μqSim: Enabling Accurate and Scalable Simulation for Interactive Microservices

Yanqi Zhang, Yu Gan, and Christina Delimitrou
Electrical and Computer Engineering Department
Cornell University
{*yz2297, yg397, delimitrou*}*@cornell.edu*

***Abstract*— Current cloud services are moving away from monolithic designs and towards graphs of many loosely-coupled, single-concerned microservices. Microservices have several advantages, including speeding up development and deployment, allowing specialization of the software infrastructure, and helping with debugging and error isolation. At the same time they introduce several hardware and software challenges. Given that most of the performance and efficiency implications of microservices happen at scales larger than what is available outside production deployments, studying such effects requires designing the right simulation infrastructures.**

We present μqSim, a scalable and validated queueing network simulator designed specifically for interactive microservices. μqSim provides detailed intra- and inter-microservice models that allow it to faithfully reproduce the behavior of complex, many-tier applications. μqSim is also modular, allowing reuse of individual models across microservices and end-to-end applications. We have validated μqSim both against simple and more complex microservices graphs, and have shown that it accurately captures performance in terms of throughput and tail latency. Finally, we use μqSim to model the tail at scale effects of request fanout, and the performance impact of power management in latency-sensitive microservices.

I. Introduction

An increasing amount of computing is now performed in the cloud, primarily due to the resource flexibility benefits for end users, and the cost benefits for both end users and cloud providers [1]–[3]. Users obtain resource flexibility by scaling their resources on demand and being charged only for the time these resources are used, and cloud operators achieve cost efficiency by multiplexing their infrastructure across users [4]–[11].

Most of the services hosted in datacenters are governed by strict quality of service (QoS) constraints in terms of throughput and tail latency, as well as availability and reliability guarantees [5], [6], [12], [13]. In order to satisfy these often conflicting requirements, cloud services have seen a significant shift in their design, moving away from the traditional monolithic applications, where a single service encompasses the entire functionality, and adopting instead a multi-tier microservices application model [14], [15]. Microservices correspond to fine-grained, single-concerned and loosely-coupled application tiers, which assemble to implement more sophisticated functionality [14], [16]–[24].

Microservices are appealing for several reasons. First, they improve programmability by simplifying and accelerating deployment through modularity. Unlike monolithic services, each microservice is responsible for a small, well-defined fraction of the entire application's functionality, with different microservices being independently deployed. Second, microservices can take advantage of language and programming framework heterogeneity, since they only require a common cross-application API, typically over remote procedure calls (RPC) or a RESTful API [25]. Third, individual microservices can easily be updated, or swapped out and replaced by newer modules without major changes to the rest of the application's architecture. In contrast, *monoliths* make frequent updates cumbersome and error-prone, and limit the set of programming languages that can be used for development.

Fourth, microservices simplify correctness and performance debugging, as bugs can be isolated to specific components, unlike monoliths, where troubleshooting often involves the end-to-end service. Finally, microservices fit nicely the model of a container-based datacenter, with a microservice per container, and microservices being scaled independently according to their resource demands. An increasing number of cloud service providers, including Twitter, Netflix, AT&T, Amazon, and eBay have adopted this application model [14].

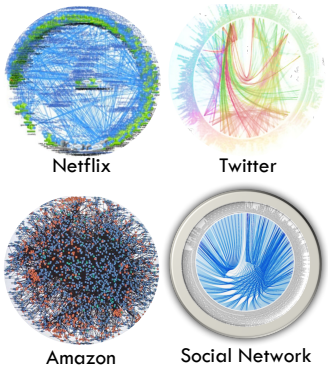

Figure 1: Microservices graphs in three large cloud providers [14]–[16], and our `Social Network` service.

Despite their advantages, microservices introduce several hardware and software challenges. First, they put increased pressure on the network system, as dependent microservices reside in different physical nodes, relying on an RPC or HTTP layer for communication. Second, microservices further complicate cluster management, as instead of allocating resources to a single service, the scheduler must now determine the impact of dependencies between any two microservices in order to guarantee end-to-end QoS.

Microservices further exacerbate tail-at-scale effects [12] as a single poorly-configured microservices on the critical path can cause cascading QoS violations resulting in end-to-end performance degradation. Typical dependency graphs between microservices in production systems involve several hundred microservices; Fig. 1 shows a few representative examples from Netflix, Twitter, Amazon [14], and one of the applications used later in this paper.

Given the fact that unpredictable performance often only emerges at large scale, it is critical to study the resource management, availability, and responsiveness of microservices at scales larger than what is possible in typical research facilities. Towards this effort we leverage the following insight: a positive side-effect of the simplicity of individual microservices is that - unlike complex monoliths - they conform to the principles of queueing theory. In this work we use this insight to enable accurate microservices simulation that relies on queueing networks to capture the impact of dependencies between microservices.

We present μqSim, an accurate and scalable simulator for interactive cloud microservices. μqSim captures dependencies between individual microservices, as well as application semantics such as request batching, blocking connections, and request pipelining. μqSim relies on a user-provided high-level declarative specification of the microservices dependency graph, and a description of the available server platforms. μqSim is an event-driven simulator, and uses a centralized scheduler to dispatch requests to the appropriate microservices instances.

We validate μqSim against both simple and complex microservices graphs, including multi-tier web applications, services with load balancing, and services with high request fanout, where all leaf nodes must respond before the result is returned to the user. We show that μqSim faithfully reproduces the performance (throughput and tail latency) of the real application in all cases. We then use μqSim to evaluate two use cases for interactive microservices. First, we show that tail at scale effects become worse in microservices compared to single-tier services, as a single underperforming microservice on the critical path can degrade end-to-end performance. Second, we design and evaluate a QoS-aware power management mechanism for microservices both in a real server and in simulation. The power manager determines the appropriate per-microservice QoS requirements needed to meet the end-to-end performance constraints, and dynamically adjusts frequency accordingly.

II. Related Work

Queueuing network simulators are often used to gain insight on performance/resource trade-offs in systems that care about request latency [26]–[30]. The closest system to μqSim is BigHouse, an event-driven queueing simulator targeting datacenter service simulation. BigHouse represents workloads as inter-arrival and service distributions, whose characteristics are obtained through offline profiling and online instrumentation. The simulator then models each application as a single queue, and runs multiple instances in parallel until performance metrics converge. While this offers high simulation speed, and can be accurate for simple applications, it introduces non-negligible errors when simulating microservices because intra-microservice stages cannot be captured by a single queue. For example, a request to an application like memcached must traverse multiple stages before completion, including network processing, e.g., TCP/IP rx, processing in OS event-driven libraries like `epoll` and `libevent`, reading the received requests from the socket, and finally processing the user request. Along each of these stages there are queues, and ignoring them can underestimate the impact of queueing on tail latency. Furthermore, BigHouse is only able to model single tier applications. Microservices typically consist of multiple tiers, and inter-tier dependencies are not a straightforward pipeline, often involving blocking, synchronous, or cyclic communication between microservices.

μqSim takes a different approach by explicitly modeling each application's execution stages, and accounting for queueing effects throughout execution, including request batching, e.g., in `epoll`, request blocking, e.g., in http 1/1.1 & disk I/O, and request parallelism, e.g., across threads or processes. Given that many of these sources of queueing are repeated across microservices, μqSim provides modular models of application stages, which can be reused across diverse microservices. Moreover, μqSim supports user-defined microservice graph topology and flexible blocking and synchronization behaviors between tiers of microservices.

III. Design

μqSim is designed to enable modularity and reuse of models across microservices and end-to-end applications, as well as to allow enough flexibility for users to incorporate their own applications in the simulator. The core of μqSim is implemented in standard C++ with 25,000 lines of codes. The simulator interface requires the user to express a microservice's internal logic using a JSON configuration file, similar to the one shown in 1. The user additionally needs to express the end-to-end service's architecture, i.e., how are different microservices connected to each other, and what hardware resources are available in the cluster. We summarize the input users express to the simulator in Table I. Below we detail each of these components.

A. Simulation Engine

μqSim uses discrete event simulation, as shown in Fig. 2. An event may represent the arrival or completion of a job (a request in a microservice), as well as cluster administration operations, like changing a server's DVFS setting. Each event is associated with a timestamp, and all events are stored in increasing time order in a priority queue. In every simulation

Table I: Simulation inputs.

`service.json`	Internal architecture of a microservice
`graph.json`	Inter-microservice topology
`path.json`	Paths (sequence of microservices) that requests follow across microservices
`machines.json`	Server machines & available resources
`client.json`	Input load pattern
`histograms`	Processing time PDF per microservice

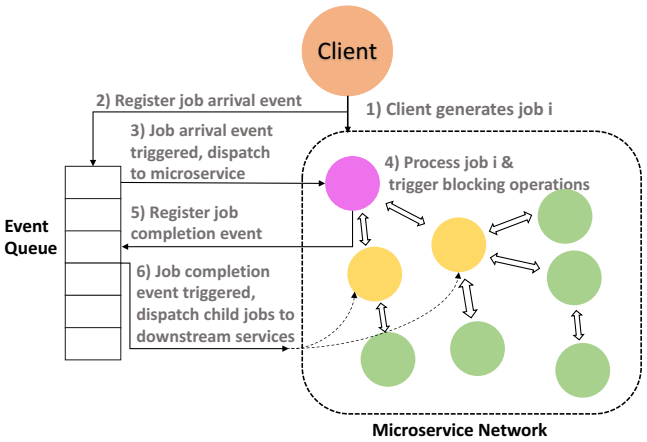

Figure 2: Overview of the event-driven behavior in μqSim.

cycle, the simulation queue manager queries the priority queue for the earliest event. It then uses the microservice model the event corresponds to to compute the execution time and resources required to process the event, as well as to insert causally dependent events to the priority queue. These include requests triggered in other microservices as a result of the processed event. Simulation completes when there are no more outstanding events in the priority queue.

B. Modeling Individual Microservices

μqSim models each individual microservice with two orthogonal components: *application logic* and *execution model*. The application logic captures the behavior of a microservice. The basic element of the application logic is a stage, which represents an execution phase within the microservice, and is essentially a queue-consumer pair, as defined in queueing theory [31], [32]. Each stage can be configured with different queueing features like batching or pipelining, and is coupled with a job queue. For example, the `epoll` stage is coupled with multiple subqueues, one per network connection. A stage is also assigned to one or more execution time distributions that describe the stage's processing time under different settings, like different DVFS configurations, different loads and different thread counts. μqSim supports processing time expressed using regular distributions, such as exponential, or via processing time histograms collected through profiling, which requires users to instrument applications and record timestamps at boundaries of queueing stages.

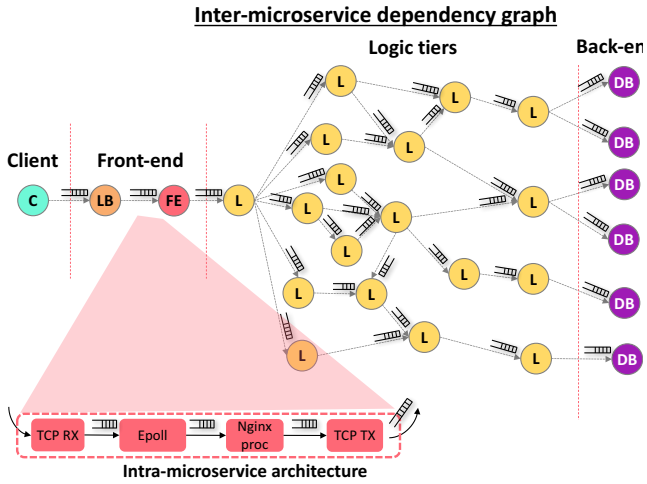

Figure 3: Overview of the modeling of an end-to-end service built with microservices in μqSim (top), and modeling of the execution stages within a single microservice (bottom). C: client, LB: load balancer, FE: front-end, L: logic tier, DB: database tier.

Multiple application logic stages are assembled to form execution paths, corresponding to a microservice's different code paths. Finally, the model of a microservice also includes a state machine that specifies the probability that a microservice follows different execution paths.

Fig. 1 shows memcached's application logic and execution path using μqSim's JSON template. The main stages include `epoll`, `socket_read`, and `memcached_processing`. This excludes network processing, which is modeled as a separate process in the simulator: each server is coupled with a network processing process as a standalone service, and all microservices deployed on the same server share the processed handling interrupts. Both `epoll` and `socket_read` use request batching, and can return more than one jobs. The `epoll` queue classifies jobs into subqueues based on socket connections, and returns the first N jobs of each active subqueue (defined in "queue_parameter"). `socket_read` similarly classifies jobs based on connections, but returns the first N jobs from a single ready connection at a time. The `memcached_processing` and `socket_send` stages do not use batching, and their queues simply store all jobs in one queue.

It is important to note that the processing time of `epoll` and `socket_read` are runtime dependent: `epoll`'s execution time increases linearly with the number of active events that are returned, and `socket_read`'s processing time is also proportional to the number of bytes read from socket. Finally, memcached's execution model consists of two paths, one for `read` requests, and one for `write`. These two paths consist of exactly the same stages in the same order, and are only used to distinguish between different

Table II: Specification of the server used for all validation experiments of μqSim.

Model	Intel(R) Xeon(R) CPU E5-2660 v3
OS	Ubuntu 14.04(kernel 4.4.0)
Sockets	2
Cores/Socket	10
Threads/Core	2
Min/Max DVFS Freq.	1.2GHz/2.6GHz
L1 Inst/Data Cache	32KB/32KB
L2 Cache	256KB
L3 (Last-Level) Cache	25.6MB
Memory	8x16GB 2400MHz DDR4
Hard Drives	2x 2TB 7.2K RPM SATA
Network Bandwidth	1Gbps

processing time distributions. Each path for memcached is deterministic, therefore there is no need for a probability distribution to select an execution path. One example where probabilistically selecting execution paths is needed is MongoDB, where a query could be either a cache hit that only accesses memory, or a cache miss that results in disk I/O. The probability for each path in that case is a function of MongoDB's working set size and allocated memory. Currently μqSim only models applications running on standard Linux. We defer modeling acceleration techniques like user level networking, such as DPDK and FPGA accelerated networking to future work.

The execution model of a microservice also describes how a job is processed by the simulator. Currently μqSim supports two models: simple and multi-threaded. A simple model directly dispatches jobs onto hardware resources like CPU, and is mainly used for simple (single stage) services. Multi-threaded models add the abstraction of a thread or process, which users can specify either statically or using a dynamic thread/process spawning policy. In a multi-threaded model, a job will be first dispatched to a thread, and the microservice will search for adequate resources to execute the job, or stall if no resources are available. The multi-threaded model captures context switching and I/O blocking overheads, and is typically used for microservices with multiple execution stages that include blocking/synchronization.

C. Modeling a Microservice Architecture

A microservice architecture is specified in three JSON files that describe the cluster configuration, microservice deployment, and inter-microservice logic. The cluster configuration file (`machines.json`) records the available resources on each server. The microservice deployment file (`graph.json`) specifies the server on which a microservice is deployed – if specified, the resources assigned to each microservice, and the execution model (simple or multi-threaded) each microservice is simulated with. The microservice deployment also specifies the size of the connection pool of each microservice, if applicable.

Listing 1: JSON Specification for memcached

```
{"service_name": "memcached",
  "stages": [{
      "stage_name":"epoll", "stage_id":0,
      "queue_type":"epoll", "batching":true,
      "queue_parameter":[null, N]}, {
      "stage_name":"socket_read", "stage_id":1,
      "queue_type":"socket", "batching":true,
      "queue_parameter":[N] }, {
      "stage_name":"memcached_processing",
      "stage_id":2, "queue_type":"single",
      "batching":false, "queue_parameter":null}, {
      "stage_name":"socket_send", "stage_id":3,
      "queue_type":"single", "batching":false,
      "queue_parameter":null}],
  "paths": [{
      "path_id":0, "path_name":"memcached_read",
      "stages":[0, 1, 2, 3]}, {
      "path_id":1, "path_name":"memcached_write",
      "stages":[0, 1, 2, 3]}]}
```

Finally, the inter-microservice path file (`path.json`) specifies the sequence of individual microservices each job needs to go through. Fig. 3 shows such a dependency graph between microservices, from a client (C), to a load balancer (LB), front-end (FE), logic tiers (L), and back-end databases (DB), The same figure also shows the intra-microservice execution path for one of the microservices, `front-end`, implemented in this example using NGINX [33]. If the application exhibits control flow variability, users can also specify multiple inter-microservice paths, and the corresponding probability distribution for them. The basic elements of an inter-microservice path are path nodes, which are connected in a tree structure and serve three roles:

- *Specify the microservice, the execution path within the microservice, and the order of traversing individual microservices*. When entering a new path node, the job is sent to a microservice instance designated by the path node and connected with the microservices that the job has already traversed, as defined in `graph.json`. Each path node can have multiple children, and after execution on the current path node is complete, μqSim makes a copy of the job for each child node, and sends it to a matching microservice instance.
- *Express synchronization*. Synchronization primitives are prevalent in microservice architectures, such as in the case where a microservice can start executing a job if and only if it has received all information from its upstream microservices. In μqSim, synchronization requirements are expressed in terms of the fan-in of each inter-microservice path node: before entering a new path node, a job must wait until execution in all parent nodes is complete. For example, if NGINX_0 serves as a proxy and NGINX_1 and NGINX_2 operate as file servers, for each user request, NGINX_0 sends requests to both NGINX_1 and NGINX_2, waits to synchronize their responses, and then sends the final response to the client.

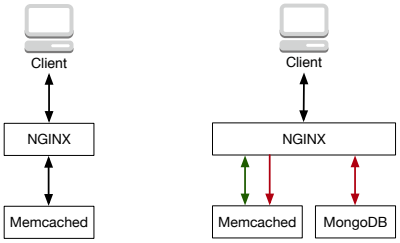

Figure 4: The architecture of the 2- and 3-tier applications (NGINX-memcached and NGINX-memcached-MongoDB).

- *Encode blocking behavior*. Blocking behavior between microservices is common in RPC frameworks, http1/1.1 protocols, and I/O accessing. To represent arbitrary blocking behavior, each path node has two operation fields, one upon entering the node and another upon leaving the node, to trigger blocking or unblocking events on a specific connection or thread. Assume for example a two-tier application, with NGINX as the front-end webserver, and `memcached` as the in-memory caching tier. The client queries the webserver over http 1.1. Once a job starts executing, it blocks the receiving side of the incoming connection (since only one outstanding request is allowed per connection in http 1.1). The condition to unblock this path once request processing is complete is also specified in the same JSON file. When the job later returns the `<key,value>` pair from memcached to NGINX, μqSim searches the list of job ids for the one matching the request that initiated the blocking behavior, in order to unblock the connection upon completion of the current request. Users can also specify other blocking primitives, like thread blocking, in μqSim.

IV. Validation

We now validate μqSim against a set of real microservices running on a server cluster. The configuration of each server is detailed in Table II. We validate μqSim with respect to two aspects of application behavior.

- First, we verify that the simulator can reproduce the load-latency curves of real applications, including their saturation point. Given that latency increases exponentially beyond saturation, ensuring that the simulator captures the bottlenecks of the real system is essential in its effectiveness.
- Second, we verify that μqSim accurately captures the magnitude of the end-to-end average and tail latency of real applications.

A. Simple Multi-tier Microservices

We first validate μqSim against a simple 2- and 3-tier application, comprised of popular microservices deployed in many production systems. The 2-tier service consists

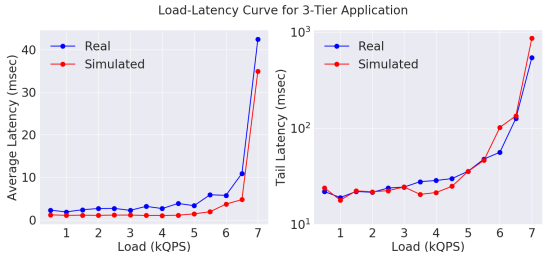

Figure 6: Validation of the three-tier (NGINX-memcached-MongoDB) application.

of a front-end webserver, implemented using NGINX, and an in-memory caching key-value store, implemented with memcached. The 3-tier application additionally includes a persistent back-end database, implemented using MongoDB. The architectures of the two applications are shown in Fig.4(a) and (b) respectively. In the 2-tier service, NGINX receives the client request over http 1.1, queries memcached for the requested `key`, and returns the `<key,value>` pair to the client. In the 3-tier service, NGINX first queries the cache for the requested `key` (memcached), and if not present, queries the back-end database (MongoDB). Memcached implements a write-allocate policy; on a (mem)cache miss, the `<key,value>` is also written to memcached to speed up subsequent accesses.

For all experiments, we use an open-loop workload generator, implemented by modifying the `wrk2` client [34]. The client runs on a dedicated server, and uses 16 threads and 320 connections to ensure no client-side saturation. For this experiment both job inter-arrival times and request value sizes are exponentially distributed. Finally, for both the 2- and 3-tier services, memcached is allocated 1GB memory, and MongoDB has unlimited disk capacity.

For the 2-tier application, we varied the number of threads and processes for NGINX and memcached to observe their respective scalability. We specifically evaluate configurations with 8 processes for NGINX and {4,2} threads for memcached, as well as a 4-process configuration for NGINX and {2,1}-thread configurations for memcached. The 3-tier application is primarily bottlenecked by the disk I/O bandwidth of MongoDB, so scaling the number of downstream microservices does not have a significant impact on performance. We evaluate an 8-process configuration for NGINX and and a 2-thread configuration for memcached. Each thread (or process) of every microservice is pinned to a dedicated physical core to avoid interference from the OS scheduler's decisions. Results reported for the real experiments are averaged across 3 runs.

Fig. 5 shows the comparison between the real and simulated two-tier application across thread/process configurations. Across all concurrency settings, μqSim faithfully reproduces the load-latency curve of the real system, including its saturation point. It also accurately captures that giving

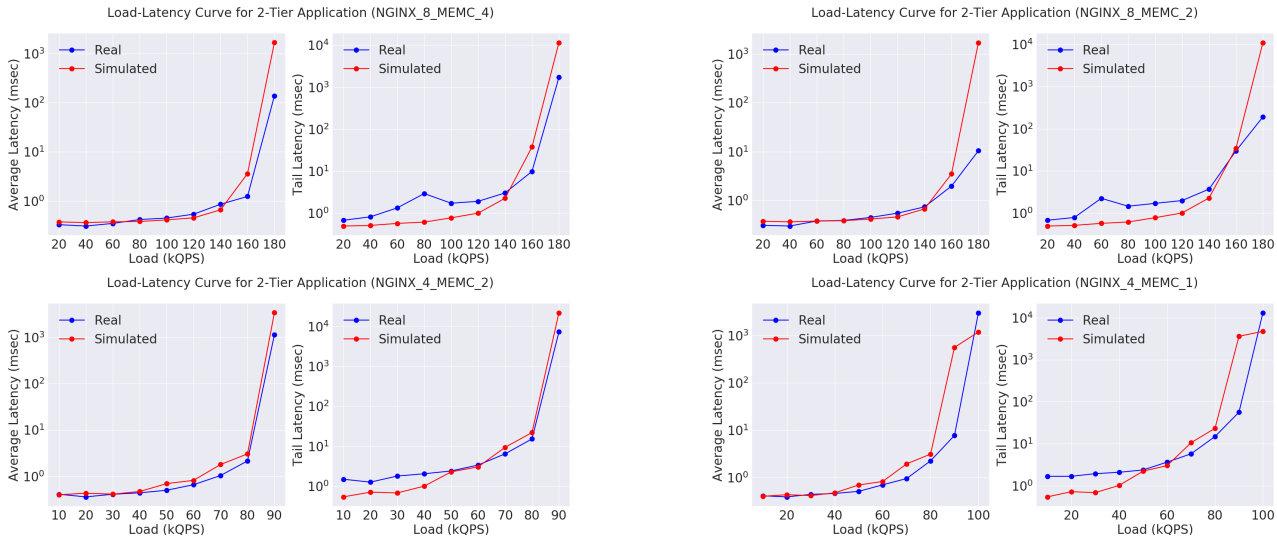

Figure 5: Validation of the two-tier (NGINX-memcached) application across different thread configurations for each microservice.

more resources to memcached does not further improve throughput before saturation, since the limiting factor is NGINX. Additionally, before the 2-tier application reaches saturation, the simulated mean latencies are on average 0.17ms away from the real experiments, and the simulated tail latencies are on average 0.83ms away from the real ones.

Fig. 6 shows the same experiment for the 3-tier service. The results are again consistent between real and simulated performance, with the simulated mean latencies being on average 1.55ms away from real measurements, and the simulated tail latencies deviating by 2.32ms on average.

B. Capturing Load Balancing & Fanout Effects

Load balancing: Load balancers are used in most large-scale cloud environments to fairly divide the load across instances of a scale-out application. We now examine how accurate μqSim is in capturing the performance impact of load balancing. We construct a load balancing scenario using an instance of NGINX as a load-balancing proxy, and several instances of NGINX of the same setup as the scaled-out web-servers behind the proxy. For each client request, the proxy chooses one webserver to forward the request to, in a round-robin fashion. Fig. 7 shows the setup for load balancing, and Fig. 8 shows the load-latency (99^{th} percentile) curves for the real and simulated system. The saturation load scales linearly for a scale out factor of 4 and 8 from 35kQPS to 70kQPS, and sub-linearly beyond that, e.g., for scale-out of 16, saturation happens at 120kQPS as the cores handling the interrupts (`soft_irq` processes) saturate before the NGINX instances. In all cases, the simulator accurately captures the saturation pattern of the real load balancing scenario.

Request fanout: We also experiment with request fanout, which is common in applications with distributed state. In this case, a request only completes when responses from all fanout services have been received [12]. Request fanout is a well-documented source of unpredictable performance in cloud infrastructures, as a single slow leaf node can degrade the performance of the majority of user requests [12], [35]–[37]. As with load-balancing, the fanout experiment uses an NGINX instance as a proxy, which - unlike load balancing - now forwards each request to all NGINX instances of the next tier. We scale the fanout factor from 4 to 16 servers, and assign 1 core and 1 thread to each fanout service. We also dedicate 4 cores to network interrupts. Each requested webpage is 612 bytes in size, and the workload generator is set up in a similar way to the 2- and 3-tier experiments above. The system configuration is shown in Fig. 9 and the load-latency (99^{th} percentile) curve for the real and simulated system is shown in Fig. 8. For all fanout configurations, μqSim accurately reproduces the tail latency and saturation point of the real system, including the fact that as fanout increases, there is a small decrease in saturation load, as the probability that a single slow server will degrade the end-to-end tail latency increases.

C. Simulating RPC Requests

Remote Procedure Calls (RPC) are widely deployed in microservices as a cross-microservice RESTful API. In this section we demonstrate that μqSim can accurately capture the performance of a popular RPC framework, Apache Thrift [25], and in the next section we show that it can faithfully reproduce the behavior of complex microservices using Thrift as their APIs. We set up a simple Thrift client and server; the server responds with a "Hello World" message to each request. Given the lack of application logic in this case, all time goes towards processing the RPC request. The real

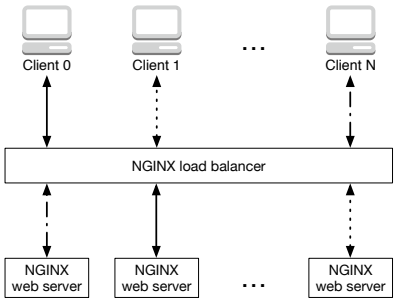

Figure 7: Load balancing in NGINX.

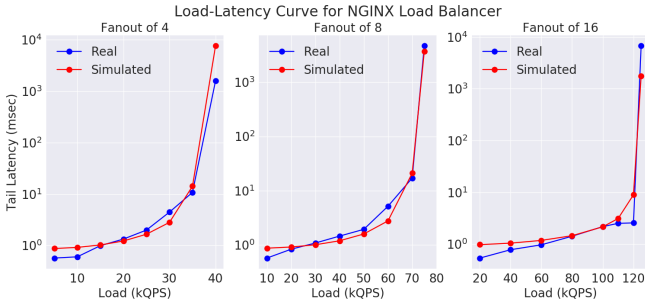

Figure 8: Validation of load balancing results.

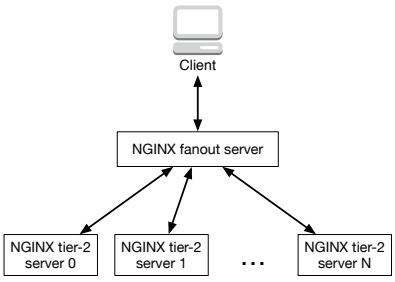

Figure 9: Request fanout in NGINX.

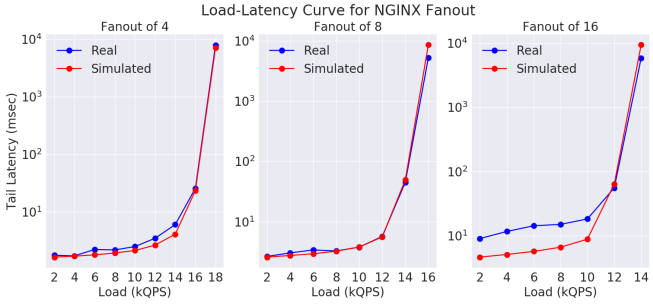

Figure 10: Validation of request fanout impact in NGINX.

and simulated results are shown in Fig.12. In both cases the Thrift server saturates beyond 50kQPS, while the low-load latency does not exceed 100us. Beyond the saturation point the simulator expects latency that increases more gradually with load compared to the real system. The reason for this is that the simulator does not capture timeouts and the associated overhead of reconnections, which can cause the real system's latency to increase rapidly.

D. Simulating Complex Microservices

We have also built a simplified end-to-end application implementing a social network using microservices, illustrated in Fig.11. The service implements a unidirectional, broadcast-style social network, where users can follow each other, post messages, reply publicly or privately to another user, and browse information about a given user. We focus on the later function in this scenario for simplicity. Specifically, the client wants to retrieve a post from a certain user, via the *Thrift Frontend* by specifying a given `userId` and `postId`. Upon receiving this request from the client, the *Thrift Frontend* sends requests to *User Service* and *Post Service*, which search for the user profile and corresponding post respectively. Once the user and post information are received, *Thrift Frontend* extracts any media embedded to the user's post via *Media Service*, composes a response with the user's metadata, post content, and media (if applicable), and returns the response to the client. The user, post, and media objects are stored in the corresponding MongoDB instances, and cached in memcached to lower request latency. All cross-microservice communication in this application happens using Apache Thrift. The comparison between the real and simulated system is shown in Fig. 12b. At low load, μqSim closely matches the latency of the real application, while at high load it saturates at a similar throughput as the real social network service. This application contains a large number of queues and dependent microservices, including applications with fanout, synchronization, and blocking characteristics, showing that μqSim can capture the behavior of complex microservices accurately.

E. Comparison with BigHouse

Fig. 13 shows the comparison of μqSim and BigHouse simulating a single-process NGINX webserver and a 4-thread memcached. In BigHouse both NGINX and memcached are modeled as a single stage server. In μqSim we use the model shown in Fig. 1 for memcached, and we adopt a similar model for NGINX, consisting of two stages: `epoll` and `handler_processing`. For both applications, μqSim captures the real saturation point closely, while BigHouse saturates at much lower load than the real experiments. The reason is that in μqSim the processing time of batching stage `epoll` is amortized across all batched requests, as in the real system. In BigHouse, however, each application is modeled as a single stage so the entire processing time of `epoll` is accounted for in every request, leading to overestimation of the accumulated tail latency.

V. Use Cases

In this section we discuss two case studies that leverage μqSim to obtain performance and efficiency benefits. In the first case study, we experiment with the effect of slow servers

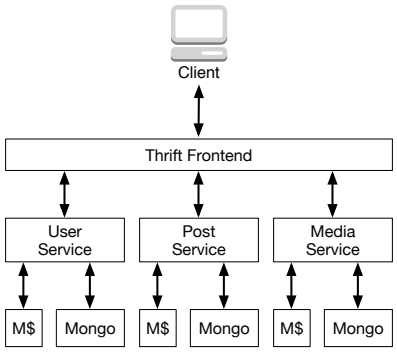

Figure 11: Architecture of the social network microservices application.

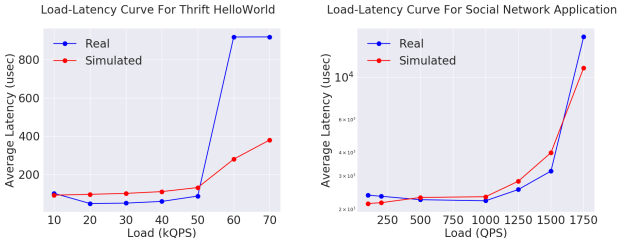

Figure 12: Validation of RPC request simulation using Apache Thrift [25].

in clusters of different sizes in order to reproduce the tail-at-scale effects documented in [12]. In the second study, we design a power management algorithm for interactive microservices, and show its behavior on real and simulated servers. The platforms we use for real experiments are the same as before (Table II).

A. Tail@Scale Effect

As cluster sizes and request fanouts grow, the impact of a small fraction of slow machines are amplified, since a few stragglers dominate tail latency. In this scenario we simulate

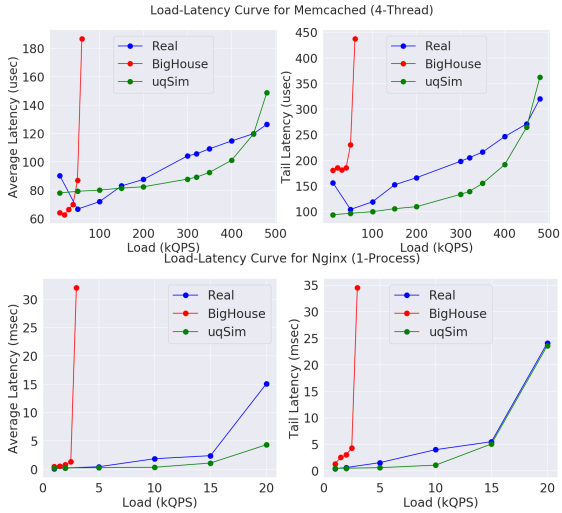

Figure 13: Comparison of μqSim and BigHouse

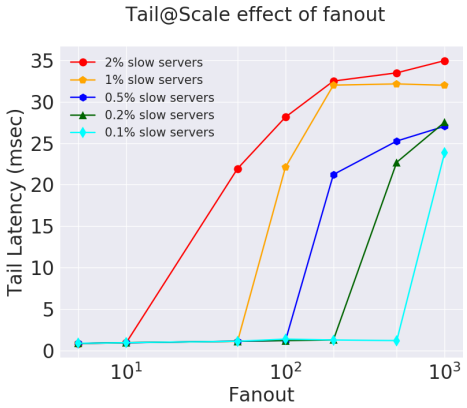

Figure 14: The tail at scale effects of request fanout.

clusters of different sizes, ranging from 5 servers to 1000 servers, in a similar fanout configuration as the one discussed in [12]. Under this setup a user request fans out to all servers in the cluster, and only returns to the user after the last server responds. To capture similar effects as in [12] the application is a simple one-stage queueing system with exponentially distributed processing time, around a 1ms mean. To emulate slow servers, we increase the average processing time of a configurable fraction of randomly-selected servers by $10\times$. Fig. 14 shows the impact of slow servers on tail latency as fanout (cluster size) increases. For the same percentage of slow servers, e.g., 1%, the larger the size of the cluster, the more likely it is that tail latency is defined by the slow machines. Similarly, as the fraction of slow servers increases, so does the probability for high tail latency, complying to the probabilistic expression discussed in [12]. Note that for cluster sizes greater than 100 servers, 1% of slow servers is sufficient to drive tail latency high, consistent with the results in [12].

B. Power Management

The lack of energy proportionality in datacenters is a well-documented problem [4], [36], [38]. Quality-of-service-aware power management is challenging for cloud applications, as resource utilization does not always correlate closely with QoS, resulting in significant increases in tail latency, even when the server is not saturated [36]. Power management is even more challenging in multi-tier applications and microservices, since dependencies between neighboring microservices introduce backpressure effects, creating cascading hotspots and QoS violations through the system. This makes it hard to determine the appropriate frequency setting for each microservice, and to identify which microservice is the culprit of a QoS violation.

Our proposed power management algorithm is based on the intuition that reasoning about QoS guarantees for the end-to-end application requires understanding the interactions between dependent microservices, and how changing the performance requirements of one tier affects the rest

Algorithm 1 Power Management Algorithm

```
while True do
    if time_now − time_prev_cycle < Interval then
        sleep(Interval)
    end if
    if stats[end2end] < Target then
        if stats no_relaxed_than fail_tuples then
            bucket.insert(stats)
        end if
        increase bucket.preference
        if CycleCount > Interval then
            Choose new target bucket
            Choose per-tier QoS
        end if
        Slow down at most 1 tier based on slack
    else
        decrease bucket.preference
        bucket.failing_list.insert(current target)
        Choose new target bucket
        Choose per-tier QoS
        Speed up all tiers with higher latency than target
    end if
end while
```

of the application. We adopt a divide-and-conquer approach by dividing the end-to-end QoS requirement to per-tier QoS requirements, because, as queueing theory suggests, the combination of per-tier state should be recurrent and reproducible, which indicates that as long as the per-tier latencies achieve values that have allowed the end-to-end QoS to be met in the past, the system should be able to recover from a QoS violation.

Based on this intuition, our algorithm divides the tail latency space into a number of buckets, with each bucket corresponding to a given end-to-end QoS range, and classifies the observed per-tier latencies into the corresponding buckets. At runtime, the scheduler picks one per-tier latency tuple from a certain bucket, and uses it as the per-tier QoS target. Different buckets are equally likely to be visited initially, and as the application execution progresses, the scheduler learns which buckets are more likely to meet the end-to-end tail latency requirement, and adjusts the weights accordingly. To refine the recorded per-tier latencies, every bucket also keeps a list of previous per-tier tuples that fail to meet QoS when used as the latency target, and a new per-tier tuple is only inserted if it is no more relaxed than any of the failing tuples of the corresponding bucket. This way the scheduler eventually converges to a set of per-tier QoS requirements that have a high probability to meet the required end-to-end performance target. In order to test whether more aggressive power management settings are acceptable, the scheduler periodically selects a tier with high latency slack to slow down, and observes the change in end-to-end performance. The scheduler only slows down 1 tier at a time, to prevent cascading violations caused by interactions between tiers (like connection pool exhaustion and blocking). The pseudo-code for the power management algorithm is shown in Algo. 1.

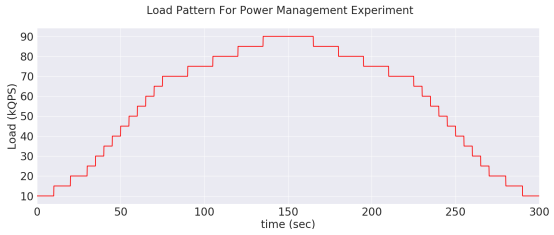

Figure 15: Load fluctuation under the examined diurnal load.

Table III: Power management QoS violation rates.

Decision Intervals	0.1s	0.5s	1s
Simulated System	0.6%	2.2%	5.0%
Real System	1.5%	2.7%	6.0%

We evaluate the power management algorithm above with the 2-tier application both using μqSim and real servers. To highlight the potential of power management, we drive the application with a diurnal input load, shown in Fig. 15. To simulate the impact of power management in μqSim, we adjust the processing time of each execution stage as frequency changes by providing histograms corresponding to different frequencies. We also vary the decision interval of the scheduler, from $0.1s$ to $1s$. Fig.16 shows the tail latency and per-tier frequency over time under different decision intervals, and Table III shows the fraction of time for which QoS is violated.

Unsurprisingly, the real system is slightly more noisy compared to μqSim, due to effects not modeled in the simulator, such as request timeouts, TCP/IP contention, and operating system interference from scheduling and context switching. The lower latency jitter in the simulator also results in less frequent changes in power management decisions. Nonetheless, both systems follow similar patterns in their decision process, and converge to very similar tail latencies. The reason why tail latency in both cases converges to around 2ms despite a 5ms QoS target, is the coarse frequency-voltage granularity of the power management technique we use – DVFS. The discrete frequency settings enabled by DVFS can only lead to discrete processing speeds, and therefore discrete latency ranges. As a result, further lowering frequency to reduce latency slack would result in QoS violations. More fine-grained power management techniques, such as RAPL [36], would help bring the instantaneous tail latency closer to the QoS.

Figure 16: Tail latency and frequency settings when employing the power management mechanism of Algo. 1. We show tail latencies for both the real and simulated systems.

VI. Conclusions

We presented μqSim, a scalable and validated queueing network simulator for interactive microservices. μqSim offers detailed models both for execution phases within a single microservice, and across complex dependency graphs of microservices. We have validated μqSim against applications with few up to many tiers, as well as scenarios of load balancing and request fanout, and showed minimal differences in throughput and tail latency in all cases. Finally, we showed that μqSim can be used to gain insight into the performance effects that emerge in systems of scale larger than what can be evaluated outside a production cloud environment, as well as when using mechanisms, such as power management, that aim to improve the resource efficiency of large-scale datacenters. We plan to open-source μqSim to motivate more work in the field of microservices.

Acknowledgements

We sincerely thank the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was supported in part by NSF grant CNS-1422088, a Facebook Faculty Research Award, a John and Norma Balen Sesquicentennial Faculty Fellowship, and generous donations from GoogleCompute Engine, Windows Azure, and Amazon EC2.

References

[1] L. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.

[2] L. Barroso, "Warehouse-scale computing: Entering the teenage decade," *ISCA Keynote, SJ, June 2011*.

[3] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozych, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of SOCC*, 2012.

[4] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, 2015.

[5] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, USA, 2013.

[6] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proc. of ASPLOS*, Salt Lake City, 2014.

[7] J. Mars and L. Tang, "Whare-map: heterogeneity in "homogeneous" warehouse-scale computers," in *Proceedings of ISCA*, Tel-Aviv, Israel, 2013.

[8] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *Proceedings of EuroSys*, Paris,France, 2010.

[9] R. Nathuji, C. Isci, and E. Gorbatov, "Exploiting platform heterogeneity for power efficient data centers," in *Proceedings of ICAC*, Jacksonville, FL, 2007.

[10] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of MICRO*, Porto Alegre, Brazil, 2011.

[11] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of ISCA*, 2013.

[12] J. Dean and L. A. Barroso, "The tail at scale," in *CACM, Vol. 56 No. 2*.

[13] C. Delimitrou and C. Kozyrakis, "QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon," in *ACM Transactions on Computer Systems (TOCS), Vol. 31 Issue 4*, December 2013.

[14] "Microservices workshop: Why, what, and how to get there." http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.

[15] "The evolution of microservices." https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, 2016.

[16] "Decomposing twitter: Adventures in service-oriented architecture." https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture.

[17] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *Proc. of IISWC*, 2016.

[18] A. Sriraman and T. F. Wenisch, "usuite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018*, pp. 1–12, 2018.

[19] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, (New York, NY, USA), pp. 323–324, ACM, 2018.

[20] N. Kratzke and P.-C. Quint, "Ppbench," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, CLOSER 2016, (Portugal), pp. 223–231, SCITEPRESS - Science and Technology Publications, Lda, 2016.

[21] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

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

[23] Y. Gan, M. Pancholi, D. Cheng, S. Hu, Y. He, and C. Delimitrou, "Seer: Leveraging Big Data to Navigate the Complexity of Cloud Debugging," in *Proceedings of the Tenth USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, July 2018.

[24] Y. Gan, Y. Zhang, K. Hu, Y. He, M. Pancholi, D. Cheng, and C. Delimitrou, "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 2019.

[25] "Apache thrift." https://thrift.apache.org.

[26] 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.

[27] X. Zeng, R. Bagrodia, and M. Gerla, "Glomosim: a library for parallel simulation of large-scale wireless networks," in *ACM SIGSIM Simulation Digest*, vol. 28, pp. 154–161, IEEE Computer Society, 1998.

[28] P. Xie, Z. Zhou, Z. Peng, H. Yan, T. Hu, J.-H. Cui, Z. Shi, Y. Fei, and S. Zhou, "Aqua-sim: An ns-2 based simulator for underwater sensor networks," in *OCEANS 2009, MTS/IEEE biloxi-marine technology for our future: global and local challenges*, pp. 1–7, IEEE, 2009.

[29] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[30] T. Issariyakul and E. Hossain, "Introduction to network simulator 2 (ns2)," in *Introduction to Network Simulator NS2*, pp. 21–40, Springer, 2012.

[31] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[32] L. Kleinrock, "Queueing systems volume 1: Theory," pp. 101-103, 404.

[33] "Nginx." https://nginx.org/en.

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

[35] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)*, August 2015.

[36] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, Minneapolis, MN, 2014.

[37] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. F. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pp. 271–282, 2015.

[38] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," in *Proceedings of the 14th international ASPLOS*, ASPLOS '09, 2009.