μTune: Auto-Tuned Threading for OLDI Microservices

Akshitha Sriraman Thomas F. Wenisch

University of Michigan
akshitha@umich.edu, twenisch@umich.edu

ABSTRACT

Modern On-Line Data Intensive (OLDI) applications have evolved from monolithic systems to instead comprise numerous, distributed microservices interacting via Remote Procedure Calls (RPCs). Microservices face sub-millisecond (sub-ms) RPC latency goals, much tighter than their monolithic counterparts that must meet $\geq$ 100 ms latency targets. Sub-ms–scale threading and concurrency design effects that were once insignificant for such monolithic services can now come to dominate in the sub-ms–scale microservice regime. We investigate how threading design critically impacts microservice tail latency by developing a *taxonomy of threading models*—a structured understanding of the implications of how microservices manage concurrency and interact with RPC interfaces under wide-ranging loads. We develop μ*Tune*, a system that has two features: (1) a novel framework that abstracts threading model implementation from application code, and (2) an automatic load adaptation system that curtails microservice tail latency by exploiting inherent latency trade-offs revealed in our taxonomy to transition among threading models. We study μ*Tune* in the context of four OLDI applications to demonstrate up to 1.9× tail latency improvement over static threading choices and state-of-the-art adaptation techniques.

1 Introduction

On-Line Data Intensive (OLDI) applications, such as web search, advertising, and online retail, form a major fraction of data center applications [113]. Meeting soft real-time deadlines in the form of Service Level Objectives (SLOs) determines end-user experience [21, 46, 55, 95] and is of paramount importance. Whereas OLDI applications once had largely monolithic software architectures [50], modern OLDI applications comprise numerous, distributed microservices [66, 90, 116] like HTTP connection termination, key-value serving [72], query rewriting [48], click tracking, access-control management, protocol routing [25], etc. Several companies, such as Amazon [6], Netflix [1], Gilt [37], LinkedIn [17], and SoundCloud [9], have adopted microservice architectures to improve OLDI development and scalability [144]. These microservices are composed via standardized Remote Procedure Call (RPC) interfaces, such as Google's Stubby and `gRPC` [18] or Facebook/Apache's Thrift [14].

Whereas monolithic applications face $\geq$ 100 ms tail (99^{th}+%) latency SLOs (e.g.,$\sim$300 ms for web search [126, 133, 142, 150]), microservices must often achieve sub-ms (e.g., $\sim$100 μs for protocol routing [151]) tail latencies as many microservices must be invoked serially to serve a user's query. For example, a Facebook news feed service [79] query may flow through a serial pipeline of many microservices, such as (1) `Sigma` [15]: a spam filter, (2) `McRouter` [118]: a protocol router, (3) `Tao` [56]: a distributed social graph data store, (4) `MyRocks` [29]: a user database, etc., thereby placing tight sub-ms latency SLOs on individual microservices. We expect continued growth in OLDI data sets and applications to require composition of ever more microservices with increasingly complex interactions. Hence, the pressure for better microservice latency SLOs continually mounts.

Threading and concurrency design have been shown to critically affect OLDI response latency [76, 148]. But, prior works [71] focus on monolithic services, which typically have $\geq$ 100 ms tail SLOs [111]. Hence, sub-ms–scale OS and network overheads (e.g., a context switch cost of 5-20 μs [101, 141]) are often insignificant for monolithic services. However, sub-ms–scale microservices differ intrinsically: spurious context switches, network/RPC protocol delays, inept thread wakeups, or lock contention can dominate microservice latency distributions [39]. For example, even a single 20μs spurious context switch implies a 20% latency penalty for a request to a 100 μs SLO protocol routing microservice [151]. Hence, prior conclusions must be revisited for the microservice regime [49].

In this paper, we study how threading design affects mi-

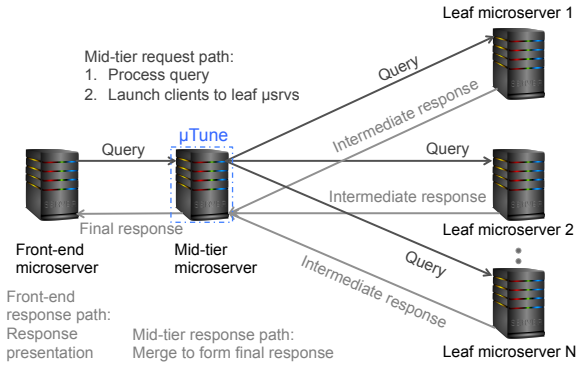


Figure 1: A typical OLDI application fan-out.

croservice tail latency, and leverage these design effects to dynamically improve tails. We develop a system called *μTune*, which features a framework that builds upon open-source RPC platforms [18] to enable microservices to abstract threading model design from service code. We analyze a *taxonomy of threading models* enabled by *μTune*. We examine synchronous or asynchronous RPCs, in-line or dispatched RPC handlers, and interrupt- or poll-based network reception. We also vary thread pool sizes dedicated to various purposes (network polling, RPC handling, response execution). These design axes yield a rich space of microservice architectures that interact with the underlying OS and hardware in starkly varied ways. These threading models often have surprising OS and hardware performance effects including cache locality and pollution, scheduling overheads, and lock contention.

We study *μTune* in the context of four full OLDI services adopted from μSuite [134]. Each service comprises sub-ms microservices that operate on large data sets. We focus our study on *mid-tier microservers*: widely-used [50] microservices that accept service-specific RPC queries, fan them out to leaf microservers that perform relevant computations on their respective data shards, and then return results to be integrated by the mid-tier microserver, as illustrated in Fig. 1. The mid-tier microserver is a particularly interesting object of study since (1) it acts as both an RPC client and an RPC server, (2) it must manage fan-out of a single incoming query to many leaf microservers, and (3) its computation typically takes tens of microseconds, about as long as OS, networking, and RPC overheads.

We investigate threading models for mid-tier microservices. Our results show that the best threading model depends critically on the offered load. For example, at low loads, models that poll for network traffic perform best, as they avoid expensive OS thread wakeups. Conversely, at high loads, models that separate network polling from RPC execution enable higher service capacity and blocking outperforms polling for incoming network traffic as it avoids wasting precious CPU on fruitless poll loops.

We find that the relationship between optimal threading model and service load is complex—one could not expect a developer to pick the best threading model a priori. So, we build an intelligent system that uses offline profiling to automatically adapt to time-varying service load.

μTune's second feature is an adaptation system that determines load via event-based load monitoring and tunes both the threading model (polling vs. blocking network reception; inline vs. dispatched RPC execution) and thread pool sizes in response to load changes. *μTune* improves tail latency by up to 1.9× over static peak load-sustaining threading models and state-of-the-art adaptation techniques, with $< 5\%$ mean latency and instruction overhead. Hence, *μTune* can be used to dynamically curtail sub-ms–scale OS/network overheads that dominate in modern microservices.

In summary, we contribute:

- A *taxonomy of threading models*: A structured understanding of microservice threading models and their implications on performance.
- *μTune's framework* [1] for developing microservices, which supports a wide variety of threading models.
- *μTune's load adaptation system* for tuning threading models and thread pools under varying loads.
- A detailed performance study of OLDI services' key tier built with *μTune*: the mid-tier microserver.

2 Motivation

We motivate the need for a threading taxonomy and adaptation systems that respond rapidly to wide-ranging loads.

Many prior works have studied leaf servers [63, 107, 108, 123, 142, 143], as they are typically most numerous, making them cost-critical. *Mid-tier* servers [68, 98], which manage both incoming and outgoing RPCs to many clients and leaves, perhaps face greater tail latency optimization challenges, but have not been similarly scrutinized. Their network fan-out multiplies underlying software stack interactions. Hence, performance and scalability depend critically on mid-tier threading model design.

Expert developers extensively tune critical OLDI services via trial-and-error or experience-based intuition [84]. Few services can afford such effort; for the rest, we must appeal to software frameworks and automatic adaptation to improve performance. *μTune* aims to empower small teams to develop performant mid-tier microservices that meet latency goals without enormous tuning efforts.

The need for a threading model taxonomy. We develop a structured understanding of rational design options for architecting microservices' OS/network interactions in the form of a *taxonomy of threading models*. We

[1] Available at https://github.com/wenischlab/MicroTune

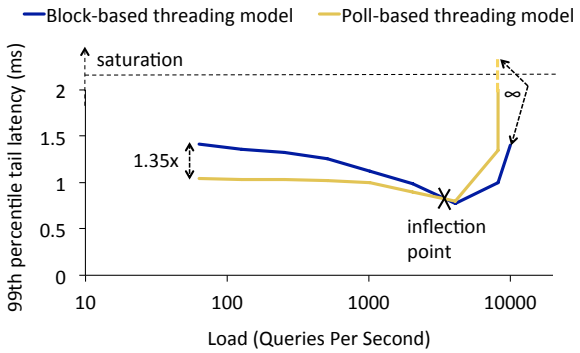


Figure 2: $99^{th}\%$ tail latency for an RPC handled by a block-based & poll-based model: poll-based model improves latency by 1.35x at low load, and saturates at high load.

study these models' latency effects under diverse loads to offer guidance on when certain models perform best.

Prior works [69,83,84,146,148] broadly classify monolithic services as: thread-per-request *synchronous* or event-driven *asynchronous*. We note threading design space dimensions beyond these coarse-grain designs. We build on prior works' insights, such as varying parallelism to reduce tail latency [76], to consider a more diverse taxonomy and spot sub-ms performance concerns.

The need for automatic load adaptation. Subtle changes in a microservice's OS interaction (e.g., how it accepts incoming RPCs) can cause large tail latency differences. For example, Fig. 2 depicts the $99^{th}\%$ tail latency for a sample RPC handled by an example mid-tier microservice as a function of load. We use a mid-tier microserver with 36 physical cores that dispatches requests received from the front-end to a group of worker threads which then invoke synchronous calls to the leaves. The yellow line is the tail latency when we dedicate a thread to poll for incoming network traffic in a CPU-unyielding spin loop. The blue line blocks on the OS socket interface awaiting work to the same RPC handler. We see a stark load-based performance inflection even for these simple designs. At low load, a poll-based model gains 1.35× latency as it avoids OS thread wakeups. Conversely, at high load, fruitless poll loops waste precious CPU that might handle RPCs. The poll-based model becomes saturated, with arrivals exceeding service capacity and unbounded latency growth. Blocking-based models conserve CPU and are more scalable.

We assert that such design trade-offs are not obvious: no single threading model is optimal at all loads, and even expert developers have difficulty making good choices. Moreover, most software adopts a threading model at design time and offers no provision to vary it at runtime.

A microservice framework. Instead, we propose a novel microservice framework in μ*Tune* that abstracts threading design from the RPC handlers. The μ*Tune* system adapts to load by choosing optimal threading models and thread pool sizes dynamically to reduce tail latency.

μ*Tune* aims to allow a microservice to be built once and be scalable across wide-ranging loads. Many OLDI services experience drastic diurnal load variations [79]. Others may face "flash crowds" that cause sudden load spikes (e.g., intense traffic after a major news event). New OLDI services may encounter explosive customer growth that surpasses capacity planning (e.g., the meteoric launch of Pokemon Go [31]). Supporting load scalability over many orders of magnitude in a single framework facilitates rapid scale-up of a popular new service.

3 A Taxonomy of Threading Models

A *threading model* is a software system design choice that governs how responsibility for key application functionality will be divided among threads and how the application will achieve request concurrency. Threading models critically impact the service's throughput, latency, scalability, and programmability. We characterize preemptive instead of co-operative (e.g., node.js [140]) threading models.

3.1 Key dimensions

We identify three threading model dimensions and discuss their programmability and performance implications.

Synchronous vs. asynchronous communication. Prior works have identified synchronous vs. asynchronous communication as a key design choice in monolithic OLDI services [69,83,84,146,148]. Synchronous models map a request to a single thread throughout its lifetime. Request state is implicitly tracked via the thread's PC and stack—programmers simply maintain request state in automatic variables. Threads use blocking I/O to await responses from storage or leaf nodes. In contrast, asynchronous models are event-based—programmers explicitly define state machines for a request's progress [83]. Any ready thread may progress a request upon event reception; threads and requests are not associated.

Programmability: Synchronous models are typically easier to program, as they entail writing straight-forward code without worrying about elusive concurrency-related subtleties. Conversely, asynchronous models require explicit reasoning about request state, synchronization, and races. Ensuing code is often characterized as "spaghetti"—control flow is obscured by callbacks, continuations, futures, promises, and other sophisticated paradigms. Due to this vast programmability gap, we spent three weeks implementing synchronous and four months for asynchronous models.

Performance: As synchronous models await leaf responses before progressing new requests, they face request/response queuing delays, producing worse response latencies and throughput than asynchronous [69,114,146]. Adding more synchronous threads can allay queuing, but

can induce secondary bottlenecks, such as cache pollution, lock contention, and scheduling/thread wakeup delays.

Synchronous apps: Azure SQL [5], Google Cloud SQL's Redmine [10, 100], MongoDB replication [28]

Asynchronous apps: Apache [3], Azure blob storage [27], Redis replication [34], Server-Side Mashup [105], CORBA Model, Aerospike [2]

In-line vs. dispatch-based RPC processing. In in-line models, a single thread manages the entire RPC lifetime, from the point where it is accepted from the RPC library until its response is returned. Dispatch-based models separate responsibilities between network threads, which accept new requests from the underlying RPC interface, and worker threads, which execute RPC handlers.

Programmability: In-line models are simple; thread pools block/poll on the RPC arrival queue and execute an RPC completely before receiving another. Dispatched models are more complex; RPCs are explicitly passed from network to worker threads via thread-safe queues.

Performance: In-line models avoid the explicit state hand-off and thread-hop to pass work from network to worker threads. Hence, they are efficient at low loads and for short requests, where dispatch overheads dominate service times. But, if a single thread cannot sustain the service load, multiple threads contending to accept work typically outweighs hand-off costs, which can be carefully honed. In-line models are prone to high queuing, as each thread processes whichever request it receives. In contrast, dispatched models can explicitly prioritize requests.

In-line apps: Redis [41, 58], MapReduce workers [64]

Apps that dispatch: IBM's WebSphere for z/OS [22, 81], Oracle's EDT image search [20], Mule ESB [12], Malwarebytes [19], Celery for RabbitMQ and Redis [11], Resque [35] and RQ [36] Redis queues, NetCDF [74]

Block- vs. poll-based RPC reception. While the synchronous and in-line dimensions address outgoing RPCs, the block vs. poll dimension concerns incoming RPCs. In block-based models, threads await new work via blocking system calls, yielding CPU if no work is available. Threads block on I/O interfaces (e.g., `read()` or `epoll()` system calls) awaiting work. In poll-based models, a thread spins in a loop, continuously looking for new work.

Performance: The poll vs. block trade-off is intrinsic: polling reduces latency, while blocking frees a waiting CPU to perform other work. Polling incurs lower latency as it avoids OS thread wakeups [106] to which blocking is prone. But, polling wastes CPU time in fruitless poll loops, especially at low loads. Yet, many latency-sensitive services opt to poll [34], perhaps solely to avoid unexpected hardware or OS actions, such as a slow transition to a low-power mode [51]. Many polling threads can contend to cause pathologically poor performance [88].

Apps that block: Redis BLPOP [7]

Apps that poll: Intel's DPDK Poll Driver [32], Redis replication [34], Redis LPOP [24], DoS attacks and defenses [117, 125, 132], GCP Health Checker [38]

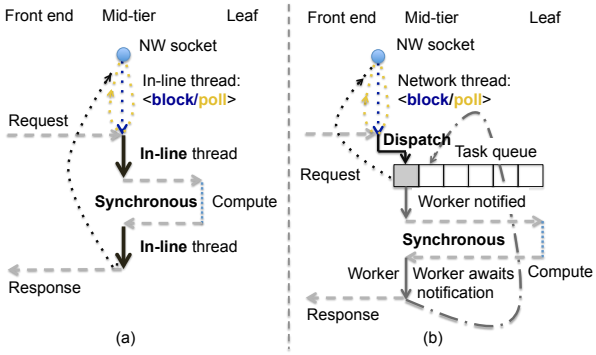


Figure 3: Execution of an RPC by (a) SIB/SIP (b) SDB/SDP

These three dimensions lead to eight mid-tier threading models. We also vary thread pool sizes for these models.

3.2 Synchronous models

In synchronous models, we create maximally sized thread pools on start-up and then "park" extraneous threads on condition variables, to rapidly supply threads as needed without `pthread_create()` call overheads. To simplify our figures, we omit parked threads from them.

The main thread handling each RPC uses *fork-join* parallelism to fan concurrent requests out to many leaves. The main thread wakes a parked thread to issue each outgoing RPC, blocking on its reply. As replies arrive, these threads decrement a shared atomic counter before parking on a condition variable to track the last reply. The last reply signals the main thread to execute the continuation that merges leaf results and responds to the client.

We next detail each synchronous model with respect to a single RPC execution. For simplicity, our figures show a three-tier service with a single client, mid-tier, and leaf.

Synchronous In-line Block (SIB). This model is the simplest, having only a single thread pool (Fig. 3(a)). *In-line* threads *block* on network sockets awaiting work, and then execute a received RPC to completion, signalling parked threads for outgoing RPCs as needed. The thread pool must grow with higher load.

Synchronous In-line Poll (SIP). SIP differs from SIB in that threads poll for new work using non-blocking APIs (Fig. 3(a)). SIP avoids blocked thread wakeups when work arrives, but, each in-line thread fully utilizes a CPU.

Synchronous Dispatch Block (SDB). SDB comprises two thread pools (Fig. 3(b)). The *network threads* block on socket APIs awaiting new work. But, rather than executing the RPC, they *dispatch* the RPC to a *worker* thread pool by using producer-consumer task-queues and signalling condition variables. Workers pull requests from task queues, and then process them much like the prior in-line threads (i.e., forking for fan-out and issuing syn-

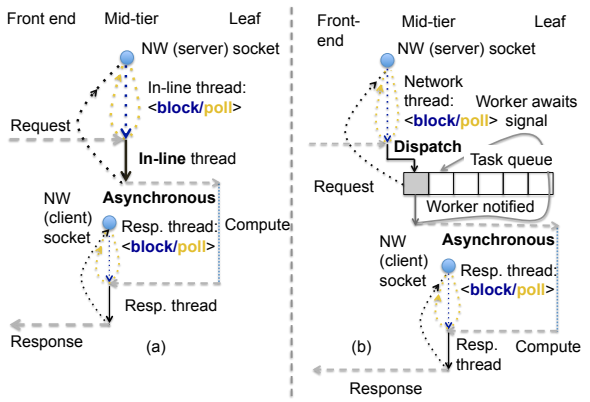


Figure 4: Execution of an RPC by (a) AIB/AIP (b) ADB/ADP

chronous leaf requests). A worker sends the RPC reply to the front-end, before blocking on the condition variable to await new work. Both network and worker pool sizes are variable. Concurrency is limited by the worker pool size. Typically, a single network thread is sufficient.

SDB restricts incoming socket interactions to the network threads, which improves locality; RPC and OS interface data structures do not migrate among threads.

Synchronous Dispatch Poll (SDP). In SDP, network threads *poll* on front-end sockets for new work (Fig. 3(b)).

3.3 Asynchronous models

Asynchronous models differ from synchronous in that they do not tie an execution thread to a specific RPC—all RPC state is explicit. Such models are event-based—an event, such as a leaf request completion, arrives on any thread and is matched to its parent RPC using shared data structures. So, any thread may progress any RPC through its next execution stage. This approach requires drastically fewer thread switches during an RPC lifetime. For example, leaf request fan-outs require a simple for loop, instead of a complex *fork-and-wait*.

To aid non-blocking calls to both leaves and front-end servers, we add another thread pool that exclusively handles leaf server responses—the *response* thread pool.

Asynchronous In-line Block (AIB). AIB (Fig. 4(a)) uses in-line threads to handle incoming front-end requests, and response threads to execute leaf responses. Both thread pools block on their respective sockets awaiting new work. An in-line thread initializes a data structure for an RPC, records the number of leaf responses it expects, records a functor for the continuation to execute when the last response returns, and then fans leaf requests out in a simple for loop. Responses arrive (potentially concurrently) on response threads, which record their results in the RPC data structure and count down until the last response arrives. The final response invokes the continuation to merge responses and complete the RPC.

Asynchronous In-line Poll (AIP). In AIP, in-line and

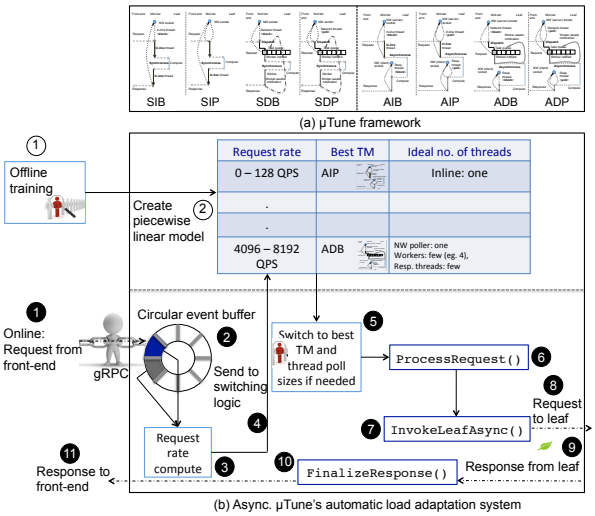


Figure 5: µTune: system design

response threads *poll* their respective sockets (Fig. 4(a)).

Asynchronous Dispatch Block (ADB). In ADB, dispatch enables network thread concentration, improving locality and socket contention (Fig. 4(b)). Like SDB, network and worker threads accept and execute RPCs, respectively. Response threads count-down and merge leaf responses. We do not explicitly dispatch responses, as all but the last response thread do negligible work (stashing a response packet and decrementing a counter). All three thread pools vary in size. Typically, one network thread is sufficient, while the other pools must scale with load.

Asynchronous Dispatch Poll (ADP). Network and response threads *poll* for new work (Fig. 4(b)).

4 µTune: System Design

µTune has two features: (a) an implementation of all eight threading models, abstracting RPC (OS/network interactions) within the framework (Fig. 5(a)); and (b) an adaptation system that judiciously tunes threading models under changing load (Fig. 5(b)). *µTune*'s system design challenges include (1) offering a simple interface that abstracts threading from service code, (2) quick load shift detection for efficient dynamic adaptation, (3) adept threading models switches, and (4) sizing thread pools without thread creation, deletion, or management overheads. We discuss how *µTune*'s design meets these challenges.

Framework. *µTune* abstracts the threading model boiler-plate code from service-specific RPC implementation details, wrapping the underlying RPC API. *µTune* enables characterizing the pros and cons of each model.

µTune offers a simple abstraction where service-specific code must implement RPC execution interfaces. For synchronous modes, the service must supply a `ProcessRequest()` method per RPC. `ProcessRequest()` is invoked by in-line or worker threads. This method pre-

pares a concurrent outgoing leaf RPC batch and passes it to `InvokeLeaf()`, which fans it out to leaf nodes. `InvokeLeaf()` returns to `ProcessRequest()` after receiving all leaf replies. The `ProcessRequest()` continuation merges replies and forms a response to the client.

For asynchronous modes, *µTune*'s interface is slightly more complex. Again, the service must supply `ProcessRequest()`, but, it must explicitly represent RPC state in a shared data structure. `ProcessRequest()` may make one/more calls to `InvokeLeafAsync()`. These calls are passed an outgoing RPC batch, a tag identifying the parent RPC, and a `FinalizeResponse()` callback. The tags enable request-response matching. The last arriving response thread invokes `FinalizeReponse()`, which may access the RPC data structure and response protocol buffers from each leaf. A developer must ensure thread-safety. `FinalizeResponse()` may be invoked any time after `InvokeLeafAsync()`, and may be concurrent with `ProcessRequest()`. Reasoning about races is the key challenge of asynchronous RPC implementation.

Automatic load adaptation. A key feature of *µTune* is its ability to automatically select among threading models in response to load, thereby relieving developers of the burden of selecting a threading model a priori.

Synchronous vs. asynchronous microservices have a major programmability gap. Although *µTune*'s framework hides some complexity, it is not possible to switch automatically and dynamically between synchronous and asynchronous modes, as their API and application code requirements necessarily differ. If an asynchronous implementation is available, it will outperform its synchronous counterpart. So, we build *µTune*'s adaption separately for synchronous and asynchronous models.

µTune picks the latency-optimal model among the four options (in-line vs. dispatch; block vs. poll) and tunes thread pool sizes dynamically with load. *µTune* aims to curtail $99^{th}\%$ tail latency. It monitors service load and (a) picks a latency-optimal threading model, then (b) scales thread pools by parking/unparking threads. Both adaptations use profiles generated during an offline training phase. We describe the training and adaptation steps shown in Fig. 5(b).

Training phase. (1) During offline characterization, we use a synthetic load generator to drive specific load levels for sustained intervals. During these intervals, we vary threading model and thread pool sizes and observe $99^{th}\%$ tail latencies. The load generator then ramps load incrementally, and we re-characterize at each load step. (2) *µTune* then builds a piece-wise linear model relating offered load to observed tail latency at each load level.

Runtime adaptation. (1) *µTune* uses event-based windowing to monitor loads offered to the mid-tier at runtime. (2) *µTune* records each request's arrival timestamp in a circular buffer. (3) It then estimates the inter-arrival rate by using the circular buffer's size, and youngest and oldest recorded timestamps. The adaptation system's responsiveness can be tuned by adjusting the circular buffer's size. Careful buffer size tuning can ensure quick, efficient adaptation by avoiding oscillations triggered by outliers. Event-based monitoring can quickly detect precipitous load increases. (4) The inter-arrival rate estimate is then fed as input to the switching logic that interpolates within the piece-wise linear model to estimate tail latency for each configuration under each model and thread pool size. (5) *µTune* then transitions to the predicted lowest latency threading model. *µTune* transitions by "parking" the current threading model and "unparking" the newly selected model using its framework abstraction and condition variable signaling, to (a) alternate between poll/block socket reception, (b) process requests in-line or via predefined task queues that dispatch requests to workers, or (c) park/unpark various thread pools' threads to handle new requests. Successive asynchronous requests invoke the (6) `ProcessRequest()`, (7) `InvokeLeafAsync()`, and (10) `FinalizeResponse()` pipeline as dictated by the new threading model. In-flight requests during transitions are handled by the earlier model.

5 Implementation

Framework. *µTune* builds upon Google's open-source `gRPC` [18] library, which uses *protocol buffers* [33]—a language-independent interface definition language and wire format—to exchange RPCs. *µTune*'s mid-tier framework uses `gRPC`'s C++ APIs: (1) `Next()` and `AsyncNext()` with a zero second timeout are used to respectively block or poll for client requests, (2) `RPCName()` and `AsyncRPCName()` are called via `gRPC`'s `stub` object to send requests to leaves. *µTune*'s asynchronous models explicitly track request state using finite state machines. Asynchronous models' response threads call `Next()` or `AsyncNext()` for block- or poll-based receive.

µTune uses `AsyncRPCName()` to handle asynchronous clients. For asynchronous *µTune*, leaves must use `gRPC`'s `Next()` APIs to accept requests through explicitly managed completion queues; for synchronous, the leaves can use underlying synchronous `gRPC` abstractions.

Using *µTune*'s framework to build a new microservice is simple, as only a few service specific functions must be defined. We took $\sim$2 days for each service in Sec. 6.

Automatic load adaptation. We construct the piecewise linear model of tail latency by averaging five 30s measurements of each threading model-thread pool pair at varying loads. *µTune*'s load detection relies on a thread-safe circular buffer built using scoped locks and condition variables. The circular buffer capacity is tuned to quickly detect load transients while avoiding oscillation. We use a 5-entry circular buffer in all experiments. *µTune*'s switching logic uses C++ atomics and condition variables to

switch among threading models seamlessly. μ*Tune*'s adaptation code spans 2371 LOC of C++.

6 Experimental Setup

We characterize threading models in the context of four information retrieval OLDI applications' mid-tier and leaf microservices adopted from μSuite [134].

HDSearch. `HDSearch` performs content-based image similarity search by matching nearest neighbors (NN) in a high-dimensional feature space. It serves a 500K image corpus from Google's Open Images data set [30]. Each image is indexed via a 2048-dimensional feature vector created using Google's Inception V3 model [136] implemented in `TensorFlow` [42]. `HDSearch` locates response images whose feature vectors are near the query's [65,96].

Mid-tier microservice. Modern k-NN libraries use indexing structures, such as Locality-Sensitive Hash (LSH) tables, kd-trees, or k-means, to reduce exponentially the search space relative to brute-force linear search [44, 52, 75,85,110,129,137–139]. `HDSearch`'s mid-tier uses LSH (an accurate and fast algorithm [45, 62, 67, 131]) via an open-source k-NN library called Fast Library for Approximate Nearest Neighbors (FLANN) [115]. The mid-tier's LSH tables store {leaf-server, point id} tuples indicating feature vectors in the leaf's data shards. While executing RPCs, the mid-tier probes its in-memory LSH tables to gather potential NNs. It then sends RPCs with potential NN point IDs to the leaves. Leaves compute distances to return a distance-sorted list. The mid-tier merges leaf responses to return the k-NN across all shards.

Leaf microservice. The leaf's distance computations are embarrassingly parallel, and can be accelerated with SIMD, multi-threading, and distributed computing [65]. We employ all techniques. We distribute distance computations over multiple leaves until the distance computation time and network communication overheads are roughly balanced. Hence, the mid-tier's latency, and its ability to fan out RPCs quickly, becomes critical: the mid-tier microservice and network overheads limit the leaf microservice's scalability. Leaves compare query feature vectors against point lists received from the mid-tier using the high-accuracy Euclidean distance metric [75].

Router. `Router` performs replication-based protocol routing for scaling fault-tolerant key-value stores. Queries are `get` or `set` requests. `Gets` contain keys, and return the corresponding value. `Sets` contain key-value pairs, and return a `set` completion acknowledgement. `Get` and `set` query distributions mimic YCSB's Workload A [59] (1:1 ratio). Queries are from a "Twitter" data set [71].

Mid-tier microservice. The mid-tier uses SpookyHash [8] to distribute keys uniformly across leaf microservers and route `get` and `set` queries. `Router` replicates data for better availability, allowing the same data to reside on several leaves. The mid-tier routes `sets` to all replicas and distributes `gets` among replicas. The mid-tier merges leaf responses and sends them to the client.

Leaf microservice. The leaf microserver builds a `gRPC`-based communication wrapper around a memcached [72] instance, exporting `get` and `set` RPCs.

Set Algebra. `Set Algebra` performs document search by intersecting posting lists. It searches a corpus of 4.3 million WikiText documents in Wikipedia [40] sharded uniformly across leaf microservers, to identify documents containing all search terms. Leaf microservers index posting lists for each term in their shard of the document corpus. Stop words determined by collection frequency [149] are excluded from the term index to reduce leaf computation. Search queries (typically a series of ≤ 10 words [4]) are synthetically generated based on the probability of word occurrences in Wikipedia [40].

Mid-tier microservice. The mid-tier forwards client queries containing search terms to the leaf microservers, which then return intersected posting lists to the mid-tier for their respective shards. The mid-tier aggregates the per-shard posting lists and returns their union to the client.

Leaf microservice. Leaves look up posting lists for all search terms and then intersect the sorted lists. The resulting intersection is returned to the mid-tier.

Recommend. `Recommend` is a recommendation service that performs user-based collaborative filtering on a data set of 10K {user, item, rating} tuples—derived from the `MovieLens` movie recommendation data set [78]—to predict a user's rating for an item. The data set is sharded equally among leaves. `Recommend` uses a fast, flexible open-source ML library called `mlpack` [60] to perform collaborative filtering using matrix decomposition.

Mid-tier microservice. The mid-tier gets {user, item} query pairs and forwards them to the leaves. Item ratings sent by the leaves are averaged and sent to the client.

Leaf microservice. Leaves perform collaborative filtering on a pre-composed matrix of {user,item,rating} tuples. Rating predictions are then sent to the mid-tier.

We use a load generator that mimics many clients to send queries to each mid-tier microservice under controlled load scenarios. It operates in a closed-loop mode while measuring peak sustainable throughput. We measure end-to-end (across all microservices) 99^{th}% latency by operating the load generator in open-loop mode with Poisson inter-arrivals [57]. The load generator runs on separate hardware and we validated that the load generator and network bandwidth are not performance bottlenecks.

Our distributed system has a load generator, a mid-tier microservice, and (1) four-way sharded leaf microservice for `HDSearch`, `Set Algebra`, and `Recommend` and (2) 16-way sharded leaf microservice with three replicas for `Router`. The hardware configuration of our measurement setup is in Table 1. The leaf microservers run within

Table 1: Mid-tier microservice hardware specification.

Processor	Intel Xeon E5-2699 v3 "Haswell"
Clock frequency	2.30 GHz
Cores / HW threads	36 / 72
DRAM	500 GB
Network	10Gbit/s
Linux kernel version	3.19.0

Linux `tasksets` limiting them to 20 logical cores for `HDSearch`, `Set Algebra`, and `Recommend` and 5 logical cores for `Router`. Each microservice runs on a dedicated machine. The mid-tier is not CPU bound; saturation throughput is limited by leaf server CPU.

To test the effectiveness of *µTune*'s load adaptation system and measure its responsiveness to load changes, we construct the following load generator scenarios. (1) *Load ramp*: We increase offered load in discrete 30s steps from 20 Queries Per Second (QPS) up to a microservice-specific near-saturation load. (2) *Flash crowd*: We increase load suddenly from 100 QPS to 8K/13K QPS. In addition to performance metrics measured by our load generator, we also report OS and microarchitectural statistics. We use Linux's `perf` utility to profile the number of cache misses and context switches incurred by the mid-tier microservice. We use Intel's HITM (hit-Modified) PEBS coherence event to detect true sharing of cache lines; an increase in HITM events indicates a corresponding increase in lock contention [109]. We measure thread wakeup delays (reported as latency histograms) using the BPF run queue (scheduler) latency tool [23].

7 Evaluation

We first characterize our threading models. We then compare *µTune* to state-of-the-art adaptation systems.

7.1 Threading model characterization

We explore microservice threading models by first comparing synchronous vs. asynchronous performance. We then separately explore trade-offs among the synchronous and asynchronous models to report how the latency-optimal threading model varies with load.

7.1.1 Synchronous vs. Asynchronous

The synchronous vs. asynchronous trade-off is one of programmability vs. performance. It would be unusual for a development team to construct both microservice designs; if the team invests in the asynchronous design, it will almost certainly be more performant. Still, our performance study serves to quantify this gap.

Saturation throughput. We record saturation throughput for the "best" threading model at saturation (SDB/ADB). In Fig. 6, we see that the greater asynchronous efficiency improves saturation throughput for *µTune*'s asynchronous models, a 42% mean throughput

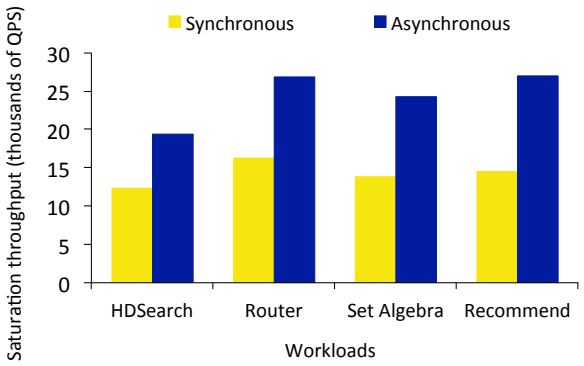


Figure 6: Sync. vs. async. saturation throughput: async. does better by a mean 42%.

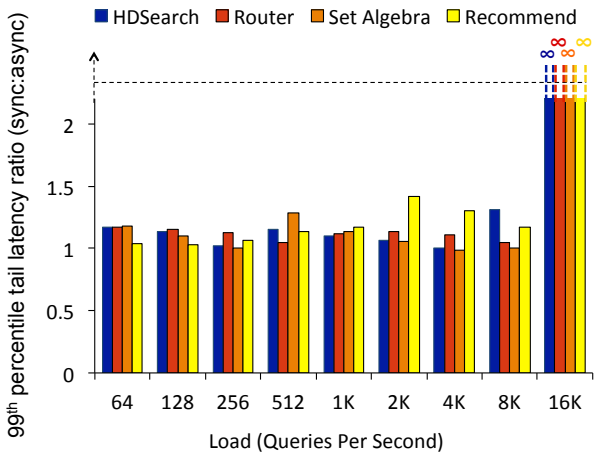


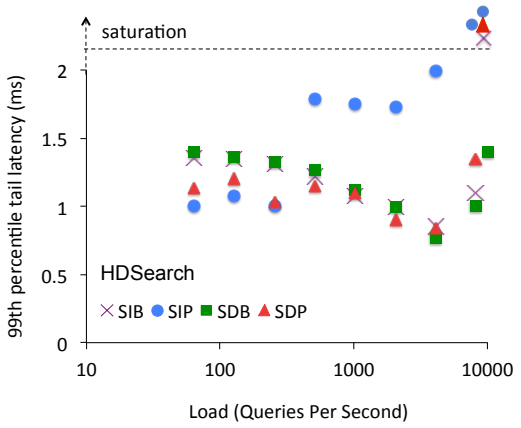
Figure 7: Best sync:async tail latency ratio: async. is faster by a mean 12% at sync.-achievable loads & infinitely faster at high loads.

boost across all services. But, we spent 5× more effort to build, debug, and tune the asynchronous models.

Tail latency. Latency cannot meaningfully be measured at saturation, as the offered load is unsustainable and queuing delays grow unbounded. So, we compare tail latencies at load levels from 64 QPS up to synchronous saturation. In Fig. 7, we show the best sync-to-async ratio of 99^{th}% tail latency across all threading models and thread pool sizes at each load level; we study inter-model latencies later. We find asynchronous models improve tail latency up to $\sim 1.3\times$ (mean of $\sim 1.12\times$) over synchronous models (for loads that synchronous models can sustain; i.e., $\leq$ 8K). This substantial tail latency gap arises because asynchronous models prevent long queuing delays.

7.1.2 Synchronous models

We study the tail latency vs. load trade-off for services built with *µTune*'s synchronous models. We show a cross-product of the threading taxonomy across loads for `HDSearch` in Fig. 8. Each data point is the best 99^{th}% tail latency for that threading model and load based on an exhaustive thread pool size search. Points above the dashed



QPS	64	128	256	512	1024	2048	4096	8192	10K
SIB	1.4	1.3	1.3	**1**	**1**	**1**	1.1	1.1	∞
SIP	**1**	**1**	**1**	1.6	1.6	1.9	2.6	∞	∞
SDB	1.4	1.3	1.3	1.1	1.1	1.1	**1**	**1**	**1**
SDP	1.2	1.1	**1**	**1**	**1**	**1**	1.1	1.4	∞

Figure 8: Graph: Latency vs. load trade-off for `HDSearch` sync. models. Table: Latencies at each load normalized to the best latency for that load—No threading model is always the best.

line are in saturation, where tail latencies are very high and meaningless. The table reports the same graph data with each load latency normalized to the best latency for that load, which is highlighted in blue. We omit graphs for other applications as they match the `HDSearch` trends.

We make the following observations:

SDB enables highest load. SDB, with a single network thread and a large worker pool of 50 threads is the only model that sustains peak loads ($\geq$ 10K QPS). SDB is best at high loads as (1) its worker pool has enough concurrency so that leaf microservers, rather than the mid-tier, pose the bottleneck; and (2) the single network thread is enough to accept and dispatch the offered load. SDB outperforms SDP at high load as polling consumes CPU in fruitless poll loops. For example, at 10,000 QPS, the mid-tier microserver receives one query every 100 microseconds. In SDP, poll loops are often shorter than 100 microseconds. Hence, some poll loops that do not retrieve any requests are wasted work and may delay critical work scheduling, such as RPC response processing. Under SDB, the CPU time wasted in empty poll loops can instead be used to progress an ongoing request.

SIP has lowest latency at low load. While SDB sustains peak loads, it is latency-suboptimal at low loads. SIP offers 1.4× better low-load tail latency by avoiding up to two OS thread wakeups relative to alternative models: (1) network thread wakeups via interrupts on query arrivals, and (2) worker wakeups for RPC dispatch. Work hand-off among threads may cause OS-induced scheduling tails.

SDP is best at intermediate loads. SIP ceases being the best model when the offered load grows too large for

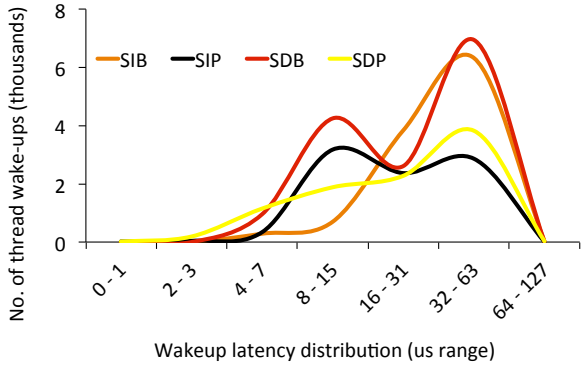


Figure 9: `HDSearch` sync. thread wakeups at 64 QPS: Block incurs more wakeups.

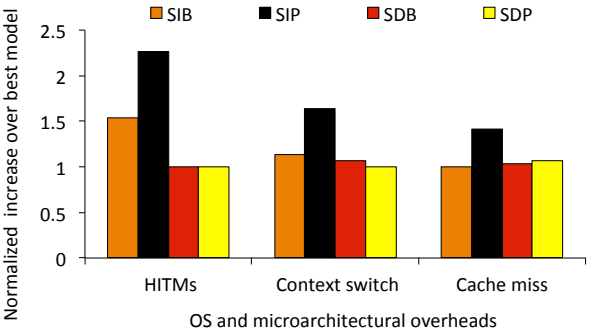
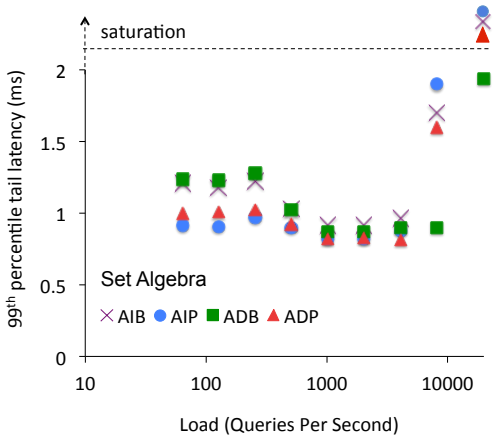


Figure 10: Relative frequency of sync. contention, context switches & cache misses at 10K QPS: SIP does worst.

one in-line thread to sustain. Adding more in-line polling threads causes contention in the OS and RPC reception code paths. Additional in-line blocking threads are less disruptive, but SIB never outperforms SDP. By switching to a dispatched model, a single network thread can still accept the incoming RPCs, avoiding contention and locality losses of running the `gRPC` [18] and network receive stacks across many cores. The workers add sufficient concurrency to sustain RPC and response processing. We further note that SDP tail latencies at intermediate loads are better than at low load, since there is better temporal locality and OS and networking performance tend to improve due to batching effects in the networking stack.

OS and microarchitectural effects. We report OS thread wakeup latency distributions for `HDSearch` synchronous models at 64 QPS in Fig. 9. Although some OS thread wakeups are fast (∼5 μs), blocking models frequently incur 32-64 μs range wakeups. This data also depicts the advantage of in-line over dispatched models with respect to low-load worker wakeup costs.

Fig. 10 shows the relative frequency of true sharing misses (HITM), context switches, and cache misses for threading models at high load (10K QPS). These results show why SIP fails to scale as load increases. SIP needs multiple threads to sustain loads $\geq$ 512 QPS. Multiple pollers contend pathologically on the network receive pro-



QPS	64	128	256	512	1024	2048	4096	8192	20K
AIB	1.3	1.3	1.3	1.2	1.1	1.1	1.2	1.9	∞
AIP	**1**	**1**	**1**	**1**	**1**	**1**	1.1	2.1	∞
ADB	1.4	1.4	1.3	1.1	1.1	1.1	1.1	**1**	**1**
ADP	1.1	1.1	1.1	**1**	**1**	**1**	**1**	1.8	∞

Figure 11: Graph: Latency vs. load for `Set Algebra` async. models. Table: Latencies at each load normalized to the best latency for that load—No threading model is always the best.

cessing, incurring many sharing misses, context switches, and cache misses. SIB in-line threads contend less as they block, rather than poll. SDB and SDP exhibit similar contention. However, SDB outperforms SDP, since SDP incurs a mean $\sim$ 10% higher wasted CPU utilization.

Additional Tests. (1) We measured *µTune* with null (empty) RPC handlers. Complete services incur higher tails than null RPCs as mid-tier and leaf computations add to tails. For null RPCs, SIP outperforms SDB by 1.57× at low loads. (2) We measured `HDSearch` on another hardware platform (Intel Xeon "Skylake" vs. "Haswell"). We notice similar trends as on our primary Haswell platform, with SIP outperforming SDB by 1.42× at low loads. (3) We note that the median latency follows a similar trend, but, with lower absolute values (e.g., `HDSearch`'s SIP outperforms SDB by 1.26× at low load). We omit figures for these tests as they match the reported `HDSearch` trends. Threading performance gaps will be wider for faster services (e.g., 200K QPS Memcached [26]) as slightest OS/network overheads will become magnified [122].

7.1.3 *Asynchronous models*

We show results for `Set Algebra`'s asynchronous models in Fig. 11. As above, we omit figures for additional services as they match `Set Algebra` trends. Broadly, trends follow the synchronous models, but latencies are markedly lower. We note the following differences:

Smaller thread pool sizes. Significantly smaller (≤ 4 threads) thread pool sizes are sufficient at various loads, since asynchronous models capitalize on the available concurrency by quickly moving on to successive requests.

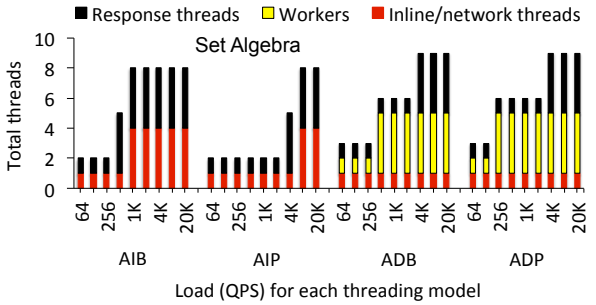


Figure 12: Async. thread pools for best tails: Big pools contend.

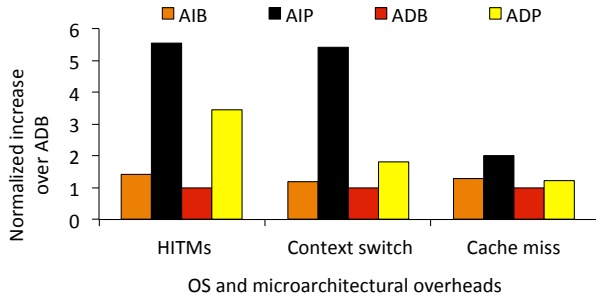


Figure 13: Async. `Set Algebra`'s relative frequency of contention, context switches, & cache misses over best model at peak load: AIP performs worst.

Fig. 12 shows `Set Algebra`'s asynchronous thread pool sizes that achieve the best tails for each load level. We find four threads enough to sustain high loads. Larger thread pools deteriorate latency by contending for network sockets or CPU resources. In contrast, SIB, SDB, and SDP need many threads (as many as 50) to exploit available concurrency.

AIP scales much better than SIP. AIP with just one in-line and response thread can tolerate much higher load (up to 4096 QPS) than SIP, since queuing delays engendered by both the front-end network socket and leaf node response sockets are avoided by the asynchronous design.

ADP scales worse than SDP. ADP with 4 worker and response threads copes worse than SDP at loads $\geq$ 8192 QPS even though it does not have a large thread pool contending for CPU (in contrast to SDP at high loads). This design fails to scale since response threads contend on the completion queue tied to leaf node response sockets.

OS and microarchitectural effects. Unlike SDP, ADP incurs more context switches, caches misses, and HITMs, due to response thread contention (Fig. 13).

7.2 Load adaptation

We next compare *µTune*'s load adaptation against state-of-the-art baselines [43, 76, 97] for various load patterns.

7.2.1 *Comparison to the state-of-the-art*

We compare *µTune*'s run-time performance to state-of-the-art adaptation techniques [43, 76, 97]. We find that

μ*Tune* offers better tail latency than these approaches.

Few-to-Many (FM) parallelism. FM [76] uses offline profiling to vary parallelism during a query's execution. The FM scheduler decides *when* to add parallelism for long-running queries and by *how much*, based on the dynamic load that is observed every 5 ms. In consultation with FM's authors, we opt to treat a microservice as an FM query, to create a fair performance analogy between μ*Tune* and FM. In our FM setup, we mimic FM's offline profiling by building an offline interval table that notes the software parallelism to add for varied loads in terms of thread pool sizes. We use the peak load-sustaining synchronous and asynchronous models (SDB and ADB). During run-time, we track the mid-tier's loads every 5 ms and suitably vary SDB/ADB's thread pool sizes. FM varies only pool sizes (vs. μ*Tune* also varying threading models), and we find that FM underperforms μ*Tune*.

Integrating Polling and Interrupts (IPI). Langendoen et al. [97] propose a user-level communication system that adapts between poll- and interrupt-driven request reception. The system initially uses interrupts. It starts to poll when all threads are blocked. It reverts to interrupts when a blocked thread becomes active. We study this system for synchronous modes only; as its authors note [97], it does not readily apply for asynchronous modes.

To implement this technique, we keep (1) a global count of all threads, and (2) a shared atomic count of *blocked* threads for the mid-tier. Before a thread becomes *blocked* (e.g., invokes a synchronous call), it increments the shared count and decrements it when it becomes active (i.e., synchronous call returns). After revising the shared count, a thread checks if the system's *active* thread count exceeds the machine's logical core count. If higher, the system blocks, otherwise, it shifts to polling. We find that μ*Tune* outperforms this technique, as it considers additional model dimensions (such as inline/dispatch), as well as dynamically scales thread pools based on load.

Time window-Based Detection (TBD). Abdelzaher *et al.* [43] periodically observe request arrival times in fixed observation windows to track request rate. In our setup, we replace μ*Tune*'s event-based detector with this time-based detector. We pick 5 ms time-windows (like FM) to track low loads and react quickly to load spikes.

We evaluate the tail latency exhibited by μ*Tune* across all services, and compare it to these state-of-the-art approaches [43, 76, 97] for both steady-state and transient loads. We examine μ*Tune*'s ability to pick a suitable threading model and size thread pools for time-varying load. We offer loads that differ from those used in training. We aim to study if μ*Tune* selects the best threading model, as compared to an offline exhaustive search.

7.2.2 Steady-state adaptation

Fig. 14 shows μ*Tune*'s ability in converging to the best threading model and thread pool size for steady-state loads. Our test steps up and down through the displayed load levels. We report the tail latency at each load averaged over five trials. The SIP1, SDP1-20, and SDB1-50 bars are optimal threading configurations for some loads. The nomenclature is the threading model followed by the pool sizes, in the form model-network-worker-response. The FM [76], Integrated Poll/Interrupt (IPI) [97], and Time-Based Detection (TBD) [43] bars are the tail latency of state-of-the-art systems. The red bars are μ*Tune*'s tail latency; bars are labelled with the configuration μ*Tune* chose.

In synchronous mode (Fig. 14 (top)), μ*Tune* first selects an SIP model with a single thread, until load grows to about 1K QPS, at which point it switches to SDP, and begins ramping up the worker thread pool size. At 8K QPS, it switches to SDB and continues growing the worker thread pool, until it reaches 50 threads, which is sufficient to meet the peak load the leaf microservice can sustain.

μ*Tune* boosts tail latency by up to 1.7× for `HDSearch`, 1.6× for `Router`, 1.4× for `Set Algebra`, and 1.5× for `Recommend` (at 20 QPS) over SDB—the static model that sustains peak loads. μ*Tune* boosts tail latency by a mean 1.3x over SDB across all loads and services. μ*Tune* also outperforms all state-of-the-art [43, 76, 97] techniques (except TBD) for at least one load level and never underperforms. μ*Tune* outperforms FM by up to 1.3× for `HDSearch` and `Recommend`, and 1.4× for `Router` and `Set Algebra` under low loads, as FM only varies SDB's thread pool sizes and hence incurs high network poller and worker wakeups. μ*Tune* outperforms the IPI approach by up to 1.6× for `HDSearch`, 1.5× for `Router` and `Recommend`, and 1.4× for `Set Algebra` under low loads. At low load, IPI polls with many threads (to sustain peak load), succumbing to expensive contention. TBD does as well as μ*Tune* as the requests mishandled during the 5 ms monitor window fall in tails greater than the 99^{th}% percentile that we monitor for 30s for each load level.

In asynchronous mode (Fig. 14 (bottom)), μ*Tune* again initially selects an in-line poll model with small-sized pools, transitioning to ADP and then ADB as load grows. Four worker and response threads suffice for all loads. We show that μ*Tune* outperforms static threading choices and state-of-the-art techniques by up to 1.9× for at least one load level.

Across all loads, μ*Tune* selects threading models and thread pool sizes that perform within 5% of the best model as determined by offline search. μ*Tune* incurs less than 5% mean instruction overhead over the load-specific "best" threading model, as depicted in Fig. 15. Hence, we find our piece-wise linear model sufficient to make good threading decisions. Note that μ*Tune* always prefers a

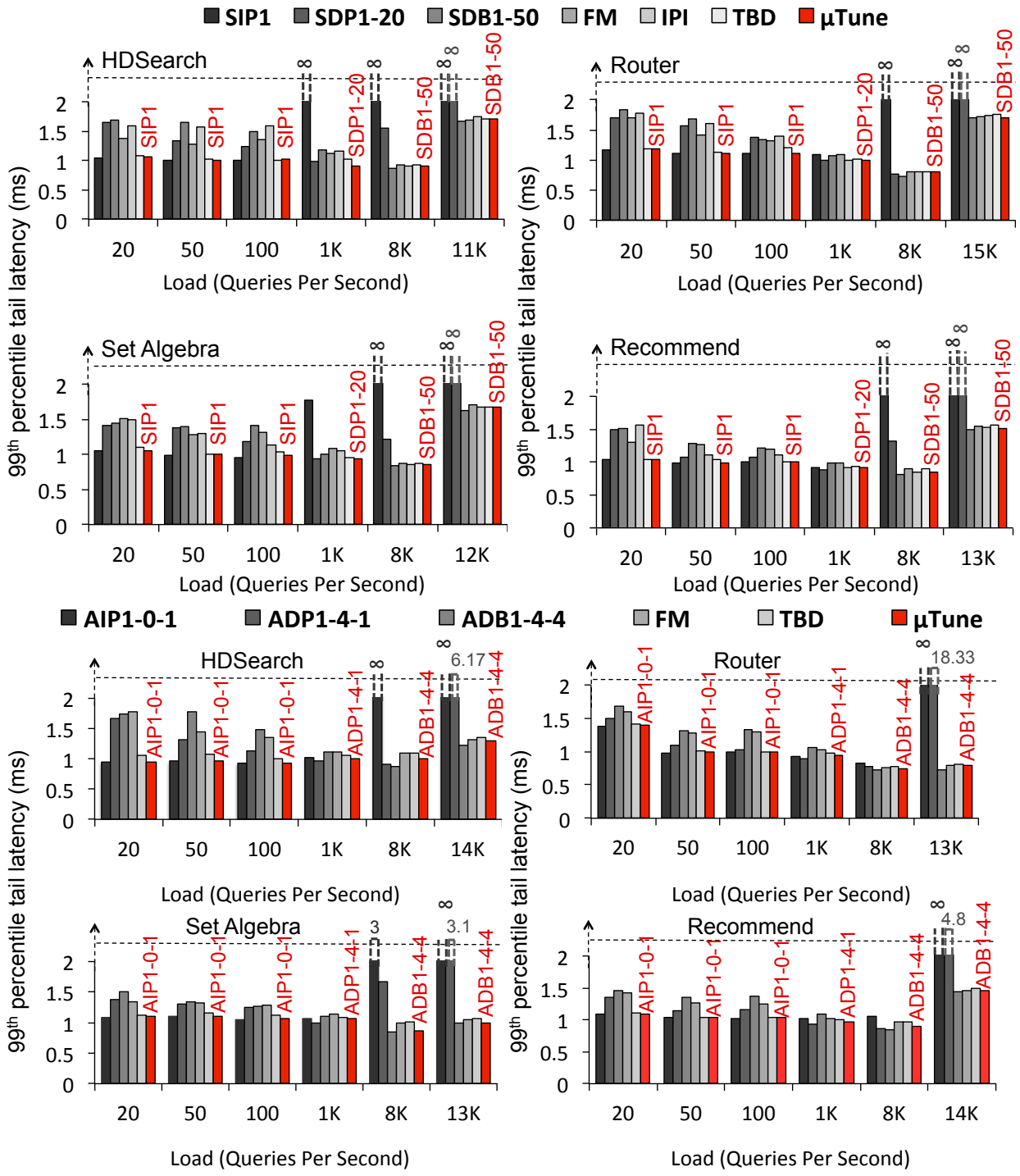


Figure 14: Synchrounous (top) & asynchronous (bottom) steady-state adaptation.

single thread interacting with the front-end socket. This finding underscores the importance of maximizing locality and avoiding contention on the RPC receive path.

7.2.3 *Load transients*

Table 2 indicates *µTune*'s response to load transients, where the columns are a series of varied-duration load levels. The rows are the 99^{th}% tail latency for the models between which *µTune* adapts in this scenario (SIP/AIP

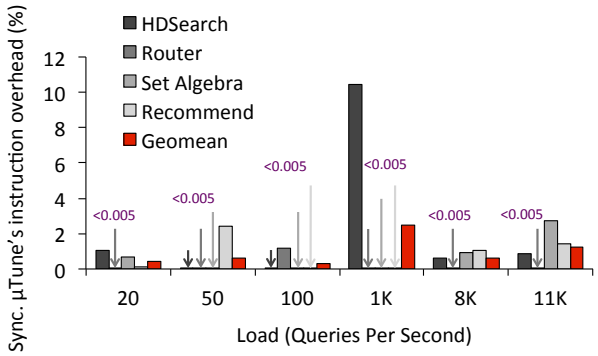


Figure 15: Sync. *µTune*'s instruction overhead for steady-state loads: less than 5% mean overhead incurred.

		Synchronous				Asynchronous		
		100 QPS (0 - 30s)	8K QPS (30s - 31s)	100 QPS (31 - 61s)		100 QPS (0 - 30s)	13K QPS (30s - 31s)	100 QPS (31 - 61s)
HDSearch	SIP	0.99	>1s	>1s	AIP	0.95	>1s	>1s
	SDB	1.49	1.07	1.40	ADB	1.48	1.10	1.40
	FM	1.35	13.00	1.32	FM	1.28	4.73	1.33
	IPI	1.59	1.10	1.50	IPI	NA	NA	NA
	TBD	1.03	8.69	1.02	TBD	1.06	2.63	1.08
	µTune	1.01	1.09	0.99	*µTune*	0.98	1.13	0.96
Router	SIP	1.10	>1s	>1s	AIP	1.01	>1s	>1s
	SDB	1.31	0.83	1.36	ADB	1.35	1.13	1.31
	FM	1.33	9.40	1.40	FM	1.30	12.95	1.30
	IPI	1.4	1.10	1.38	IPI	NA	NA	NA
	TBD	1.13	4.51	1.11	TBD	1.03	6.24	1.01
	µTune	1.12	0.88	1.13	*µTune*	0.99	1.02	0.98
Set Algebra	SIP	0.95	>1s	>1s	AIP	1.04	>1s	>1s
	SDB	1.30	0.92	1.32	ADB	1.26	0.99	1.23
	FM	1.30	12.00	1.25	FM	1.28	4.14	1.27
	IPI	1.20	0.94	1.12	IPI	NA	NA	NA
	TBD	1.00	8.45	1.03	TBD	1.09	6.62	1.1
	µTune	0.97	0.92	1.03	*µTune*	1.06	1.1	1.06
Recommend	SIP	1.00	>1s	>1s	AIP	1.03	>1s	>1s
	SDB	1.26	0.96	1.22	ADB	1.37	1.30	1.32
	FM	1.23	>1s	>1s	FM	1.28	8.61	1.20
	IPI	1.13	1.02	1.13	IPI	NA	NA	NA
	TBD	1.02	4.96	1.03	TBD	1.06	6.00	1.07
	µTune	1.00	1.00	1.00	*µTune*	1.06	1.39	1.04

Table 2: 99^{th}% tail latency (ms) for load transients.

and SDB/ADB), state-of-the-art [43, 76, 97] techniques, and *µTune*. The key step in this scenario is the 8K/13K QPS load level, which lasts only 1s. We pick spikes of 8K QPS and 13K QPS for synchronous and asynchronous as these loads are SIP and AIP saturation levels, respectively.

We find that the in-line poll models accumulate a large backlog during the transient as they saturate, and thus perform poorly even during successive low loads. FM and TBD incur high transient tail latencies as they allow requests during the 5 ms load detection window to be handled by sub-optimal threading choices. FM saturates at 8K QPS for `Recommend` since the small SDB thread pool size opted by FM at 100 QPS causes unbounded queuing during the load monitoring window. IPI works only for synchronous and performs poorly at low loads as its fixed-size thread pool leads to polling contention. We show that *µTune* detects the transient and transitions from SIP/AIP to SDB/ADB fast enough to avoid accumulating a backlog that affects tail latency. Once the flash crowd subsides, *µTune* transitions back to SIP/AIP, avoiding the latency penalty SDB/ADB suffer at low load.

8 Discussion

We briefly discuss open questions and *µTune* limitations.

Offline training. *µTune* uses offline training to build a piece-wise linear model. This phase might be removed by analyzing dynamically OS and hardware signals, such as context switches, thread wakeups, queue depths, cache misses, and lock contention, to switch threading models. Designing heuristics to switch optimally based on such run-time metrics remains an open question; our performance characterization can help guide their development.

Thread pool sizing. *µTune* tunes thread pool sizes using a piece-wise linear model. *µTune* differs from prior thread pool adaptation systems [76, 86, 93] in that it also tunes threading models. Some of these systems use more sophisticated tuning heuristics, but we did not observe opportunity for further improvement in our microservices.

CPU cost of polling. *µTune* polls at low loads to avoid thread wakeups. Polling can be costly as it wastes CPU time in fruitless poll loops. However, as most operators over-provision CPU to sustain high loads [130], when load is low, spare CPU time is typically available [79].

***µ*Tune's asynchronous framework.** Asynchronous RPC state must be maintained in thread-safe structures, which is challenging. More library/language support might simplify building asynchronous microservices with *µTune*. We leave such support to future work.

Comparison with optimized systems that use kernel-bypass, multi-queue NICs, etc. It may be interesting to study the implications of optimized systems [53, 87, 91, 103, 121, 122] that incorporate kernel-bypass, multi-queue NICs, etc., on threading models and *µTune*. Multi-queue NICs may improve polling scalability; multiple network pollers currently contend for the underlying `gRPC` [18] queues under *µTune*. OS-bypass may further increase the application threading model's importance; for example, it may magnify the trade-off between in-line and dispatch RPC execution, as OS-bypass eliminates latency and thread hops in the OS TCP/IP stack, shifting the break-even point to favor in-line execution for longer RPCs. However, in this paper, we have limited our scope to study designs that can layer upon (unmodified) `gRPC` [18]; we defer studies that require extensive `gRPC` [18] changes (or an alternative reliable transport) to future work.

9 Related Work

We discuss several categories of related work.

Web server architectures. Web servers can have (a) thread-per-connection [119], (b) event-driven [120], (c) thread-per-request [84], or (d) thread-pool architectures [104]. Pai *et al.* [119] build thread-per-connection servers as multi-threaded processes. Knot [145] is a thread-per-connection non-blocking server. In contrast, μ*Tune* is a thread-per-request thread-pool architecture that scales better for microservices [104]. The Single Process Event-Driven (SPED) [119] architecture operates on asynchronous ready sockets. In contrast, μ*Tune* supports both synchronous and asynchronous I/O. The SYmmetric Multi-Process Event-Driven (SYMPED) [120] architecture runs many processes as SPED servers via context switches. The Staged Event-Driven Architecture (SEDA) [148] joins event-driven stages via queues. A stage's thread pool is driven by a resource controller. Apart from considering synchronous and asynchronous I/O like prior works [69,83,84,120,146,148], μ*Tune* also studies a full microservice threading model taxonomy. gRPC-based systems such as Envoy [13] or Finagle [16] act as load balancers or use a single threading model.

Software techniques for tail latency: Prior works [84, 148] note that monolithic service software designs can significantly impact performance. But, microsecond–scale OS and network overheads that dominate in μ*Tune*'s regime do not manifest in these slower services. Some works improve web server software via software pattern re-use [127, 128], caching file systems [84], or varying parallelism [76], all of which are orthogonal to the questions we investigate. Kapoor *et al.* [91] also note that OS and network overheads impact short-running cloud services, but, their kernel bypass solution may not apply for all contexts (e.g., a shared cloud infrastructure).

Parallelization to reduce latency: Several prior works [54,61,89,94,99,104,124,135,147] reduce tails via parallelization. Others [47,70,80,112] improve medians by adaptively sharing resources. Prior works use prediction [86,93], hardware parallelism [76], or data parallelism [73] to reduce monolithic services' tail latency. Lee *et al.* [99] use offline analysis (like μ*Tune*) to tune thread pools. We study microservice threading, and vary threading models altogether. But, we build on prior works' thread pool sizing insights.

Hardware mechanisms for tail latency: Several other prior works improve leaf service tail latency via better co-location [102], voltage boosting [82,92], or applying heterogeneity in multi-cores [77]. But, they do not study microservice tail latency effects engendered by software threading, OS, or network.

10 Conclusion

Prior works study monolithic OLDI services, where microsecond-scale overheads are negligible. The rapid advent of faster I/O and low-latency microservices calls for analyzing threading effects for the microsecond regime. In this paper, we presented a structured threading model taxonomy for microservices. We identified different models' performance effects under diverse load. We proposed μ*Tune*—a novel framework that abstracts microservice threading from application code and automatically adapts to offered load. We show that selecting load-optimal threading models can improve tail latency by up to 1.9x.

11 Acknowledgement

We thank our shepherd, Dr. Hakim Weatherspoon, and the anonymous reviewers for their valuable feedback. We thank Amlan Nayak for creating the HDSearch data set, and Neha Agarwal for reviewing μ*Tune*'s code base.

We acknowledge Ricardo Bianchini, Manos Kapritsos, Mosharaf Chowdhury, Baris Kasikci, David Devecsery, Md Haque, Aasheesh Kolli, Karin Strauss, Inigo Goiri, Geoffrey Blake, Joel Emer, Gabriel Loh, A. V. Madhavapeddy, Zahra Tarkhani, and Eric Chung for their insightful suggestions that helped improve this work.

We are especially grateful to Vaibhav Gogte and Animesh Jain for their input in shaping the μ*Tune* concept. We thank P.R. Sriraman, Rajee Sriraman, Amrit Gopal, Akshay Sriraman, Vidushi Goyal, and Amritha Varshini for proof-reading our draft.

This work was supported by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. This work was also supported by NSF Grant IIS1539011 and gifts from Intel.

12 References

[1] Adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[2] Aerospike. https://www.aerospike.com/docs/client/java/usage/async/index.html.

[3] Apache http server project. https://httpd.apache.org/.

[4] Average number of search terms for online search queries in the United States as of August 2017. https://www.statista.com/statistics/269740/number-of-search-terms-in-internet-research-in-the-us/.

[5] Azure Synchronous I/O antipattern. https://docs.microsoft.com/en-us/azure/architecture/resiliency/high-availability-azure-applications.

[6] The biggest thing amazon got right: The platform. https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/.

[7] BLPOP key timeout. https://redis.io/commands/blpop.

[8] Bob Jenkins. SpookyHash: a 128-bit noncryptographic hash. http://burtleburtle.net/bob/hash/spooky.html.

[9] Building products at soundcloud: Dealing with the monolith. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith.

[10] Building Scalable and Resilient Web Applications on Google Cloud Platform. https://cloud.google.com/solutions/scalable-and-resilient-apps.

[11] Celery: Distributed Task Queue. http://www.celeryproject.org/.

[12] Chasing the bottleneck: True story about fighting thread contention in your code. https://blogs.mulesoft.com/biz/news/chasing-the-bottleneck-true-story-about-fighting-thread-contention-in-your-code/.

[13] Envoy. https://www.envoyproxy.io/.

[14] Facebook Thrift. https://github.com/facebook/fbthrift.

[15] Fighting spam with haskell. https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/.

[16] Finagle. https://twitter.github.io/finagle/guide/index.html.

[17] From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. https://www.infoq.com/presentations/linkedin-microservices-urn.

[18] gRPC. https://github.com/heathermiller/dist-prog-book/blob/master/chapter/1/gRPC.md.

[19] Handling 1 Million Requests per Minute with Go. http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang/.

[20] Improve Application Performance With SwingWorker in Java SE 6. http://www.oracle.com/technetwork/articles/javase/swingworker-137249.html.

[21] Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/blog/2009/7/25/latency-iseverywhere-and-it-costs-you-sales-how-to-crush-it.html.

[22] Let's look at Dispatch Timeout Handling in WebSphere Application Server for z/OS. https://www.ibm.com/developerworks/community/blogs/aimsupport/entry/dispatch_timeout_handling_in_websphere_application_server_for_zos?lang=en.

[23] Linux bcc/BPF Run Queue (Scheduler) Latency. http://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html.

[24] LPOP key. https://redis.io/commands/lpop.

[25] Mcrouter. https://github.com/facebook/mcrouter.

[26] Memcached performance. https://github.com/memcached/memcached/wiki/Performance.

[27] Microsoft Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[28] mongoDB. https://www.mongodb.com/.

[29] Myrocks: A space- and write-optimized MySQL database. https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/.

[30] OpenImages: A public dataset for large-scale multi-label and multi-class image classification. https://github.com/openimages.

[31] Pokemon go now the biggest mobile game in US history. http://www.cnbc.com/2016/07/13/pokemon-go-now-the-biggest-mobile-game-in-us-history.html.

[32] Programmer's Guide, Release 2.0.0. https://www.intel.com/content/dam/www/public/us/en/documents/guides/dpdk-programmers-guide.pdf.

[33] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[34] Redis Replication. https://redis.io/topics/replication.

[35] Resque. https://github.com/defunkt/resque.

[36] RQ. http://python-rq.org/.

[37] Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture. https://www.infoq.com/presentations/scale-gilt.

[38] Setting Up Internal Load Balancing. https://cloud.google.com/compute/docs/load-balancing/internal/.

[39] What is microservices architecture? https://smartbear.com/learn/api-design/what-are-microservices/.

[40] Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Plagiarism&oldid=5139350.

[41] Workers inside unit tests. http://python-rq.org/docs/testing/.

[42] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Computing Research Repository*, 2016.

[43] T. F. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service*, 1999.

[44] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *IEEE Symposium on Foundations of Computer Science*, 2006.

[45] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and Optimal LSH for Angular Distance. In *Advances in Neural Information Processing Systems*. 2015.

[46] I. Arapakis, X. Bai, and B. B. Cambazoglu. Impact of Response Latency on User Behavior in Web Search. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2014.

[47] N. Bansal, K. Dhamdhere, J. Könemann, and A. Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 2004.

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

[49] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 2017.

[50] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. In *IEEE Micro*, 2003.

[51] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 2007.

[52] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *International conference on World Wide Web*, 2005.

[53] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX Conference on Operating Systems Design and Implementation*, 2014.

[54] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Computing*, 2007.

[55] A. Bouch, N. Bhatti, and A. Kuchinsky. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *ACM Conference on Human Factors and Computing Systems*, 2000.

[56] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, 2013.

[57] J. Cao, M. Andersson, C. Nyberg, and M. Kihl. Web server performance modeling using an m/g/1/k* ps queue. In *International Conference on Telecommunications*. IEEE.

[58] J. L. Carlson. *Redis in Action*. 2013.

[59] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing*, 2010.

[60] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 2013.

[61] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Annual International conference on Supercomputing*, 2006.

[62] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Annual Symposium on Computational Geometry*, 2004.

[63] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 2013.

[64] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[65] C. C. Del Mundo, V. T. Lee, L. Ceze, and M. Oskin. NCAM: Near-Data Processing for Nearest Neighbor Search. In *International Symposium on Memory Systems*, 2015.

[66] N. Dmitry and S.-S. Manfred. On micro-services architecture. *International Journal of Open Information Technologies*, 2014.

[67] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling LSH for performance tuning. In *ACM conference on Information and knowledge management*, 2008.

[68] D. Ersoz, M. S. Yousif, and C. R. Das. Characterizing network traffic in a cluster-based, multi-tier data center. In *International Conference on Distributed Computing Systems*, 2007.

[69] Q. Fan and Q. Wang. Performance comparison of web servers with different architectures: a case study using high concurrency workload. In *IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2015.

[70] D. G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. *IBM Research Division*, 1994.

[71] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[72] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004.

[73] E. Frachtenberg. Reducing query latencies in web search using fine-grained parallelism. *World Wide Web*, 2009.

[74] B. Furht and A. Escalante. *Handbook of cloud computing*. Springer, 2010.

[75] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *International Conference on Very Large Data Bases*, 1999.

[76] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[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 *IEEE/ACM International Symposium on Microarchitecture*, 2017.

[78] F. M. Harper and J. A. Konstan. The Movielens Datasets: History and Context. *ACM Tranactions on Interactive Intelligent Systems*, 2015.

[79] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *IEEE International Symposium on High Performance Computer Architecture*, 2018.

[80] Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient online nonclairvoyant adaptive scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[81] E. N. Herness, R. J. High, and J. R. McGee. Websphere Application Server: A foundation for on demand computing. *IBM Systems Journal*, 2004.

[82] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. Dreslinski. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *International Symposium on High Performance Computer Architecture*, 2015.

[83] J. Hu, I. Pyarali, and D. C. Schmidt. Applying the proactor pattern to high-performance web servers. In *International Conference on Parallel and Distributed Computing and Systems*, 1998.

[84] J. C. Hu and D. C. Schmidt. JAWS: A Framework for High-performance Web Servers. In *In Domain-Specific Application Frameworks: Frameworks Experience by Industry*, 1999.

[85] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *ACM Symposium on Theory of Computing*, 1998.

[86] 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 *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2014.

[87] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX Conference on Networked Systems Design and Implementation*, 2014.

[88] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling Contention Management from Scheduling. In *Architectural Support for Programming Languages and Operating Systems*, 2010.

[89] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2005.

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

[91] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *ACM Symposium on Cloud Computing*, 2012.

[92] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *International Symposium on Microarchitecture*, 2015.

[93] S. Kim, Y. He, S.-w. Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *ACM International Conference on Web Search and Data Mining*, 2015.

[94] W. Ko, M. Yankelevsky, D. S. Nikolopoulos, and C. D. Polychronopoulos. Effective cross-platform, multilevel parallelism via dynamic adaptive execution. In *Parallel and*

Distributed Processing Symposium, 2001.

[95] R. Kohavi, R. M. Henne, and D. Sommerfield. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers Not to the Hippo. In *International Conference on Knowledge Discovery and Data Mining*, 2007.

[96] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM Journal on Computing*, 2000.

[97] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating polling, interrupts, and thread management. In *Symposium on the Frontiers of Massively Parallel Computing*, 1996.

[98] P.-A. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent mid-tier database caching in SQL server. In *International Conference on Data Engineering*, 2004.

[99] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *International Symposium on Computer Architecture*, 2010.

[100] A. Lesyuk. *Mastering Redmine*. 2013.

[101] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Workshop on Experimental computer science*, 2007.

[102] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *ACM Symposium on Cloud Computing*, 2014.

[103] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *USENIX Conference on Networked Systems Design and Implementation*, 2014.

[104] Y. Ling, T. Mullen, and X. Lin. Analysis of Optimal Thread Pool Size. *SIGOPS Operating Systems Review*, 2000.

[105] D. Liu and R. Deters. The Reverse C10K Problem for Server-Side Mashups. In *International Conference on Service-Oriented Computing Workshops*, 2008.

[106] P. M. LiVecchi. Performance enhancements for threaded servers, 2004. US Patent 6,823,515.

[107] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *International Symposium on Computer Architecture*, 2014.

[108] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *International Symposium on Computer Architecture*, 2015.

[109] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. LASER: Light, Accurate Sharing dEtection and Repair. In *International Symposium on High Performance Computer Architecture*, 2016.

[110] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search. In *International Conference on Very Large Data Bases*, 2007.

[111] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. 2010.

[112] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 1993.

[113] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-intensive Services. In *International Symposium on Computer Architecture*, 2011.

[114] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed event-based systems*. 2006.

[115] M. Muja and D. G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[116] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. 2016.

[117] R. M. Needham. Denial of Service. In *ACM Conference on Computer and Communications Security*, 1993.

[118] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, and P. Saab. Scaling Memcache at Facebook. In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[119] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, 1999.

[120] D. Pariag, T. Brecht, A. S. Harji, P. A. Buhr, A. Shukla, and D. R. Cheriton. Comparing the performance of web server architectures. In *European Conference on Computer Systems*, 2007.

[121] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 2016.

[122] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Symposium on Operating Systems Principles*, 2017.

[123] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *ACM Symposium on Cloud Computing*, 2015.

[124] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *IEEE International Symposium on Workload Characterization*, 2011.

[125] D. R. Raymond and S. F. Midkiff. Denial-of-service in wireless sensor networks: Attacks and defenses. *IEEE Pervasive Computing*, 2008.

[126] R. Rojas-Cessa, Y. Kaymak, and Z. Dong. Schemes for fast transmission of flows in data center networks. *IEEE Communications Surveys & Tutorials*, 2015.

[127] D. Schmidt and P. Stephenson. Experience using design patterns to evolve communication software across diverse OS platforms. In *European Conference on Object-Oriented Programming*, 1995.

[128] D. C. Schmidt and C. Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine*, 1999.

[129] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *IEEE International Conference on Computer Vision*, 2003.

[130] R. K. Sharma, C. E. Bash, C. D. Patel, R. J. Friedrich, and J. S. Chase. Balance of power: Dynamic thermal management for internet data centers. *IEEE Internet Computing*, 2005.

[131] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal Processing Magazine*, 2008.

[132] S. M. Specht and R. B. Lee. Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures. In *ISCA International Conference on Parallel and Distributed Computing (and Communications) Systems*, 2004.

[133] A. Sriraman, S. Liu, S. Gunbay, S. Su, and T. F. Wenisch. Deconstructing the Tail at Scale Effect Across Network Protocols. *The Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2016.

[134] A. Sriraman and T. F. Wenisch. μSuite: A Benchmark Suite for Microservices. In *IEEE International Symposium on Workload Characterization*, 2018.

[135] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[136] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[137] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *ACM SIGMOD International Conference on Management of data*, 2009.

[138] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems*, 2010.

[139] K. Terasawa and Y. Tanaka. Spherical LSH for approximate nearest neighbor search on unit hypersphere. In *Workshop on Algorithms and Data Structures*, 2007.

[140] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 2010.

[141] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Workshop on Experimental computer science*, 2007.

[142] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2012.

[143] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar. Timetrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *International Symposium on Microarchitecture*, 2015.

[144] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference*, 2015.

[145] J. R. Von Behren, J. Condit, and E. A. Brewer. Why Events Are a Bad Idea (for High-Concurrency Servers). In *Hot Topics in Operating Systems*, 2003.

[146] 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 *International Conference on Distributed Computing Systems*, 2017.

[147] Z. Wang and M. F. O'Boyle. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[148] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *ACM Symposium on Operating Systems Principles*, 2001.

[149] W. J. Wilbur and K. Sirotkin. The automatic identification of stop words. *Journal of information science*, 1992.

[150] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *ACM SIGCOMM Conference*, 2011.

[151] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the Source of Tail Latency Through Precise Load Testing and Statistical Inference. In *International Symposium on Computer Architecture*, 2016.