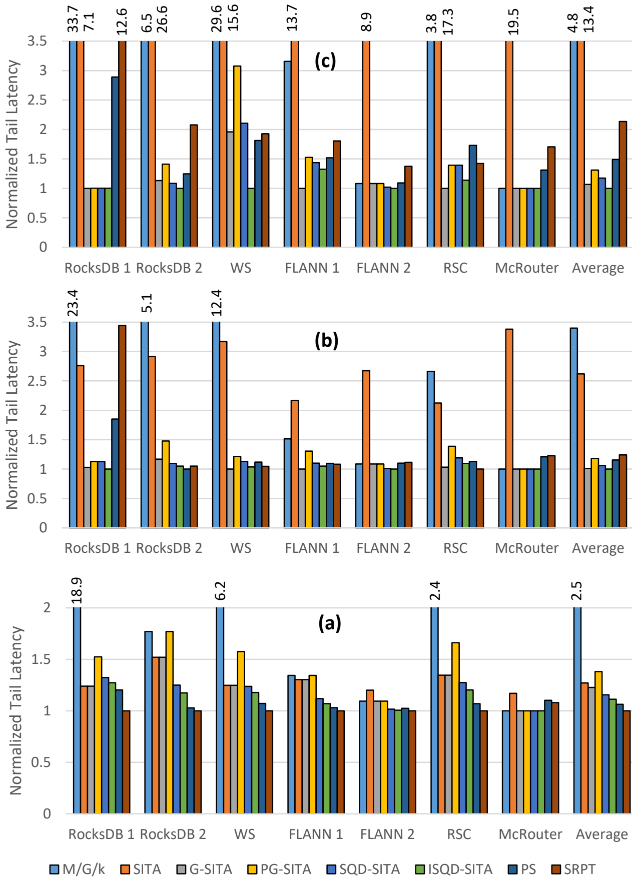

Figure 8: Normalized 99th percentile latency under various organizations for (a) 2, (b) 4, and (c) 8 servers.

yields significantly higher tail latencies than G-SITA, which, in many cases, are even higher than those of the baseline M/G/k system. Having too many lanes results in unnecessary load imbalance in systems like SITA, which, unlike SQD-SITA, statically assign servers to lanes; only a few lanes are sufficient to eliminate HoL blocking, regardless of the number of servers.

We observe that, with 8 servers, ISQD-SITA consistently outperforms PS and SRPT (by 49% and 2.13 ×, on average), and with 4 servers ISQD-SITA consistency outperforms PS and is outperformed by SRPT only for RSC workload (16% and 24% average improvement over PS and SRPT). However, with two servers, ISQD-SITA is outperformed by PS and SRPT by 6% and 11%, on average, respectively. Neither PS nor SRPT respect FIFO ordering of the tasks; PS fairly shares the system capacity among all tasks and SRPT strictly prioritizes short tasks. These algorithms are well-suited only for high-disparity service distributions and fall short of a FIFO-ordered (M/G/k) system for low-disparity distributions [72, 73] (e.g., McRouter). However, with enough servers, ISQD-SITA isolates short tasks from long ones while respecting their FIFO arrival ordering, outperforming PS and SRPT.

7.2 CoreZilla performance analysis

In this section, we seek to find the optimal core microarchitecture, and therefore only consider non–size-based scheduling policies, which can be practically adopted by a system like CoreZilla, and also consider the costs of preemption. Figures 9(a), (b), and (c) report normalized 99th percentile tail latencies achieved by differ-

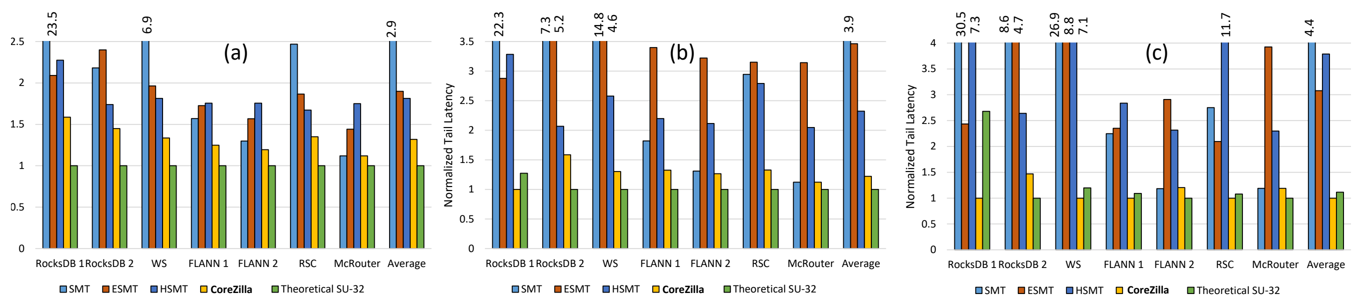

Figure 9: Normalized 99th percentile latency of CoreZilla and alternatives for (a) 2, (b) 4, and (c) 8 hardware threads.

ent core microarchitectures with 2, 4, and 8 physical SMT contexts, implementing different scheduling policies. We compare SMT, Hierarchical SMT (HSMT) [40], Express-Lane SMT (ESMT) [43], and CoreZilla, with the same number of physical contexts (2/4/8). HSMT effectively implements PS scheduling by time multiplexing all virtual contexts on the physical contexts at 2μs time quanta. We also compare against a theoretical 32 cores scale-up organization—in all other designs, there is a distinct task queue per physical core.

CoreZilla improves tail latency over a conventional SMT core and an ESMT core with 2, 4, 8 physical contexts by 2.25×, 3.23×, 4.38×, and 38%, 2.83×, 3.07× on average, respectively. Improvements are slightly smaller than reported in the previous section as the costs of preemption are now included. Also it is notable that whereas ESMT achieves only slightly higher tail latency than CoreZilla with 2 physical contexts, it falls well short of CoreZilla's performance with more physical contexts. ESMT suffers significantly from underutilization of physical contexts as it statically maps each physical context to a context queue. Server ganging and dynamic reallocation of physical contexts to context queues in (I)SQD-SITA solve this problem in CoreZilla. Moreover, note that while ESMT and HSMT may result in tail latencies that are higher than SMT, CoreZilla never falls short of SMT performance, thanks to the same mechanisms.

CoreZilla significantly outperforms HSMT (which implements PS), since the number of preemptions incurred under PS are much higher than under (I)ISQD-SITA; with PS, each task requires, on average, $mean-task-size/scheduling-quantum$ preemptions ($mean-task-size$ can be 10s – 100s of microseconds). However, with SQD-SITA, each task incurs at most one preemption per cutoff/lane. Unlike SQD-SITA, the number of preemptions under ISQD-SITA is not bounded. However, as these results show, the net gain is always positive.

Finally, we observe that, with two and four physical contexts, CoreZilla achieves 99th percentile tail latency that is within 31% and 22% of a theoretical 32-core scale-up organization, respectively. With eight physical contexts, however, due to superior task scheduling, CoreZilla is even able to achieve 12% lower average tail latency compared to a theoretical 32-core scale-up organization, obviating the need for having a cross-core shared queue and its attendant overheads.

7.3 Impact of preemptions in ISQD-SITA

As noted before, ISQD-SITA may incur additional preemptions compared to SQD-SITA. Figure 10 reports the average number of preemptions in 8-server Preemptive-SITA, SQD-SITA, and ISQD-SITA systems. Here we consider non-ganged (i.e., 8-lane) variants of these algorithms, as these incur the most preemptions. Whereas ISQD-SITA increases the number of preemptions relative to SQD-SITA, it often significantly reduces their number compared to Preemptive-SITA. Interestingly, the average number of preemptions per task of ISQD-SITA exceeds that of Preemptive-SITA for the RSC workload (0.84 vs. 0.63) because RSC includes both exceptionally long and exceptionally short tasks. In this scenario, before any nominal or long task finishes, one task that violates the upper bound criterion may be be preempted and resumed multiple times, due to rapid arrivals of such short tasks. Nevertheless, as shown in Figure 10, the overall average number of preemptions in (I)SQD-SITA is negligible.

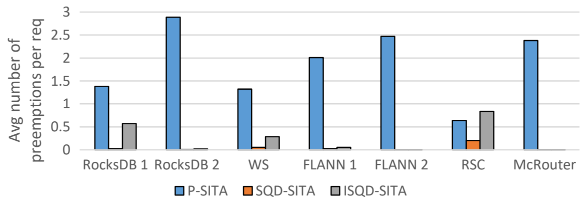

Figure 10: Average number of preemptions per request for different scheduling policies in various microservices.

8. RELATED WORK

The most related work to ours is RPCValet [74], which proposes a potential solution for approximating a scale-up queuing organization on a scale-out system in the presence of on-chip integrated NICs. With RPCValet, instead of the integrated NIC "pushing" packets into each core's dedicated queues, which may result in load imbalance and HoL blocking, each core "pulls" a packet from the NIC once it is done processing the previous packet. The single shared packet queue is managed in hardware by the on-chip NIC and distributes packets into the cores' local queues. RPCValet's solution is only applicable to systems with on-chip integrated NICs and can at best achieve the lower-bound tail latency of a theoretical scale-up system. However, as we showed in Section 7, with sufficient physical contexts, CoreZilla can reduce the tail latency even beyond that of a theoretical 32-core scale-up system because CoreZilla augments the queuing model with the (I)SQD-SITA scheduling policy, which is inherently able to isolate long and short tasks and prevent HoL blocking.

A large body of prior work seeks to lower the tail latency of interactive services. However, most past studies target classic monolithic services with millisecond- to second-scale service times and, hence, require different approaches for our target microservices. We discuss various classes of such studies:

Parallelization and Heterogeneity. One class seeks to accelerate long tasks by parallelizing them on multiple cores to reduce their processing time and queuing impact. Jeon et al. [75] propose an adaptive solution to determine the required degree of parallelism for each query based on the offered load. In a follow up work [76], they propose a feature-based prediction model to predict long tasks and parallelize them. Haque et al. [5] propose an incremental approach that increases the degree of parallelism as a task advances in execution. In a follow up work [77], they move the longer tasks to faster cores in heterogeneous platforms to accelerate them further. All of these techniques are only applicable to ms-scale services that are easily parallelized, such as web search.

Voltage/Frequency Boosting. Another class seeks to boost core voltage and frequency to accelerate long tasks. Adrenaline [6] considers SET requests in a key-value store as long tasks and accelerates them; their approach is application-specific and not easily generalizable. Rubik [78] takes a more general approach and by probabilistically accelerating queries based on the service time distribution and the position of each query in the queue. However, it fails to capture heavy-tailed service distributions, where the probability of HoL blocking is high, and relative position of queries in the queue has low correlation with their queuing time.

Non-FIFO Scheduling. Another class seeks to minimize tail latency of high-disparity services by employing better-than-FIFO scheduling schemes to eliminate HoL blocking. Various authors [47, 53] propose per core task queues augmented by work-stealing to improve tail latency. While work-stealing is an effective approach for improving utilization and throughput, it is not well-suited for server applications, where the objective is to minimize the response-time. Work-stealing allows cores to take tasks from other queues when their own queue is empty to improve utilization; it does not solve the HoL blocking problem within each queue.

Shinjuku [7] proposes to address tail latency by employing PS for high-disparity task distributions. As we have shown, (I)SQD-SITA usually outperforms PS as PS does not respect the FIFO ordering of task arrivals. Baraat [79] proposes a FIFO with limited parallelism (FIFO-LM) scheme, wherein a number of oldest tasks (e.g., 8) are time-multiplexed but younger tasks wait for them in a FIFO queue. Interestingly, this mechanism is already implemented in SMT cores as the active threads (which serve the oldest tasks) are truly sharing the processor. CoreZilla outperforms conventional SMT designs which implement such a policy.

Size-based Scheduling. Finally, a few prior studies propose size-based scheduling mechanisms by correlating the processing time of a request with one of its features. Harchol-Balter et al. [56] propose to use SRPT for file and web servers by estimating request processing times based on file sizes. Didona et al. [80] propose a cross-core sharding of key-value store queries, based on object sizes. Their approach effectively implements server-ganged SITA across cores by estimating task lengths based on object sizes. These approaches are only applicable to their respective services and are not comparable to generic approaches like SQD-SITA.

9. CONCLUSION

In this paper, we proposed *Q-Zilla* as a scheduling framework and its hardware instantiation to tackle the tail latency of microservices. Q-Zilla is composed of *Server-Queue Decoupled – Size-Interval Task Assignment* (SQD-SITA), as a tail-aware scheduling algorithm, and Interruptible SQD-SITA (ISQD-SITA) which further improves tail latency at the cost of additional preemptions. (I)SQD-SITA dynamically reallocates servers to lanes to increase server utilization with no performance penalty. Finally, we proposed *CoreZilla*, as a hardware realization of (I)SQD-SITA in a multithreaded core. CoreZilla improves tail latency over a conventional SMT core with 8 threads by 4.38× and outperforms a theoretical 32-core scale-up organization by 12%, on average.

10. ACKNOWLEDGEMENTS

We thank Hossein Golestani, Mohammad Bakhshalipour, and the anonymous reviewers for their valuable feedback. This work was supported by the Center for Applications Driving Architectures (ADA), a JUMP center co-sponsored by SRC and DARPA.

11. REFERENCES

[1] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," *IEEE micro*, vol. 23, no. 2, pp. 22–28, 2003.

[2] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "Softsku: optimizing server architectures for microservice diversity@ scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 513–526, ACM, 2019.

[3] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[4] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pp. 1–10, IEEE, 2016.

[5] M. E. Haque, Y. He, S. Elnikety, R. Bianchini, K. S. McKinley, *et al.*, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," in *ACM SIGPLAN Notices*, vol. 50, pp. 161–175, ACM, 2015.

[6] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 271–282, IEEE, 2015.

[7] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pp. 345–360, 2019.

[8] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu, "Detecting transient bottlenecks in n-tier applications through fine-grained analysis," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pp. 31–40, IEEE, 2013.

[9] D. Skarlatos, N. S. Kim, and J. Torrellas, "Pageforge: a near-memory content-aware page-merging architecture," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 302–314, ACM, 2017.

[10] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," in *ACM SIGPLAN Notices*, vol. 53, pp. 679–692, ACM, 2018.

[11] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 329–341, 2013.

[12] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, p. 9, ACM, 2012.

[13] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–14, ACM, 2014.

[14] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang, and C. Pu, "A study of long-tail latency in n-tier systems: Rpc vs.

asynchronous invocations," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 207–217, IEEE, 2017.

[15] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 406–418, IEEE Computer Society, 2014.

[16] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 450–462, ACM, 2015.

[17] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48, pp. 77–88, ACM, 2013.

[18] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Communications of the ACM*, vol. 61, no. 8, pp. 65–72, 2018.

[19] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1063–1075, 2019.

[20] A. Mirhosseini and T. F. Wenisch, "The queuing-first approach for tail management of interactive services," *IEEE Micro*, vol. 39, no. 4, pp. 55–64, 2019.

[21] M. E. Crovella, M. Harchol-Balter, and C. D. Murta, "Task assignment in a distributed system (extended abstract): improving performance by unbalancing load," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, pp. 268–269, ACM, 1998.

[22] M. Harchol-Balter, M. E. Crovella, and C. D. Murta, "On choosing a task assignment policy for a distributed server system," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 204–228, 1999.

[23] D. Meisner, J. Wu, and T. F. Wenisch, "Bighouse: A simulation infrastructure for data center systems," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pp. 35–45, IEEE, 2012.

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

[25] A. Sriraman and T. F. Wenisch, "utune: Auto-tuned threading for oldi microservices," in *Operating Systems Design and Implementation (OSDI), 2018 USENIX Symposium on*, 2018.

[26] A. Sriraman and T. F. Wenisch, "μ suite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–12, 2018.

[27] Y. Gan and C. Delimitrou, "The Architectural Implications of Cloud Microservices," in *Computer Architecture Letters (CAL), vol.17, iss. 2*, Jul-Dec 2018.

[28] Staci D. Kramer, "The biggest thing amazon got right: The platform." [Online; accessed 27-Apr-2018].

[29] Steven Ihde and Karan Parikh, "From a monolith to microservices + rest: the evolution of linkedin's service architecture." [Online; accessed 27-Apr-2018].

[30] Tony Mauro, "Adopting microservices at netflix: Lessons for architectural design." [Online; accessed 27-Apr-2018].

[31] Phil Calcado, "Building products at soundcloud âĂŤpart i: Dealing with the monolith." [Online; accessed 27-Apr-2018].

[32] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, 2004.

[33] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing.," in *NSDI*, vol. 13, pp. 371–384, 2013.

[34] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani, "Introducing mcrouter: A memcached protocol router for scaling memcached deployments," 2014.

[35] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, "Scaling memcache at facebook.," in *nsdi*, vol. 13, pp. 385–398, 2013.

[36] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 295–306, 2015.

[37] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store.," in *USENIX Annual Technical Conference*, pp. 103–114, 2013.

[38] M. Barhamgi, D. Benslimane, and B. Medjahed, "A query rewriting approach for web service composition," *IEEE Transactions on Services Computing*, 2010.

[39] Y. Gan, Y. Zhang, D. Cheng, *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–18, ACM, 2019.

[40] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing server efficiency in the face of killer microseconds," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 185–198, IEEE, 2019.

[41] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[42] H. Golestani, A. Mirhosseini, and T. F. Wenisch, "Software data planes: You can't always spin to win," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 337–350, ACM, 2019.

[43] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch, "Express-lane scheduling and multithreading to minimize the tail latency of microservices," in *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 194–199, IEEE, 2019.

[44] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder, "Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pp. 183–194, IEEE, 2004.

[45] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 workshop on Experimental computer science*, p. 2, ACM, 2007.

[46] D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *Proceedings of the 2007 workshop on Experimental computer science*, p. 4, ACM, 2007.

[47] J. Li, K. Agrawal, S. Elnikety, Y. He, I. Lee, C. Lu, K. S. McKinley, *et al.*, "Work stealing for interactive services to meet target latency," in *ACM SIGPLAN Notices*, vol. 51, p. 14, ACM, 2016.

[48] P. Guide, "Intel® 64 and ia-32 architectures software developerâĂŹs manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.

[49] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martınez, "Workload characterization of interactive cloud services on big and small server platforms," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 125–134, IEEE, 2017.

[50] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGPLAN Notices*, vol. 47, pp. 37–48, ACM, 2012.

[51] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 158–169, IEEE, 2015.

[52] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of hardware data prefetchers on server processors," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–29, 2019.

[53] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, no. EPFL-CONF-231395, 2017.

[54] N. Bansal and M. Harchol-Balter, *Analysis of SRPT scheduling: Investigating unfairness*, vol. 29. ACM, 2001.

[55] M. Nuyens and B. Zwart, "A large-deviations analysis of the gi/gi/1 srpt queue," *Queueing Systems*, vol. 54, no. 2, pp. 85–97, 2006.

[56] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, "Size-based scheduling to improve web performance," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 207–233, 2003.

[57] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, IEEE, 2009.

[58] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[59] S. Ren, Y. He, S. Elnikety, and K. S. McKinley, "Exploiting processor heterogeneity in interactive services," 2013.

[60] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, 2014.

[61] Google, "OpenImages: A public dataset for large-scale multi-label and multi-class image classification., howpublished = "`https://github.com/openimages/dataset`"."

[62] Facebook, "Rocksdb." `https://rocksdb.org/`, 2018.

[63] Oleander-Solutions, "Oleander stemming library." `http://www.oleandersolutions.com/stemming/stemming.html`.

[64] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[65] M. F. Porter, "Snowball: A language for stemming algorithms," 2001.

[66] Wikipedia-Redux. `https://reagle.org/joseph/blog/social/wikipedia/10k-redux.html`.

[67] S. Byan, J. Lentini, A. Madan, and L. Pabon, "Mercury: Host-side flash caching for the data center," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pp. 1–12, IEEE, 2012.

[68] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *2015 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 51–60, IEEE, 2015.

[69] D. Arteaga and M. Zhao, "Client-side flash caching for cloud systems," in *Proceedings of International Conference on Systems and Storage*, pp. 1–11, ACM, 2014.

[70] D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer, "Flash caching on the storage client," in *USENIX ATC'13 Proceedings of the 2013 USENIX conference on Annual Technical Conference*, USENIX Association, 2013.

[71] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[72] O. Boxma and B. Zwart, "Tails in scheduling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 4, pp. 13–20, 2007.

[73] A. Wierman and B. Zwart, "Is tail-optimal scheduling possible?," *Operations research*, vol. 60, no. 5, pp. 1249–1257, 2012.

[74] A. Daglis, M. Sutherland, and B. Falsafi, "Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 35–48, ACM, 2019.

[75] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Adaptive parallelism for web search," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 155–168, ACM, 2013.

[76] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Predictive parallelization: Taming tail latencies in web search," in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pp. 253–262, ACM, 2014.

[77] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting heterogeneity for tail latency and energy efficiency," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 625–638, ACM, 2017.

[78] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 598–610, ACM, 2015.

[79] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 431–442, ACM, 2014.

[80] D. Didona and W. Zwaenepoel, "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). USENIX Association, Boston, MA*, 2019.