μSuite: A Benchmark Suite for 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 single-digit millisecond RPC latency goals (implying sub-ms medians)—much tighter than their monolithic ancestors that must meet ≥ 100 ms latency targets. Sub-ms-scale OS/network overheads that were once insignificant for such monoliths can now come to dominate in the sub-ms-scale microservice regime. It is therefore vital to characterize the influence of OS- and network-based effects on microservices. Unfortunately, widely-used academic data center benchmark suites are unsuitable to aid this characterization as they (1) use monolithic rather than microservice architectures, and (2) largely have request service times ≥ 100 ms. In this paper, we investigate how OS and network overheads impact microservice median and tail latency by developing a complete suite of microservices called *μSuite* that we use to facilitate our study. *μSuite* comprises four OLDI services composed of microservices: image similarity search, protocol routing for key-value stores, set algebra on posting lists for document search, and recommender systems. Our characterization reveals that the relationship between optimal OS/network parameters and service load is complex. Our primary finding is that non-optimal OS scheduler decisions can degrade microservice tail latency by up to $\sim 87\%$.**

***Index Terms*—OLDI, microservice, mid-tier, benchmark suite, OS and network overheads.**

I. Introduction

On-Line Data Intensive (OLDI) applications such as web search, advertising, and online retail form a major fraction of data center applications [105]. Meeting soft real-time deadlines in the form of Service Level Objectives (SLOs) determines end-user experience [19], [47], [57], [90] and is of paramount importance. Whereas OLDI applications once had largely monolithic software architectures [51], modern OLDI applications comprise numerous, distributed microservices [69], [87], [108], [117] such as HTTP connection termination, key-value serving [73], query rewriting [49], click tracking, access-control management, protocol routing [22], etc. Several companies such as Amazon [3], Netflix [26], Gilt [12], LinkedIn [20], and SoundCloud [38] have adopted microservice architectures to improve OLDI development and scalability [126]. These microservices are composed via standardized Remote Procedure Call (RPC) interfaces, such as Google's Stubby and `gRPC` [15] or Facebook/Apache's Thrift [42].

While monolithic applications face ≥ 100 ms tail (99^{th}+%) latency SLOs (e.g.,$\sim$300 ms for web search [116], [124]) microservices must often achieve single-digit millisecond tail latencies implying sub-ms medians (e.g., $\sim$100 μs for protocol routing [130]) as many microservices must be invoked serially to serve a user's query. For example, a Facebook news feed service [81] query may flow through a serial pipeline of many microservices, such as (1) `Sigma` [18]: a spam filter, (2) `McRouter` [109]: a protocol router, (3) `Tao` [58]: a distributed social graph data store, (4) `MyRocks` [25]: a user database, thereby placing tight single-digit millisecond latency constraints on individual microservices. We expect continued growth in OLDI data sets and applications with composition of ever more microservices with increasingly complex interactions. Hence, the pressure for better microservice latency continually mounts.

Prior academic studies have focused on monolithic services [71], which typically have ≥ 100 ms tail SLOs [103]. Hence, sub-ms-scale OS/network overheads (e.g., a context switch cost of 5-20 μs [123]) are often insignificant for monolithic services. However, the sub-ms-scale regime differs fundamentally: OS/network overheads that are often minor at ≥ 100 ms-scale, such as spurious context switches, network and RPC protocol delays, inefficient thread wakeups, or lock contention, can come to dominate microservice latency distributions. For example, even a single 20 μs spurious context switch implies a 20% latency penalty for a request to a 100 μs-response latency protocol routing microservice [130]. Hence, prior conclusions must be revisited for the microservice regime [50].

Modern OLDI applications comprise a complex web of microservices that interact via RPCs [69] (Fig. 1). Many prior works have studied leaf servers [66], [97], [98], [111], [124], [125], as they are typically most numerous, making them cost-critical. But, we find that *mid-tier* servers, which must manage both incoming and outgoing RPCs to many clients and leaves, perhaps face greater tail latency optimization challenges, but have not been similarly scrutinized. 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.

While it may be possible to study mid-tier microservice overheads in a purely synthetic context, greater insight can be drawn in the context of complete OLDI services. Widely-used academic data center benchmark suites, such as CloudSuite [71] or Google PerfKit [28], are unsuitable for characterizing microservices as they (1) include primarily leaf services, (2) use monolithic rather than microservice architectures, and (3) largely have request service times ≥ 100 ms.

No existing open-source benchmark suite represents the typi-

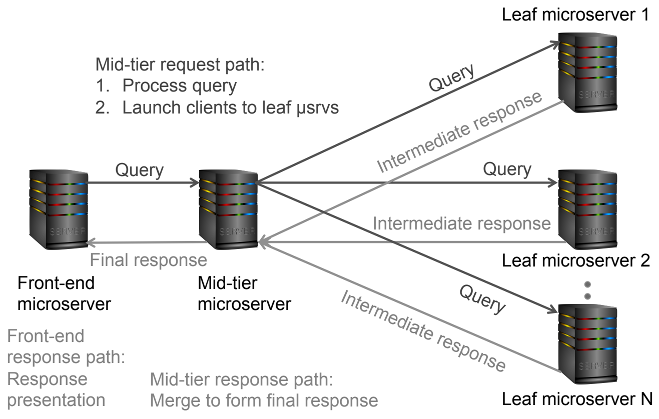

Fig. 1: A typical OLDI application fan-out.

TABLE I: Summary of a comparison of μSuite with prior work.

Prior work	Open-source	μservice arch.	Mid-tier study
SPEC [82]	✓	✗	✗
PARSEC [55]	✓	✗	✗
CloudSuite [71]	✓	✗	✗
TailBench [88]	✓	✗	✗
PerfKit [28]	✓	✗	✗
Ayers et al. [48]	✗	✓	✓
μSuite	✓	✓	✓

cal three-tier microservice structure employed by modern OLDI applications. In this paper, we introduce a benchmark suite—*μSuite*—of OLDI applications composed of three microservice tiers that exhibit traits crucial for our study (sub-ms service time, high peak request rate, scalable across cores, mid-tier with fan-out to leaves). We use *μSuite* to study the OS/network performance overheads incurred by mid-tier microservices.

μSuite includes four OLDI services that incorporate open-source software: a content-based high dimensional search for image similarity—`HDSearch`, a replication-based protocol router for scaling fault-tolerant key-value stores—`Router`, a service for performing set algebra on posting lists for document retrieval—`Set Algebra`, and a user-based item recommender system for predicting user ratings—`Recommend`. Each service's constituent microservices' goal is to perform their individual functionality in at most a few single-digit ms for large data sets.

Our main finding is that the relationship between optimal OS/network parameters and service load is complex. We find that non-optimal OS scheduler decisions can degrade microservice tail latency by up to $\sim 87\%$.

In summary, we contribute:

- *μSuite*: A benchmark suite of OLDI services composed of microservices built using a state-of-the-art open-source RPC platform, which we release as open-source software [1].
- A comprehensive performance characterization of *μSuite*'s key component: the mid-tier microservice.

II. Prior Work

We propose *μSuite* as latency-critical services studied by prior works are unsuitable to characterize microservices (Table I).

Closed-source. Many works [48], [97], [98], [101], [105], [129], [130] use workloads internal to companies such as Google or Facebook and hence do not promote further academic study.

Too few latency-critical workloads. Some academic studies analyze only one latency-critical benchmark [84], [125], thereby limiting the generality of their conclusions.

Not representative. Some works [62], [131] treat sequential/parallel workloads (e.g., SPEC CPU2006 [82]/PARSEC [55]) as OLDI services. But, these workloads are not representative of OLDI services as they intrinsically vary in terms of continuous activity vs. bursty request-responses, architectural traits, etc.

Monolithic architectures. CloudSuite [71], PerfKit [28], and TailBench [88] are perhaps closest to *μSuite*. CloudSuite focuses on microarchitectural traits that impact throughput for both latency-critical and throughput-oriented services. CloudSuite largely incurs ≥ 100 ms tail latencies and is less susceptible to sub-ms-scale OS/network overheads faced by microservices. Moreover, CloudSuite load testers (YCSB [63] and Faban [11]) model only a *closed-loop system* [130], which is methodologically inappropriate for tail latency measurements [130] due to the coordinated omission problem [121]. BigDataBench [127] also lacks a rigorous latency measurement methodology, even though it uses representative data sets. *μSuite*'s load testers account for these problems and record robust and unbiased latencies. CloudSuite, PerfKit, and TailBench employ monolithic architectures instead of microservice architectures, making them unsuitable to study overheads faced by microservices.

Target only leaves. Several studies target only OLDI leaf servers [66], [84], [97], [98], [105], [111], [124], [125] as they are typically most numerous. Hence, conclusions from these works do not readily extend to mid-tier microservers.

Machine-learning based. Recent benchmark suites such as Sirius [80] and Tonic [79] mainly scrutinize ML-based services and incur higher latencies than microservices that *μSuite* targets.

III. μSuite: Benchmarks Description

Although there are many open-source microservices, such as Memcached [73], Redis [60], McRouter [22], etc., that can serve as individual components of a service with a typical three-tier front-end;mid-tier;leaf architecture, there are no representative open-source three-tier services composed of microservices. Hence, we develop four services in *μSuite*, each composed of three microservices. To include services that dominate today's data centers in *μSuite*, we consider a set of information retrieval (IR)-based internet services based on their popularity [44].

All *μSuite* services/benchmarks are built using a state-of-the-art open-source RPC platform—`gRPC` [15].

A. HDSearch

`HDSearch` performs content-based image similarity search. Like Google's "find similar images" [14], this service searches an image repository for matches with content similar to a user's query image. This technique entails nearest neighbor (NN) matching in a high-dimensional abstract feature space to identify images that have similar content to the query image.

[1]https://github.com/wenischlab/MicroSuite

Related work. High dimensional search is an intrinsic part of many user-facing OLDI services, and hence its accuracy and performance have been extensively studied. Many prior works [53], [54], [56], [61], [96], [104], [113] improve high dimensional search via tree-based indexing. Since data sets are growing rapidly in both size and dimensionality, tree-based indexing techniques that are efficient for modest dimensionality data sets no longer apply. Instead, hashing-based indexing that exploits data locality are now more common [45], [52], [70], [100], [114], [115], [119], [120], [122]. Another indexing algorithm class clusters adjacent data [68], [72], [75], [86], [94], [102], [106], [110]. These primarily theoretical works explore high dimensional search's algorithmic foundations; their contributions are orthogonal to the software structure of a service like HDSearch.

Service description. HDSearch indexes a corpus of 500K images taken from Google's Open Images data set [27]. Each image in the corpus is represented by a feature vector, an n-dimensional numerical representation of image content. Today, feature vectors summarizing each image are typically obtained from a deep learning technique. We use the Inception V3 neural network [118] implemented in TensorFlow [43] to represent each data set image in the form of a 2048-dimensional feature vector. The data set size is ~ 10 GB.

One can find images similar to a query image by searching the corpus for response images whose feature vectors are near the query image's feature vector [92], [93]. Proximity is identified by distance metrics such as Euclidean or Hamming distance. The goal of HDSearch's constituent microservices is to perform this image search functionality in at most a few single-digit milliseconds for a large image repository. We describe HDSearch's constituent microservices below.

Front-end microservice. HDSearch's front-end presentation microservice is not studied in this work; we describe its components only to provide brief context (Fig. 2).

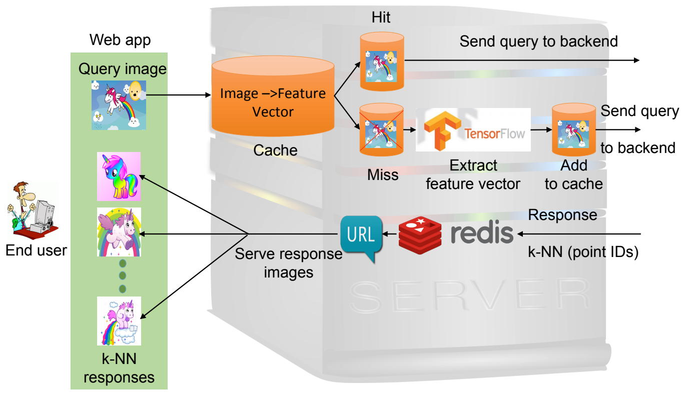

Fig. 2: HDSearch: front-end presentation microservice.

Web application. The web application is merely a useful interface that allows the end-user to upload query images to the front-end microserver and view received query responses.

Feature Extraction. The query image is initially transformed into a discriminative intermediate feature vector representation. We employ Google's Inception V3 neural network [118], implemented in TensorFlow [43], to extract a feature vector for the query image. This feature vector is sent to the mid-tier microservice to retrieve the IDs of k-NN images. This query is the object of study for HDSearch in this paper.

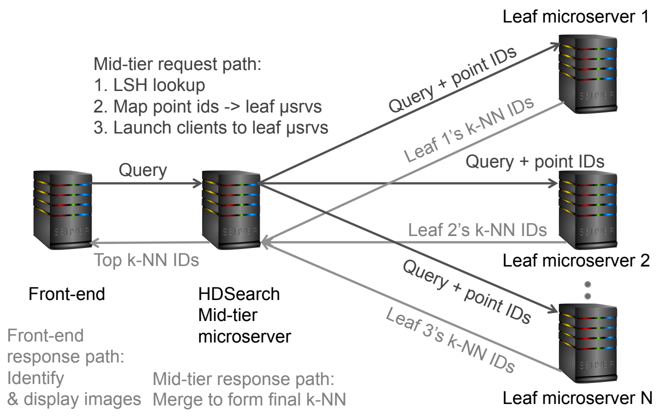

Fig. 3: HDSearch: back end request and response pipelines.

Feature Vector Caching. To minimize feature vector extraction time, a mapping from images to feature vectors is cached in a Redis [60] instance, avoiding repeated feature computations.

Response Image Look-up. Once the query returns, a second Redis [60] instance is consulted to map image IDs to URLs. The presentation microservice then constructs a response web page and returns it to the web application.

Mid-tier microservice. Solving the k-NN problem efficiently is hard due to the *curse of dimensionality* [85], and the problem has been studied extensively [45], [52], [76], [100], [114], [119], [120], [122]. To prune the search space, modern k-NN algorithms use indexing structures, such as Locality-Sensitive Hash (LSH) tables, kd-trees, or k-means clusters to exponentially reduce the search space relative to brute-force linear search.

HDSearch's mid-tier microservice uses LSH, an indexing algorithm that optimally reduces the search space within precise error bounds [45], [46], [52], [65], [70], [76], [100], [114], [115], [119], [120], [122]. We extend the LSH algorithm from the most widely-used open-source k-NN library—the Fast Library for Approximate Nearest Neighbors (FLANN) [107]—into HDSearch's mid-tier. During an offline index construction step, we construct multiple LSH tables for our image corpus. Each LSH table entry contains points that are likely to be near one another in the feature space. Most LSH algorithms use multiple hash tables, and access multiple entries in each hash table, to optimize the performance vs. error trade-off [100].

We extend FLANN's [107] LSH indexes such that the mid-tier microservice does not store feature vectors directly. Rather the LSH tables reference {leaf server, point ID list} tuples, which indirectly refer to feature vectors stored in the leaves.

During query execution, the mid-tier performs look-ups in its in-memory LSH tables to gather potential NN candidates, as shown in Fig. 3. It formulates an RPC request to each leaf microserver with a list of point IDs that may be near the query feature vector. Each leaf calculates distances and returns a distance-sorted list. The mid-tier then merges these

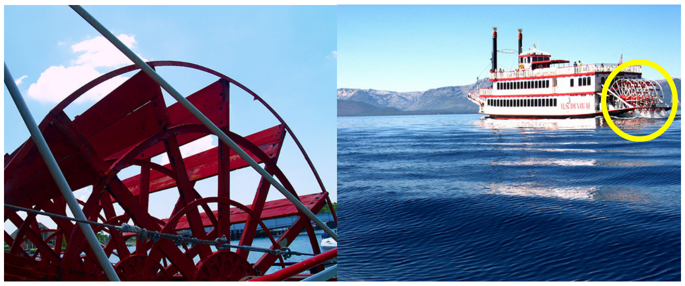

Fig. 4: HDSearch: request (left) and 1-NN response (right): response's highlighted circular segment illustrates why the images matched.

responses and returns the k-NN across all shards. We quantify `HDSearch`'s accuracy in terms of the cosine similarity between the feature vector it reports as the NN for each query and ground truth established by a brute-force linear search of the entire data set. Various LSH parameters can be tuned based on accuracy and latency needs. We tune these LSH parameters to target a sub-ms end-to-end median response time with a minimum accuracy score of 93% across all queries.

Leaf microservice. Distance computations are embarrassingly parallel, and can be accelerated with SIMD, multi-threading, and distributed computing techniques [93]. We employ all of these. We distribute distance computations over many leaves until the computation time and network communication are roughly balanced. Hence, the mid-tier microservice latency, and its ability to fan out RPCs quickly, is so critical: mid-tier microservice and network overheads limit leaf scalability.

Leaf microservers compare query feature vectors against point lists sent by the mid-tier. We use the Euclidean distance metric, which has been shown to achieve a high accuracy [76]. A sample request and response are shown in Figure 4.

B. Router

Memcached-like key-value stores are widely used by OLDI services as they are highly performant and scalable [23]. However, memcached has many drawbacks: (1) its servers are a single point of failure [30] causing frequent fallback to an underlying database access, (2) it is not scalable beyond 200K Queries Per Second (QPS) [23], and (3) it faces network saturation due to network congestion-based timeouts [30]. Memcached must be made as available and performant as the database it assists. These goals can be achieved by distributing load across many memcached servers via efficient routing. Routing-based redundancy can avoid the failure issue.

Related work. McRouter [22] is one such memcached protocol router that helps scale memcached deployments at Facebook. Through efficient routing, McRouter [22] can handle up to 5 billion QPS. It offers connection pooling, prefix routing, replicated pools, production traffic shadowing, online reconfiguration, etc. We introduce a *μSuite* service called `Router` that includes a simplified subset of McRouter's features while still drawing insights from McRouter.

Service description. `Router`'s features include (1) routing key-value stores to memcached deployments, (2) abstracting the routing and redundancy logic from clients, allowing clients to use standard memcached protocols, (3) requiring minimal client modification (i.e., it is a drop-in proxy between the client and memcached hosts), and (4) providing replication-based protocol routing for fault-tolerant memcached deployments.

`Router`'s primary functionality is to route client requests to suitable memcached servers. It supports typical memcached [73] client requests. In this study, we evaluate only `gets` and `sets`. We describe `Router`'s functionality as a series of stages. In the first stage, `Router` parses the clients' requests and forwards them to the route computation code, which uses a proven well-distributed hashing algorithm, SpookyHash [7], to distribute keys from clients' `get` or `set` requests uniformly across destination memcached servers. SpookyHash [7] is a non-cryptographic hash function that is used to produce well-distributed 128-bit hash values for byte arrays of any length. `Router` uses SpookyHash [7] as it (1) enables quick hashing (1 byte/cycle for short keys and 3 bytes/cycle for long keys), (2) can work for any key data type, and (3) incurs a low collision rate. Based on the SpookyHash [7] outcome, `Router` invokes its final stage where it calls internal client code to suitably forward the clients' requests to specific destination memcached servers. The internal client code opens only one TCP connection to a given destination per `Router` thread. All requests sent to that memcached server will share the same connection.

`Router` also provides fault-tolerance for memcached. For large-scale memcached deployments, the frequently-accessed data are read by numerous clients. Too many concurrent client connections may overwhelm a memcached server. Furthermore, ensuring high availability of critical data even when servers go down is challenging. `Router` uses replicated key-value store data pools, detailed below, to solve both these problems.

Front-end microservice. Our front-end microservice provides a client library that transports memcached `get`/`set` requests over a gRPC [15] interface. We do not study the front-end in this work. We emulate a large pool of `Router` clients using a synthetic load generator that picks key or key-value pair queries from an open-source "Twitter" data set [71]. The load generator's `get` and `set` request distributions mimic YCSB's Workload A [63] with 50/50 `gets` and `sets`.

Mid-tier microservice. The mid-tier uses SpookyHash to distribute keys uniformly across leaves and then routes `get` or `set` requests as shown in Fig. 5. `Router` uses replication both to spread load and to provide fault tolerance. `Router`'s mid-tier forwards `sets` to a fixed number of leaves (i.e., a replication pool; three replicas in our experiments), allowing the same data to reside on several leaves. The mid-tier randomly picks a leaf replica to service `get` requests, balancing load across leaves.

Leaf microservice. The leaf microserver uses gRPC [15] to build a communication wrapper around a memcached [73] server process. The leaf microservice is written such that it can handle multiple concurrent requests from several mid-tier microservices. The leaf uses `gRPC` APIs to receive the mid-tier's `get` and `set` queries. It then rewrites received queries to suitably query its local memcached server. The memcached server's responses

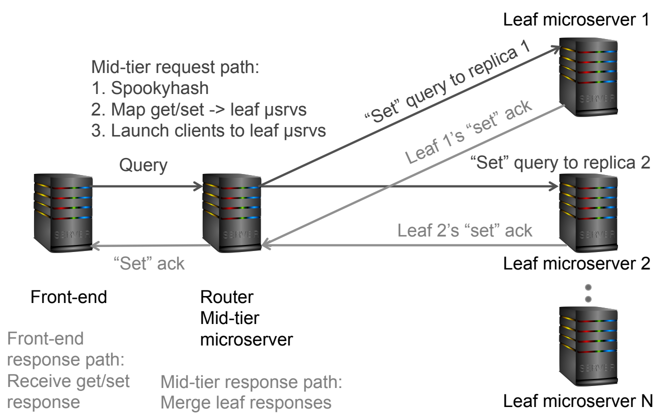

Fig. 5: Router: back end request and response pipelines.

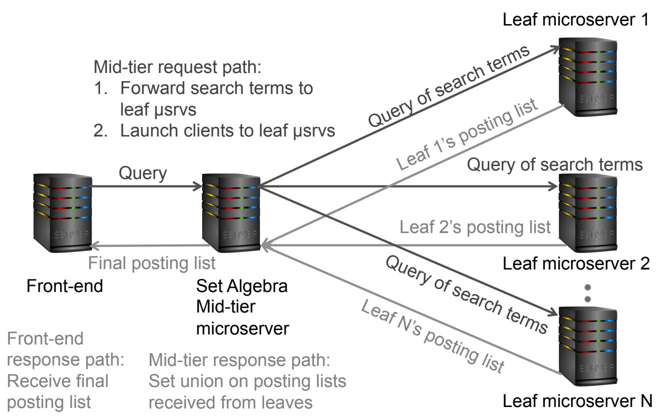

Fig. 6: Set Algebra: back end request and response pipelines.

are then sent to the mid-tier via the gRPC [15] response.

C. Set Algebra

Fast processing of set intersections is a critical operation in many query processing tasks in the context of databases and information retrieval. For example, in the database realm, set intersections are used for data mining, text analytics, and evaluation of conjunctive predicates. They are also the key operations in enterprise and web search.

Related work. Many open-source web search platforms, such as Lucene [103] and CloudSuite's Web Search [71], perform set intersections for document retrieval. But, these monolithic web searches face response latencies ≥ 100 ms as they perform many other tasks (querying a database, scoring page ranks, custom filtering, etc.) apart from set intersections. Hence, these searches are unsuitable for characterizing microservice OS/network overheads. While `Set Algebra` draws algorithmic insights from these works [71], [103], its microservices only perform set intersections to achieve single-digit ms tail latencies.

Service description. `Set Algebra` performs document retrieval for web search by performing set intersections on posting lists. The posting list of each term is a sorted list of document identifiers that is stored as a skip list [112]. A skip is a pointer i→j between two non-consecutive documents i and j in the posting list. The number of documents skipped between i and j is defined as the skip size. For a term t, the posting list $L(t)$ is a tuple (S_t, C_t) where $S_t = s_1, s_2, ..., s_k$ is a sequence of skips and C_t contains the remaining documents (between skips). These remaining documents can be stored using different compression schemes [132] where decompression can be handled by a separate microservice. Skips are typically used to speed up list intersections.

`Set Algebra` searches through a corpus of 4.3 million WikiText documents (of size ~ 10 GB) randomly drawn from Wikipedia [41] and sharded uniformly across leaves, to return documents containing all search terms to the client. Leaves index posting lists for each term in their sharded document corpus. We reduce leaf computations by excluding extremely common terms, called *stop words*, that have little value in helping select documents matching a user's need from the leaves' inverted index. `Set Algebra` determines a stop list by sorting terms by their collection frequency (the total number of times each term appears in the document collection), and then regarding the most frequent terms as a stop list. Members of the stop list are discarded during indexing.

Front-end microservice. We synthetically emulate multiple clients via a load generator that picks search queries from a query set. Each search query typically spans ≤ 10 words [6]. We synthetically generate a query set of 10K queries, based on Wikipedia's word occurrence probabilities [41].

Mid-tier microservice. The mid-tier forwards client queries of search words/terms to the leaves, which return intersected posting lists to the mid-tier, as portrayed in Fig. 6. It then merges intersected posting lists received from all leaves via set union operations and sends the outcome to the client.

Leaf microservice. The leaf microservice performs actual set intersections. Leaves hold ordered posting lists as an inverted index where documents are identified via a document ID, and for each term t, the inverted index is a sorted list of all document IDs containing t. Using this representation, the leaves intersect two sets $L1$ and $L2$ using a linear merge by scanning both lists in parallel, requiring an $O(|L1|+|L2|)$ time complexity ("merge" step in merge sort). The resulting intersected posting list is then passed to the mid-tier.

D. Recommend.

Recommendation systems help real-world services generate revenue, notably in the fields of e-commerce and behavior prediction [10]. Many web companies use smart recommender engines that study prior user behaviors to provide preference-based data, such as relevant job postings, movies of interest, suggested videos, friends users may know, items to buy, etc.

Related work. Many open-source recommendation engines such as PredictionIO [31], Raccoon [32], HapiGER [17], EasyRec [2], Mahout [21], Seldon [37], etc., use various recommendation algorithms. But, they lack the distributed microservice structure (i.e, front-end, mid-tier, and leaf) that we aim to study. So, we build `Recommend` using the state-of-the-art fast, flexible open-source ML library—`mlpack` [64] such that `Recommend` is composed of distributed microservices.

Service description. `Recommend` is a recommendation service that uses numerous users' overall preference to predict user ratings for specific items. For each {user, item} query pair, `Recommend` performs user-based collaborative filtering to predict the user's preference for that item, based on how similar users ranked the item. Collaborative filtering is typically performed on {user, item, rating} tuple data sets. Our collaborative filtering technique has three stages of (1) sparse matrix composition, (2) matrix factorization, and (3) rating approximations for missing entries in the sparse matrix (e.g., a movie that a user has not rated) via a neighborhood algorithm.

Sparse matrix composition. `Recommend`'s data set is 10K {user, item, rating} tuples from the `MovieLens` [78] movie recommendation data set. We represent the data set as a sparsely populated user-item rating matrix $V \in R^{m \times n}$—the *utility matrix*—where m is the number of users and n is the number of items (i.e., movies) in the data set. Hence, V_{ij} (if known), represents the rating of movie j by user i. Each user typically rates a small subset of movies. Many techniques address the *cold start problem* of recommending to a fresh user with no prior ratings. For simplicity, `Recommend` only focuses on users for whom the system has at least one rating.

Matrix factorization. Collaborative filtering often uses matrix factorization. For instance, a matrix factorization model won the Netflix Challenge in 2009 [91]. Matrix factorization's goal is to reduce the sparse user-item rating utility matrix V's dimensionality and to aid similarity identification. We decompose the sparse low-rank matrix V into two "user" and "item" matrices W and H. These decomposed matrices aim to approximate missing values in the utility matrix V.

We employ Non-negative Matrix Factorization (NMF) to decompose V. NMF performs $V \approx WH$ to create two non-negative matrix factors W and H of V. NMF approximately factorizes V into an $m \times r$ matrix W and $r \times n$ matrix H.

$$V = \begin{bmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,n} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{m,1} & v_{m,2} & \cdots & v_{m,n} \end{bmatrix} = WH \; where,$$

$$W_{m \times r} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,r} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,r} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,r} \end{bmatrix}, \; H_{r \times n} = \begin{bmatrix} h_{1,1} & h_{1,2} & \cdots & h_{1,n} \\ h_{2,1} & h_{2,2} & \cdots & h_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ h_{r,1} & h_{r,2} & \cdots & h_{r,n} \end{bmatrix}$$

Dimension r is V's rank, and it represents the number of similarity concepts NMF identifies [67]. For example, one similarity concept may be that some movies belong to the "comedy" category, while another may be that most users that liked the movie "Harry Potter 1" also liked "Harry Potter 2". W captures the correlation strength between a row of V and a similarity concept—it expresses how users relate to similarity concepts such as preferring "comedy" movies. H captures the correlation strength of a column of V to a similarity concept—it identifies the extent to which a movie falls in the "comedy" category. The NMF representation, hence, results in a compressed form of the user-item rating utility matrix V.

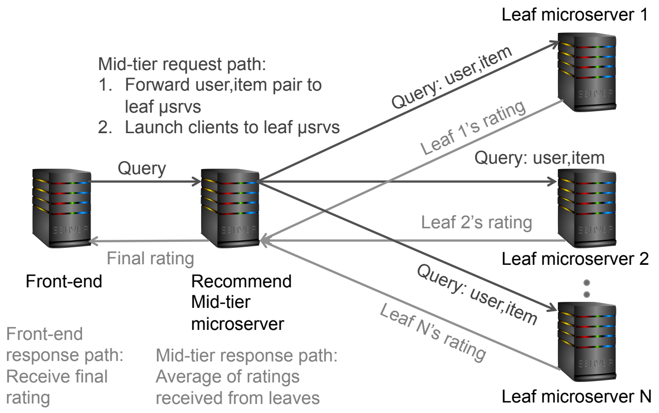
Leaf microserver 1
Mid-tier request path:
1. Forward user,item pair to leaf μsrvs
2. Launch clients to leaf μsrvs
Query: user,item
Leaf 1's rating
Query
Query: user,item
Leaf 2's rating
Final rating
Leaf microserver 2
Front-end
Recommend Mid-tier microserver
Query: user,item
Leaf N's rating
Front-end response path: Receive final rating
Mid-tier response path: Average of ratings received from leaves
Leaf microserver N

Fig. 7: Recommend: back end request and response pipelines.

Neighborhood algorithm. The NMF decomposed matrix is used to approximate missing movie ratings in the user-item rating utility matrix V. We also remember the initial movies that the users rated. We use a neighborhood algorithm, allknn [64], which relies on similarity measures such as cosine, Pearson, Euclidean, etc., to generate ratings for movies in a user's neighborhood. This algorithm can also be further extended to recommend items which were not rated by the user.

Front-end microservice. We emulate multiple `Recommend` clients via a load generator that picks 1K {user, item} query pairs from the `MovieLens` [78] movie recommendation data set. The {user, item} query pairs are different from the user→item rating mappings present in the data set (utility matrix). In other words, the load generator always picks queries from the "empty" cells of the utility matrix V so that we do not test on the same data that `Recommend` trained on.

Mid-tier microservice. `Recommend` uses the mid-tier microservice primarily as a forwarding service, as shown in Fig. 7. The mid-tier microserver receives {user, item} query pairs from the client and forwards them to the leaves. Item ratings returned by the leaves are then averaged and sent back to the client.

Leaf microservice. Leaves perform collaborative filtering by first performing sparse matrix composition and matrix factorization offline. During run-time, they perform collaborative filtering on their corresponding matrix V's shard using the allknn neighborhood approach [64], to predict movie ratings. Rating predictions are then sent to the mid-tier.

IV. μSUITE: FRAMEWORK DESIGN

In this section, we discuss the software designs used to build *μSuite*'s mid-tier microservers.

Thread-pool architecture. *μSuite* has a thread-pool architecture that concurrently executes requests by judiciously "parking" and "unparking" threads to avoid thread creation, deletion, and management overheads. *μSuite* uses thread-pool architectures (vs. architectures like thread-per-connection), as thread-pool architectures scale better for microservices [95].

We describe the following *μSuite* framework designs with the aid of a simple figure (Fig. 8) of a three-tier service with a single client, mid-tier, and leaf.

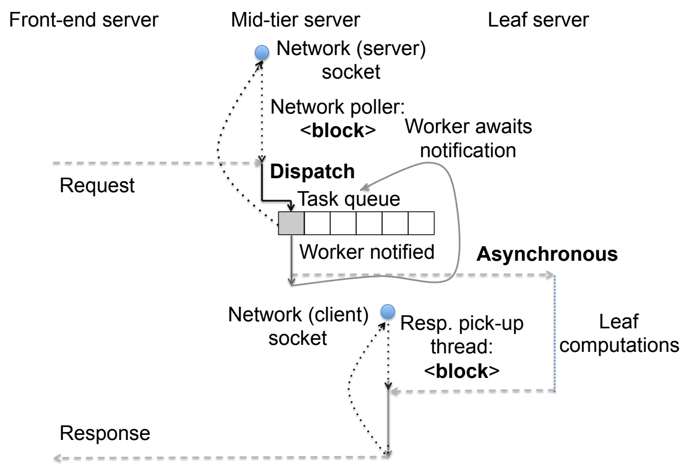

Fig. 8: μSuite's mid-tier microservice design.

Blocking on the front-end network socket. *μSuite*'s blocking design comprises *network poller* threads awaiting new work from the front-end 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. Blocking designs conserve precious CPU resources by avoiding wasting CPU time in fruitless poll loops, unlike poll-based designs. Hence, services such as Redis BLPOP [33] employ a block-based design.

Asynchronous communication with leaf microservers. There is no association between an execution thread and a particular RPC—all RPC state is explicit. Asynchronous services are event-based—an event, such as the completion of an outgoing leaf request, arrives on any leaf response reception thread and is matched to a particular parent RPC through a shared data structure. Hence, mid-tier microservers can proceed to process successive requests after sending requests to leaf microservers. We build *μSuite* asynchronously to leverage the greater performance efficiency that asynchronous designs offer compared to synchronous ones [128]. For this reason, several cloud applications such as Apache [4], Azure blob storage [24], Redis replication [34], Server-Side Mashup, CORBA Model, and Aerospike [5] are built asynchronously.

Dispatch-based processing of front-end requests. *μSuite*'s dispatch-based design separates responsibilities between network threads, which accept new requests from the underlying RPC interface, and worker threads, which execute RPC handlers. Network threads dispatch the RPC to a worker thread pool by using producer-consumer task-queues and signalling on condition variables. Workers pull requests from task queues, and then process them by forking for fan-out and issuing asynchronous leaf requests. A worker then goes back to blocking on the condition variable to await new work. To aid non-blocking calls to both leaves and front-end microservers, we add another thread pool that exclusively handles leaf server responses. These 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). Several cloud applications such as IBM's WebSphere for z/OS [40], [83], Oracle's EDT image search [39], Mule ESB [9], Malwarebytes [16], Celery for RabbitMQ and Redis [8], Resque [35] and RQ [36] Redis queues, and NetCDF [74] are dispatch-based.

TABLE II: Mid-tier microservice hardware specification.

Processor	Intel Gold 6148 CPU "Skylake"
Clock frequency	2.40 GHz
Cores / HW threads	40 / 80
DRAM	64 GB
Network	10 Gbit/s
Linux kernel version	4.13.0

V. Methodology

In this section, we describe the experimental setup that we use to characterize *μSuite*'s OS and network overheads.

We characterize each service in terms of its constituent mid-tier and leaf microservices. We use service-specific synthetic load generators that mimic many end-users to issue queries to the mid-tier. These load generators are operated in a closed-loop mode to establish each service's peak sustainable throughput. We measure tail latencies by operating the load generators in an open-loop mode, selecting inter-arrival times from a Poisson distribution [59]. Load generators are run on separate hardware and we validated that neither the load generator nor the network bandwidth is a performance bottleneck in our experiments. We average measurements over five trials.

We run experiments on a distributed system of 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`. Our setup's hardware configuration is shown in Table II. The leaves run within Linux `tasksets` limiting them to 18 logical cores for `HDSearch`, `Set Algebra`, and `Recommend` and 4 logical cores for `Router`. Each microservice runs on dedicated hardware. The mid-tier is not CPU bound; peak-load performance is limited by leaf CPU.

On this setup, we run load generators in open loop mode to characterize OS and network overheads for various loads. We use the eBPF [1] `syscount` tool to first characterize system call invocations for the mid-tier. We then study request latency breakdowns incurred within the OS (e.g., interrupt handler latency for network-based hard interrupts and scheduler-based soft interrupts, time to switch a thread from "active" to "running" state, etc.) using eBPF's [1] `hardirqs`, `softirqs`, and `runqlat` tools. We report network delays in terms of the number of TCP re-transmissions measured using eBPF [1]'s `tcpretrans` tool. Additionally, we use Linux's `perf` utility to profile context switch overheads faced by the mid-tier. We use Intel's HITM (as in hit-Modified) PEBS coherence event, to detect true sharing of cache lines; an increase in HITMs indicates a corresponding increase in lock contention [99].

VI. Results

A. Saturation throughput

Production services typically saturate at tens of thousands QPS [13]. *μSuite* aims to be representative of production

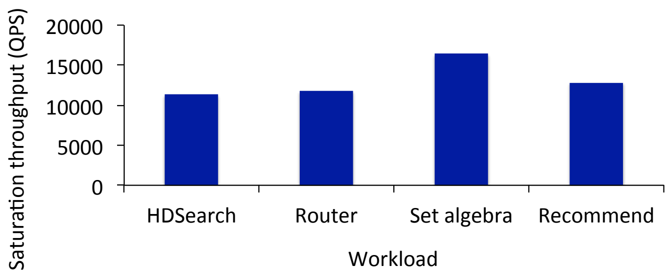

Fig. 9: Saturation throughput (QPS): *μSuite* is similar to real-world services.

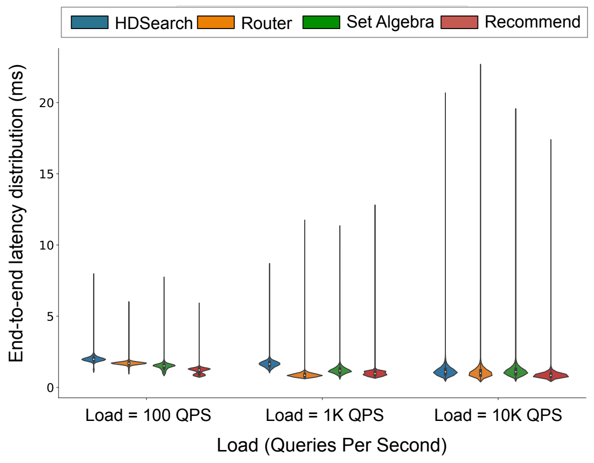

Fig. 10: End-to-end response latency across different loads for each benchmark: median latency is higher at low load.

services by achieving a similar saturation throughput range for each of its benchmark services. Using our load generator in closed-loop mode, we measure the saturation throughput for all benchmarks. We find that `HDSearch` saturates at $\sim$ 11.5K QPS, `Router` at $\sim$ 12K QPS, `Set Algebra` at $\sim$ 16.5K QPS, and `Recommend` at $\sim$ 13K QPS, as shown in Fig. 9. Hence, we find that *μSuite* is sufficiently representative of production services.

B. End-to-end response latency

Many OLDI services face (1) drastic diurnal load changes [81], (2) load spikes due to "flash crowds" (e.g., traffic after a major news event), or (3) explosive customer growth that surpasses capacity planning (e.g., the Pokemon Go [29] launch). Supporting wide-ranging loads aids rapid OLDI service scale-up. Furthermore, we cannot meaningfully measure latency at saturation, as the offered load is unsustainable and queuing grows unbounded. Hence, we characterize *μSuite*'s end-to-end (across mid-tier and leaves) response latency vs. load trade-off (Fig. 10) across wide-ranging loads up to saturation—100 QPS, 1K QPS, and 10K QPS.

Each end-to-end response latency distribution is portrayed as a violin plot with the bars in the violin centers representing the median latency and the thin black lines representing the higher-order tail latency. While the tail latency increases with an increase in load, we find that the median latency at 100 QPS is up to 1.45× higher than the median latency at 1,000 QPS since there is better temporal locality at a higher load and the OS

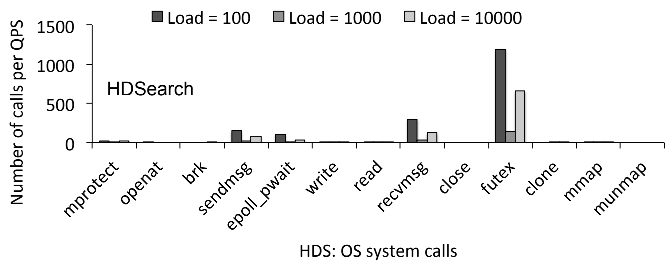

Fig. 11: HDSearch's counts of OS system call invocations per QPS: the *futex* system call is predominantly invoked.

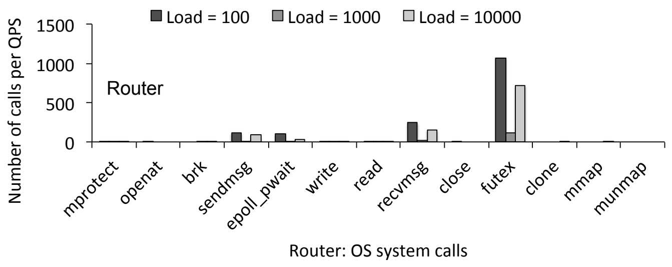

Fig. 12: Router's counts of OS system call invocations per QPS: the *futex* system call is predominantly invoked.

does not tend to "sleep" longer before waking-up threads. We explain this behavior further when we subsequently characterize *μSuite*'s OS and network overheads. Additionally, we note that the worst-case end-to-end tail latency is never more than 22 ms for any service, implying that the constituent microservices face a worst-case tail latency of at most a few single-digit ms.

C. OS and network overheads

For each service, we show a breakdown of (1) number of invocations of heavily-invoked system calls, (2) latency distributions across OS and network stacks, (3) network delays due to TCP re-transmissions, and (4) OS-induced effects such as context switches and thread contention. As before, we characterize the OS and network overheads at three distinct loads of 100 QPS, 1,000 QPS and 10,000 QPS.

System call invocations. We first analyze various system call invocation distributions per QPS for *μSuite* in Figs. 11, 12, 13, 14. We find that *futex* (*fast userspace mutex*) system calls are invoked most frequently by all services. Our services involve (1) network threads locking the front-end query reception network sockets, (2) response threads locking the leaf response reception network socket, and (3) worker threads blocking on producer-consumer task queues via condition variables, while awaiting new work. These high-level locking abstractions result in several *futex* system call invocations. Furthermore, we find, much like the end-to-end median latency distribution, futex invocations per QPS are higher at low load. At low load, several threads invoke futex(), but, only one thread successfully acquires the synchronization object the futex() protects (e.g., network socket lock). The remaining threads wake up and try to acquire the network socket lock via further `futex()` calls. Hence, for a small queries per second count (low load), a relatively large number of futex() calls are invoked by various thread pools.

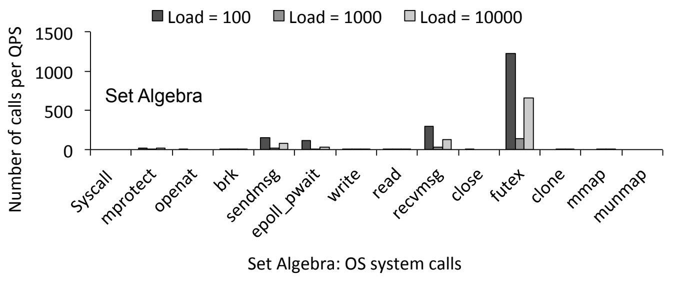

Fig. 13: Set Algebra's counts of OS system call invocations per QPS: the *futex* system call is predominantly invoked.

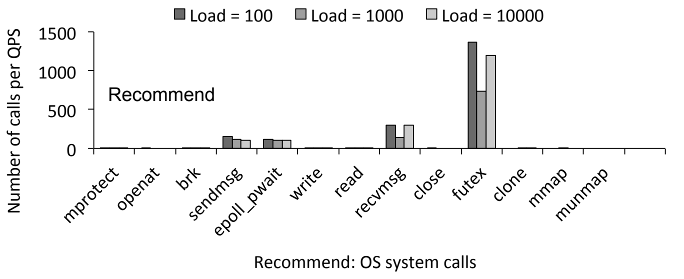

Fig. 14: Recommend's counts of OS system call invocations per QPS: the *futex* system call is predominantly invoked.

We also see several *sendmsg*, *recvmsg*, and *epoll_pwait* invocations as these system calls are regularly invoked by blocking network/worker/response threads to send/receive RPCs on the incoming/outgoing network sockets.

Overheads due to OS operations. We next scrutinize latency distributions of fine-grained OS operations performed while serving mid-tier requests. We study these latencies across various loads for each service, as shown in Figs. 15, 16, 17, 18. In each graph, the x-axis represents various OS overheads, and the y-axis shows the latency distribution across all mid-tier requests served in a 30s time frame, represented as a violin plot. The various OS overheads are: (1) *Hardirq*—interrupt handler latency while receiving hard network interrupts, (2) *Net_tx*—soft interrupt handler latency while sending network messages, (3) *Net_rx*—soft interrupt handler latency while receiving network messages, (4) *Block*—soft interrupt handler latency while a thread enters the "blocked" state, (5) *Sched*—soft interrupt handler latency while triggering scheduling actions, (6) *RCU*—soft interrupt handler latency for read-copy-updates, (7) *Active-Exe*—time from when a thread enters the *active* or *runnable* state to when it starts running on a CPU, and (8) *Net*—net mid-tier latency.

We find that μ*Suite*'s mid-tier tail latencies arise mainly from the OS scheduler. *Active-Exe* contributes to mid-tier tails by up to $\sim$50% for `HDSearch`, $\sim$75% for `Router`, $\sim$87% for `Set Algebra`, and $\sim$64% for `Recommend`. These latencies are a part of thread wakeup delays and can arise from (1) network thread wakeups via interrupts on query arrivals, (2) worker wakeups upon RPC dispatch, or (3) response thread wakeups upon leaf response arrivals. We also see some *Sched* overheads.

Context switch and thread contention overheads. We next analyze μ*Suite*'s context switch (CS) and thread contention

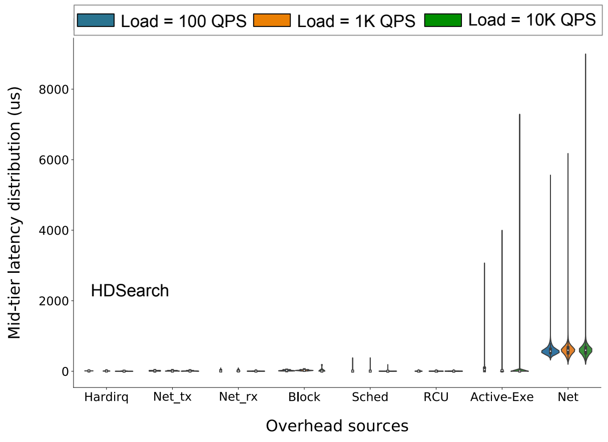

Fig. 15: HDSearch's breakdown of OS overheads: time to switch a thread from the *active* to the *running* state is high.

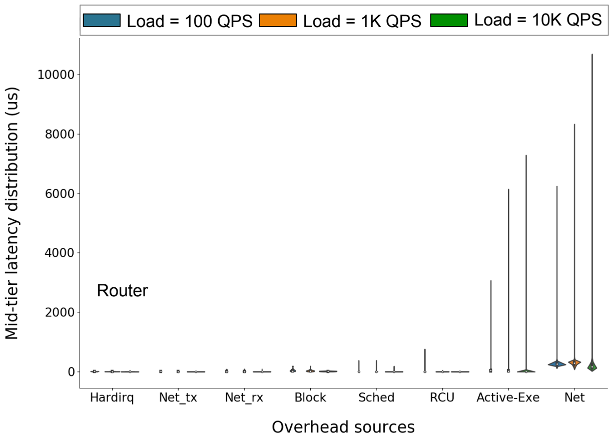

Fig. 16: Router's breakdown of OS overheads: time to switch a thread from the *active* to the *running* state is high.

(HITM) overheads across all three loads in Fig. 19. We find that the CS and HITM overheads are similar for all μ*Suite* services ("HDS"—`HDSearch`, "SA"—`Set Algebra`, and "Rec"—`Recommend`)—both overheads increase as load increases. HITM counts are more than CS counts as various threads are woken up when a *futex* returns, and they all contend with each other while trying to acquire a network socket lock.

Additionally, we see only a single-digit number of TCP re-transmissions for all services, and hence do not report it.

VII. Discussion

We briefly discuss how μ*Suite* can facilitate future research.

Latency degradation caused by blocking designs. μ*Suite* blocks on the front-end request reception network socket and leaf response reception sockets. We choose this design since polling can be prohibitively expensive as it wastes CPU time in fruitless poll loops, degrading a data center's energy efficiency. However, our results show that blocking incurs OS-induced thread wakeup latencies that significantly degrade microservice tail latency. Hence, it would be interesting to explore policies that trade-off blocking vs. polling, either statically or dynamically.

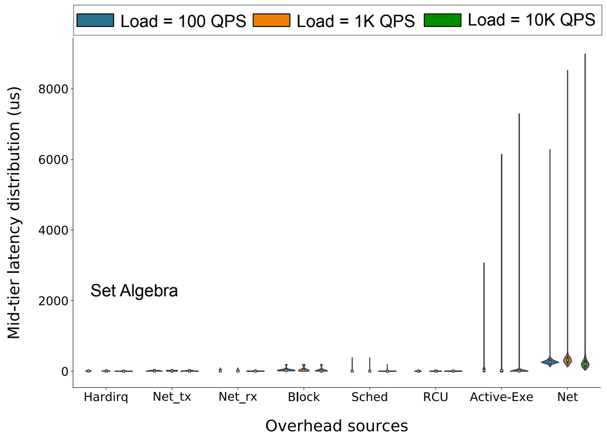

Fig. 17: Set Algebra's breakdown of OS overheads: time to switch a thread from the *active* to the *running* state is high.

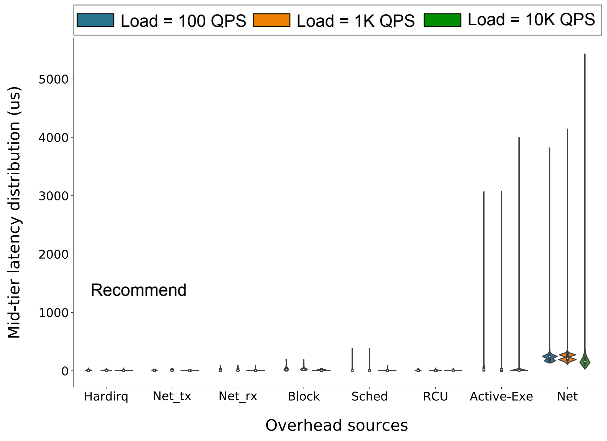

Fig. 18: Recommend's breakdown of OS overheads: time to switch a thread from the *active* to the *running* state is high.

Future microservice monitoring systems could then dynamically switch between block- and poll-based designs.

Thread wakeups due to dispatch-based designs. Thread wakeup latencies that dominate microservice latency tails arise from both (1) network threads picking up requests from the front-end socket and (2) workers waking up to receive dispatched requests. In-line designs that avoid explicit state hand-off and thread-hops to pass work from network to worker threads may avoid expensive thread wakeups. However, in-line models are only efficient at low loads and for short requests where dispatch overheads undermine service times. Since leaf nodes computationally dominate in most distributed services, most mid-tiers can benefit from dispatching. Moreover, if a single in-line thread cannot sustain the service load, multiple in-line threads contending for work will typically outweigh the dispatch design's hand-off costs, which can be carefully tuned. Additionally, in-line models are prone to high queuing, as each thread processes whichever request it receives. In contrast, dispatched models can explicitly prioritize requests. Exploring policies that trade-off in-line vs. dispatched designs can help build a dynamic adaptation system that judiciously chooses to

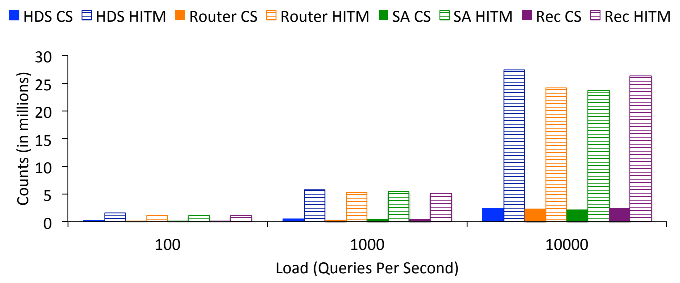

Fig. 19: Context switches (CS) and thread contention (HITM) incurred (in millions) for each benchmark across diverse loads: thread contention is high.

dispatch requests. It may also be interesting to study request dispatch ways to identify optimal microservice dispatch designs.

Thread pool sizing. *µSuite*'s design supports large thread pools that sustain peak loads by "parking" or "unparking" on condition variables, as needed. However, these large pools can contend on (1) the front-end socket while receiving requests, (2) the producer-consumer task queue while picking up dispatched requests, or (3) the leaf response reception socket. Hence, a user-level thread scheduler that dynamically selects suitable thread pool sizes can reduce thread contention and improve scalability. While prior works [77], [89] propose several ways to dynamically scale thread pools, it will still be interesting to study such schedulers in the context of microservices.

VIII. Conclusion

Prior works study monolithic services, where sub-ms-scale overheads are insignificant. Faster I/O and lower latency in microservices call for characterizing the sub-ms-regime's OS and network effects. However, widely-used open-source benchmark suites are unsuitable for this characterization as they use monolithic architectures and incur ≥ 100 ms service latencies. Hence, in this work, we presented *µSuite*, a suite of microservice benchmarks that allow researchers to explore performance, power, micro-architectural, etc., overheads in modern microservices. Our results demonstrated how *µSuite* can be used to characterize microservice performance. Our OS and network characterization revealed that the OS scheduler can influence microservice tail latency by up to 87%.

IX. Acknowledgement

We would like to thank our shepherd, Dr. Ravi Soundararajan, and the anonymous reviewers for their valuable feedback. We also acknowledge Amrit Gopal, P.R. Sriraman, Rajee Sriraman, Akshay Sriraman, and Vaibhav Gogte for their unwavering support. This work was supported by NSF Grant IIS1539011 and gifts from Intel.

References

[1] Linux bcc/BPF Run Queue Latency. http://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html. [Accessed 19-Nov-2017].
[2] Add reco. to website. www.easyrec.org. [Accessed 4/27/2018].
[3] Amazon. https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/. [Accessed 4/27/2018].

[4] Apache http server. https://httpd.apache.org/. [Accessed 11/7/2017].
[5] Asynchronous API. https://www.aerospike.com/docs/client/java/usage/async/. [Accessed 11/7/2017].
[6] Average number of search terms for online search queries in the US. https://www.statista.com/statistics/269740/number-of-search-terms-in-internet-research-in-the-us/. [Accessed 4/20/2018].
[7] Bob Jenkins. SpookyHash: a 128-bit noncryptographic hash. http://burtleburtle.net/bob/hash/spooky.html. [Accessed 3/1/2012].
[8] Celery. http://www.celeryproject.org/. [Accessed 11/7/2017].
[9] Chasing the bottleneck. https://blogs.mulesoft.com/biz/news/chasing-the-bottleneck-true-story-about-fighting-thread-contention-in-your-code/. [Accessed 11/7/2017].
[10] Collaborative filtering via matrix decomposition in mlpack. www.ratml.org/pub/pdf/2015collaborative.pdf. [Accessed 4/27/2018].
[11] Faban. http://faban.org. [Accessed 27-Apr-2018].
[12] Gilt. www.infoq.com/presentations/scale-gilt. [Accessed 4/27/2018].
[13] Google Search Statistics. http://www.internetlivestats.com/google-search-statistics/. [Accessed 11/7/2017].
[14] Google's Similar Images. https://googleblog.blogspot.com/2009/10/similar-images-graduates-from-google.html. [Accessed 11/7/2017].
[15] gRPC. https://github.com/heathermiller/dist-prog-book/blob/master/chapter/1/gRPC.md. [Accessed 11/7/2017].
[16] Handling 1M Requests Per Minute. http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang/. [Accessed 11/7/2017].
[17] Hapiger. https://github.com/grahamjenson/hapiger. [Accessed 4/27/2018].
[18] Haskell. https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/. [Accessed 4/27/2018].
[19] 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. [Accessed 11/7/2017].
[20] Linkedin. www.infoq.com/presentations/linkedin-microservices-urn. [Accessed 4/27/2018].
[21] Mahout. http://mahout.apache.org/. [Accessed 4/27/2018].
[22] McRouter. https://github.com/facebook/mcrouter. [Accessed 11/7/2017].
[23] Memcached performance. https://github.com/memcached/memcached/wiki. [Accessed 4/27/2018].
[24] Microsoft Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs. [Accessed 11/7/2017].
[25] Myrocks. https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/. [Accessed 4/27/2018].
[26] Netflix. www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/. [Accessed 4/27/2018].
[27] OpenImages: A public dataset for large-scale multi-label and multi-class image classification. https://github.com/openimages/dataset.
[28] PerfKit. https://github.com/GoogleCloudPlatform/PerfKitBenchmarker. [Accessed 11/19/2017].
[29] 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. [Accessed 11/7/2017].
[30] The power of the proxy: Request routing memcached. https://dzone.com/articles/the-power-of-the-proxy-request-routing-memcached. [Accessed 4/27/2018].
[31] Pred.io. http://predictionio.apache.org/index.html. [Accessed 4/27/2018].
[32] Raccoon. www.npmjs.com/package/raccoon. [Accessed 4/27/2018].
[33] Redis BLPOP. https://redis.io/commands/blpop. [Accessed 11/7/2017].
[34] Redis Rep. https://redis.io/topics/replication. [Accessed 11/7/2017].
[35] Resque. https://github.com/defunkt/resque. [Accessed 11/7/2017].
[36] RQ. http://python-rq.org/. [Accessed 11/7/2017].
[37] Seldon. www.seldon.io/. [Accessed 4/27/2018].
[38] Soundcloud. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith. [Accessed 4/27/2018].
[39] SwingWorker in Java. www.oracle.com/technetwork/articles/javase/. [Accessed 11/7/2017].
[40] Websphere. https://www.ibm.com/developerworks/community/blogs/aimsupport/entry/dispatch_timeout_handling_in_websphere_application_server_for_zos?lang=en. [Accessed 11/7/2017].
[41] Wiki. https://en.wikipedia.org/w/index.php?title=Plagiarismoldid=5139350. [Accessed 4/20/2018].
[42] Thrift. https://github.com/facebook/fbthrift, 2017. [Accessed 11/7/2017].
[43] 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. *CoRR*, 2016.
[44] Alexa. Alexa, the web information company. [Accessed 27-Apr-2018].
[45] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, 2006.
[46] 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.
[47] I. Arapakis, X. Bai, and B. B. Cambazoglu. Impact of response latency on user behavior in web search. In *SIGIR Conference on Research &#38; Development in Information Retrieval*, 2014.
[48] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan. Memory hierarchy for web search. In *High Performance Computer Architecture*, 2018.
[49] M. Barhamgi, D. Benslimane, and B. Medjahed. A query rewriting approach for web service composition. *IEEE Transactions on Services Computing*, 2010.
[50] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 2017.
[51] L. A. Barroso, J. Dean, and U. HÃűlzle. Web search for a planet: The google cluster architecture. In *IEEE Micro*, 2003.
[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] S. Berchtold, C. Bohm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *Data Engineering*, 2000.
[54] S. Berchtold, D. Keim, and H. Kriegel. An index structure for high-dimensional data. *Multimedia computing and networking*, 2001.
[55] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, 2008.
[56] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 2001.
[57] A. Bouch, N. Bhatti, and A. Kuchinsky. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *Conference on Human Factors and Computing Systems*, 2000.
[58] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, and H. C. Li. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.
[59] J. Cao, M. Andersson, C. Nyberg, and M. Kihl. Web server performance modeling using an m/g/1/k* ps queue. In *Telecommunications*, 2003.
[60] J. L. Carlson. *Redis in Action*. 2013.
[61] G.-H. Cha and C.-W. Chung. The gc-tree: a high-dimensional index structure for similarity search in image databases. *IEEE transactions on multimedia*, 2002.
[62] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ACM SIGARCH Computer Architecture News*, 2013.
[63] 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.
[64] 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.
[65] 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.
[66] J. Dean and L. A. Barroso. The tail at scale. *Commn. of ACM*, 2013.
[67] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.
[68] H. Ding, Y. Liu, L. Huang, and J. Li. K-means clustering with distributed dimensions. In *ICML*, 2016.
[69] N. Dmitry and S.-S. Manfred. On micro-services architecture. *International Journal of Open Information Technologies*, 2014.
[70] 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.
[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 *ASPLOS*, 2012.

[72] X. Z. Fern and C. E. Brodley. Random projection for high dimensional data clustering: A cluster ensemble approach. In *ICML*, 2003.

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

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

[75] B. Georgescu, I. Shimshoni, and P. Meer. Mean shift based clustering in high dimensions: A texture classification example. In *ICCV*, 2003.

[76] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[77] 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 *ASPLOS*, 2015.

[78] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 2015.

[79] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, R. Dreslinski, T. Mudge, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *ISCA*, 2015.

[80] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Liv, A. Rovinski, A. Khurana, R. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ASPLOS*, 2015.

[81] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, and A. Kalro. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA*, 2018.

[82] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comp. Arch. News*, 2006.

[83] E. N. Herness, R. J. High, and J. R. McGee. Websphere application server: A foundation for on demand computing. *IBM Sys. Journal*, 2004.

[84] 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 *HPCA*, 2015.

[85] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[86] K. Kailing, H.-P. Kriegel, and P. Kröger. Density-connected subspace clustering for high-dimensional data. In *Proc. SDM*, 2004.

[87] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *IISWC*, 2014.

[88] H. Kasture and D. Sanchez. Tailbench: A benchmark suite and evaluation methodology for latency-critical applications. In *IISWC*, 2016.

[89] 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 *WSDM*, 2015.

[90] R. Kohavi, R. M. Henne, and D. Sommerfield. Practical guide to controlled experiments on the web. In *KDD*, 2007.

[91] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009.

[92] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *STOC*, 2000.

[93] V. T. Lee, C. C. del Mundo, A. Alaghi, L. Ceze, M. Oskin, and A. Farhadi. NCAM: near-data processing for nearest neighbor search. *CoRR*, 2016.

[94] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 2002.

[95] Y. Ling, T. Mullen, and X. Lin. Analysis of optimal thread pool size. *SIGOPS Oper. Syst. Rev.*, 2000.

[96] T. Liu, A. W. Moore, K. Yang, and A. G. Gray. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004.

[97] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *ACM SIGARCH Computer Architecture News*. IEEE Press, 2014.

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

[99] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. LASER: Light, Accurate Sharing dEtection and Repair. In *High-Performance Computer Architecture (HPCA)*, 2016.

[100] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

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

[102] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, 2000.

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

[104] J. McNames. A fast nearest-neighbor algorithm based on a principal axis search tree. *TPAMI*, 2001.

[105] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *ISCA*, 2011.

[106] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2009.

[107] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *TPAMI*, 2014.

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

[109] 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 *NSDI*, 2013.

[110] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: a review. *KDD*, 2004.

[111] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *SoCC*, 2015.

[112] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *CACM*, 1990.

[113] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *VLDB*, 2000.

[114] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *ICCV*, 2003.

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

[116] 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 (WDDD)*, 2017.

[117] A. Sriraman and T. F. Wenisch. μTune: Auto-tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[118] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *CoRR*, 2015.

[119] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, 2009.

[120] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *TODS*, 2010.

[121] G. Tene. How not to measure latency. In *Low Latency Summit*, 2013.

[122] K. Terasawa and Y. Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In *WADS*, 2007.

[123] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *ecs*, 2007.

[124] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *SIGCOMM*, 2012.

[125] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *MICRO*, 2015.

[126] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and microservice architecture pattern to deploy web applications in the cloud. In *10CCC*, 2015.

[127] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, and S. Zhang. Bigdatabench: A big data benchmark suite from internet services. In *HPCA*, 2014.

[128] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

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

[130] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *ISCA*, 2016.

[131] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News*, 2016.

[132] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.