μSuite & μTune: Auto-Tuned Threading for OLDI Microservices

Akshitha Sriraman, Thomas F. Wenisch
University of Michigan
On-Line Data Intensive (OLDI) Services

Must meet stringent Service Level Objectives (SLOs)
OLDI: From Monoliths to Microservices

Monolithic service

Scaling

From >100ms SLOs to sub-ms SLOs

Microservices

Scaling

RPC
Tail Latency

• SLOs are impacted by the 99\textsuperscript{th}+% (tail) latency
• Negatively affects user experience

Goal: Minimize microservice tail latency
Threading Effects on Tails for Monoliths

• Our focus: Sub-ms overheads due to threading design

- Lock contention
- Thread wakeups
- Spurious context switch

Threading-induced OS/network overheads are minor for monoliths
Threading Effects on Microservice Tails

- Threading can significantly impact microservice SLOs

Prior threading conclusions must be revisited for microservices
Mid-tier Faces More Threading Overheads

- Mid-tier – subject to more threading overheads
  - Manages RPC fan-out to many leaves
  - RPC layer interactions dominate computation

Threading overheads must be characterized for mid-tier microservices
Need for a Microservice Benchmark Suite

Closed-source
[Ayers ‘18]

Monolithic
Architectures
[Ferdman ‘12]

Only one workload
[Hsu ‘15]

Only leaf nodes
[Lo ‘14]

Not representative
[Zhu ‘16]

Domain-specific
[Hauswald ‘15]

No open-source benchmark sufficiently represents microservices
Contributions

μSuite: Benchmark suite of OLDDI services composed of microservices [1]

Taxonomy of threading models: Implications of threading designs [2]

μTune: Load adaptation system to tune threading models & improve tails [2]

Achieve **1.9x** tail latency speedup over state-of-the-art adaptations [2]


Outline

• μSuite: Description of services & microservices
• Show how μSuite facilitates future research

• A taxonomy of threading models
  – Characterize threading effects on microservice tails
• μTune: Dynamic load adaptation system that improves tail latency
• Evaluation
μSuite

HDSearch

Leaf compute bound

Router

Variability in scale-out

Set Algebra

Variability in leaf compute

Recommend

Variability in mid-tier compute

μSuite HDSearch Router Set Alg. Recommend Taxonomy μTune Evaluation
Benchmark 1: HDSearch

- Content-based search for image similarity
- Leaf compute bound - mid-tier has high threading overheads
# HDSearch: Locality Sensitive Hashing

Reduces nearest neighbor computation time

<table>
<thead>
<tr>
<th>Key</th>
<th>Potentially near-by point IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Data set

μSuite  HDSearch  Router  Set Alg.  Recommend  Taxonomy  μTune  Evaluation
HDSearch: Operation

Front-End Microserver

Mid-Tier Microserver

Query

Leaf Microserver 1

Leaf Microserver 2

Point IDs

Point IDs

Query

Query

Key | Leaf id X Point ID
--- | ---
1   | ![Red points](image1)
2   | ![Green points](image2)
3   | ![Blue points](image3)
4   | ![Yellow points](image4)
HDSearch: Operation

Front-End Microserver → Mid-Tier Microserver

Query

Point IDs

Leaf Microserver 1

Leaf 1 data set shard

Leaf Microserver 2

Point IDs

Query

Leaf 2 data set shard
HDSearch: Operation

Front-End Microserver -> Mid-Tier Microserver -> Leaf Microserver 1

Query

Leaf 1’s candidates

Query

Leaf Microserver 2

Leaf 2’s candidates
HDSearch: Operation

1-NN response

Leaf Microserver 1

Leaf Microserver 2

1-NN responses
Other μSuite Services

Benchmark 2: Router
• Fault tolerance by replication
• GET:SET asymmetry
• Varied scale-out per request

Benchmark 3: Set Algebra
• Inverted index of posting lists
• Large variability in leaf compute

Benchmark 4: Recommend
• Collaborative filtering
• Mid-tier does little work
μSuite Can Facilitate Future Research
Contributions

μSuite: Benchmark suite of OLDI services composed of microservices [1]

Taxonomy of threading models: Implications of threading designs [2]

μTune: Load adaptation system to tune threading models & improve tails [2]

Achieve 1.9x tail latency speedup over state-of-the-art adaptations [2]


Threading Designs

- Taxonomy of threading models
- Threading dimensions:
  - Block vs. Poll
  - In-Line vs. Dispatch
  - Synchronous vs. Asynchronous
Threading Dimensions: Block vs. Poll

**Block or Interrupt-Driven**

- Low cost: avoids fruitless poll-loops
- High thread wakeup latency

**Poll**

- Low latency: avoids thread wakeups
- Many poll threads cause contention
Threading Dimensions: In-Line vs. Dispatch

**In-Line**
- Better for short queries: no hand-off
- Many in-line threads may contend

**Dispatch**
- Better network poller locality
- Harder to program: thread-safety
Threading Dimensions: Sync. vs. Async.

**Synchronous**

Front-End → Mid-Tier:
- Request
- Worker notified

Mid-Tier → Leaf:
- Network poller thread
- Task queue
- Synchronous
- Compute
- Worker awaits notification

Leaf → Front-End:
- Response

**Asynchronous**

Front-End → Mid-Tier:
- Request
- Worker notified

Mid-Tier → Leaf:
- Network poller thread
- NW (client) socket
- Resp. thread: <block/poll>
- Asynchronous
- Compute

Leaf → Front-End:
- Response

Synchronous & asynchronous designs are built separately.
Threading Dimensions: Thread Pools

**Synchronous**

- **Front-End**
- **Mid-Tier**
- **Leaf**

1. Network poller thread
2. Task queue
3. Synchronous: Compute
4. Worker awaits notification
5. Response thread: <block/poll>

**Asynchronous**

- **Front-End**
- **Mid-Tier**
- **Leaf**

1. Network poller thread
2. Task queue
3. Asynchronous: Compute
4. Worker

**μSuite** HDSearch Router Set Alg. Recommend Taxonomy μTune Evaluation
A Taxonomy of Threading Models

<table>
<thead>
<tr>
<th></th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>Poll</td>
<td>Block</td>
</tr>
<tr>
<td>In-line</td>
<td>SIP</td>
<td>AIB</td>
</tr>
<tr>
<td>Dispatch</td>
<td>SDP</td>
<td>ADB</td>
</tr>
</tbody>
</table>

Characterize varying thread pool sizes for each functionality
In-line Poll has lowest low-load latency: Avoids thread wakeup delays
Latency Tradeoffs Across Threading Models

In-Line Poll faces contention; Dispatch Poll with one network poller is best
Latency Tradeoffs Across Threading Models

Displacement Block is best at high load as it does not waste CPU
Latency Tradeoffs Across Threading Models

No single threading model works best at all loads
Need for Automatic Load Adaptation: \(\mu\)Tune

- Threading choice can significantly affect tail latency
- Threading latency trade-offs are not obvious
- Most software face latency penalties due to static threading

Opportunity: Exploit trade-offs among threading models at run-time
Contributions

**μSuite**: Benchmark suite of OLDI services composed of microservices [1]

Taxonomy of threading models: Implications of threading designs [2]

**μTune**: Load adaptation s/m to tune threading models & improve tails [2]

Achieve 1.9x tail latency speedup over state-of-the-art adaptations [2]


**μTune**

- Load adaptation: Vary threading model & pool size at run-time
- Abstract threading model boiler-plate code from RPC code

---

**App layer**
- Microservice functionality: ProcessReq(), InvokeLeaf(), FinalizeResp()

**μTune**
- μTune automatic load adaptation system

**Network layer**
- RPC layer

---

**Simple interface: Developer defines only three functions**
μTune: Goals & Challenges

Simple interface

μTune's threading framework

Service code

Quick load change detection

Fast threading model switches

SIP, SDP, SDB, SIB

Scale thread pools

μSuite HDSearch Router Set Alg. Recommend Taxonomy μTune Evaluation
μTune System Design: Auto-Tuner

- Dynamically picks threading model & pool sizes based on load
Experimental Setup

• **μSuite**: Three service tiers:
  – Load generator, a mid-tier, 4 or 16 leaf microservers

• **State-of-the-art load generation mechanisms** [Zhang ‘16]:
  – Closed-loop: Saturation throughput
  – Open-loop (arrivals from exponential distribution): Latency

• **Study μTune’s adaptation in two load scenarios**:
  – Steady-state
  – Transients
Evaluation: μTune’s Load Adaptation

Converges to best threading model & pool sizes to improve tails by up to 1.9x
Conclusion

- μSuite – benchmark suite of microservices
  - μSuite can facilitate future research

- Taxonomy of threading models
  - Optimal threading model is load dependent

- μTune – threading model framework + load adaptation system


μSuite & μTune: Auto-Tuned Threading for OLDI Microservices

Akshitha Sriraman, Thomas F. Wenisch

https://github.com/wenischlab/MicroSuite
https://github.com/wenischlab/MicroTune
BACKUP SLIDES
Instruction Overhead

Sync. μTune’s instruction overhead for steady-state load: <5% mean overhead
Comparison With State-of-the-Art

• Few-to-Many Parallelism:
  – Adapting thread pool sizes
• Langendoen et al.
  – Adapting poll vs. block
• Abdelzaher et al.
  – Time window-based load detection
# Load Transients

<table>
<thead>
<tr>
<th>Service</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>100 QPS</strong></td>
<td><strong>8K QPS</strong></td>
<td><strong>13K QPS</strong></td>
</tr>
<tr>
<td>(0 - 31s)</td>
<td>(30s - 31s)</td>
<td>(31 - 61s)</td>
</tr>
<tr>
<td>SIP</td>
<td>0.99</td>
<td>1.07</td>
</tr>
<tr>
<td>SDB</td>
<td>1.49</td>
<td>1.35</td>
</tr>
<tr>
<td>FM</td>
<td>0.55</td>
<td>1.35</td>
</tr>
<tr>
<td>IPI</td>
<td>1.59</td>
<td>1.10</td>
</tr>
<tr>
<td>TBD</td>
<td>1.03</td>
<td>8.69</td>
</tr>
<tr>
<td>µTune</td>
<td>1.01</td>
<td>1.09</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>100 QPS</strong></td>
<td><strong>8K QPS</strong></td>
<td><strong>13K QPS</strong></td>
</tr>
<tr>
<td>(0 - 31s)</td>
<td>(30s - 31s)</td>
<td>(31 - 61s)</td>
</tr>
<tr>
<td>SIP</td>
<td>1.10</td>
<td>1.07</td>
</tr>
<tr>
<td>SDB</td>
<td>1.31</td>
<td>0.83</td>
</tr>
<tr>
<td>FM</td>
<td>1.33</td>
<td>9.40</td>
</tr>
<tr>
<td>IPI</td>
<td>1.4</td>
<td>1.10</td>
</tr>
<tr>
<td>TBD</td>
<td>1.13</td>
<td>4.51</td>
</tr>
<tr>
<td>µTune</td>
<td>1.12</td>
<td>0.88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>100 QPS</strong></td>
<td><strong>8K QPS</strong></td>
<td><strong>13K QPS</strong></td>
</tr>
<tr>
<td>(0 - 31s)</td>
<td>(30s - 31s)</td>
<td>(31 - 61s)</td>
</tr>
<tr>
<td>SIP</td>
<td>0.95</td>
<td>1.07</td>
</tr>
<tr>
<td>SDB</td>
<td>1.30</td>
<td>0.92</td>
</tr>
<tr>
<td>FM</td>
<td>1.30</td>
<td>12.00</td>
</tr>
<tr>
<td>IPI</td>
<td>1.20</td>
<td>0.94</td>
</tr>
<tr>
<td>TBD</td>
<td>1.00</td>
<td>8.45</td>
</tr>
<tr>
<td>µTune</td>
<td>0.97</td>
<td>0.92</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>100 QPS</strong></td>
<td><strong>8K QPS</strong></td>
<td><strong>13K QPS</strong></td>
</tr>
<tr>
<td>(0 - 31s)</td>
<td>(30s - 31s)</td>
<td>(31 - 61s)</td>
</tr>
<tr>
<td>SIP</td>
<td>1.00</td>
<td>1.07</td>
</tr>
<tr>
<td>SDB</td>
<td>1.26</td>
<td>0.96</td>
</tr>
<tr>
<td>FM</td>
<td>1.23</td>
<td>&gt;1s</td>
</tr>
<tr>
<td>IPI</td>
<td>1.13</td>
<td>1.02</td>
</tr>
<tr>
<td>TBD</td>
<td>1.02</td>
<td>4.96</td>
</tr>
<tr>
<td>µTune</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>
We notice similar trends as on our primary Haswell platform RPC handlers. Complete services incur higher concurrency by quickly moving on to successive requests. Since asynchronous models capitalize on the available thread pool sizes, they are sufficient at various loads, with markedly lower latencies. We note the following differences: trends follow the synchronous models, but latencies are higher. As above, we omit figures for additional tests.

7.1.3 per cent of the tail latency (e.g., 99.5%) is performed by 1.26 but, with lower absolute values (e.g., 1.1). Hardware platform (Intel Xeon “Skylake” vs. “Haswell”).

Set Algebra

- ADP incurs more context switches, cache misses, and contention. However, SDB outperforms SDP, since SDP’s SIP outperforms SDB’s asynchronous model.

- ADP and AIB have similar trends. Broadly, ADP’s SIP outperforms AIB’s asynchronous model. For instance, at low loads, ADP has a 2% higher latency than AIB. We omit figures for additional tests.

- Asynchronous models perform SDB by 1.26 but, with lower absolute values (e.g., 1.1). ADP incurs more context switches, cache misses, and contention. However, SDB outperforms SDP, since SDP’s SIP outperforms SDB’s asynchronous model.

- Load (QPS) for each threading model: AIB, AIP, ADB, ADP.

- Response threads, Workers, Inline/network threads.

- Figure 11: Graph: Latency vs. load for each threading model.
We explore microservice threading models by first characterizing our threading models. We then compare synchronous vs. asynchronous performance. We measure thread wakeup delays (reported as latency histograms) using the BPF run queue (scheduler) latency tool [23].

The synchronous vs. asynchronous trade-off is one of programmability vs. performance. It would be unusual for a development team to construct both microservice synchronous and asynchronous models to report how the latency-up to 15% percent tail latency ratio (mean of 12% at sync.-achievable loads & infinitely faster at high loads.

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

We study the tail latency vs. load trade-off for services built with state-of-the-art adaptation systems. We show a cross-tier microservice. We use Intel's HITM (hit-Modified) cache misses and context switches incurred by the mid-tier microservice. We use Linux's Linux kernel version 3.19.0 tasksets; an increase in HITM events indicates a correspondence PEBS coherence event to detect true sharing of cache lines.

To test the effectiveness of asynchronous models, a 42% mean throughput increase in lock contention [109]. We measure thread context switches incurred by the mid-tier microservice. We use Intel's HITM (hit-Modified) cache misses and context switches incurred by the mid-tier microservice.

We record saturation throughput. To state-of-the-art adaptation systems.

We in-build, debug, and tune the asynchronous models. But, we spent 5% more effort to tune asynchronous models, a 42% mean throughput boost across all services. But, we spent 5% more effort to tune synchronous models. We show a cross-service tail latency gap arises because asynchronous models prevent long queuing delays. So, we compare tail latencies later. We find asynchronous models improve tail latencies at load levels from 64 QPS up to synchronous saturation, as the offered load is unsustainable. In Fig. 7, we show the best sync-to-async ratio of tail latency across all threading models and load based on an exhaustive thread pool size search. Points above the dashed line represent unbounded tail latencies.

In Fig. 8, each data point is the best tail latency across all threading models and load. Latency cannot meaningfully be measured at async.-achievable loads & infinitely faster at high loads.

Figure 7: Best sync:async tail latency ratio: async. is faster by a mean 42%.

Table 1: Mid-tier microservice hardware specification.

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Clock frequency (GHz)</th>
<th>Core count</th>
<th>Memory (GB)</th>
<th>Network</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDSearch</td>
<td>2.30</td>
<td>2</td>
<td>500</td>
<td>10Gbit/s</td>
<td>Intel Xeon E5-2699 v3 “Haswell”</td>
</tr>
<tr>
<td>Router</td>
<td>2.30</td>
<td>2</td>
<td>500</td>
<td>10Gbit/s</td>
<td>Intel Xeon E5-2699 v3 “Haswell”</td>
</tr>
<tr>
<td>Set Algebra</td>
<td>2.30</td>
<td>2</td>
<td>500</td>
<td>10Gbit/s</td>
<td>Intel Xeon E5-2699 v3 “Haswell”</td>
</tr>
<tr>
<td>Recommend</td>
<td>2.30</td>
<td>2</td>
<td>500</td>
<td>10Gbit/s</td>
<td>Intel Xeon E5-2699 v3 “Haswell”</td>
</tr>
</tbody>
</table>
Sync. Vs. Async.: Tail Latency

![Chart showing 99th percentile tail latency ratio (sync:async) vs. load (Queries Per Second).]

- **HDSearch**
- **Router**
- **Set Algebra**
- **Recommend**

The chart displays the 99th percentile tail latency ratio between synchronous and asynchronous models across various loads. The y-axis represents the ratio, while the x-axis shows the load in queries per second (QPS).

Key observations:
- Asynchronous models generally have a lower 99th percentile tail latency ratio compared to synchronous models, indicating faster response times.
- The ratio varies with load, with asynchronous models showing a more consistent performance across different loads compared to synchronous models.
- The chart highlights the benefit of asynchronous models in reducing tail latency, especially at higher loads.
Thread Wakeup Delays
OS & Microarchitectural Effects

[Bar chart showing normalized increase over best model for HITMs, Context switch, and Cache miss for SIB, SIP, SDB, and SDP.]
We show results for services (e.g., 200K QPS Memcached [26]) as slightest concurrency by quickly moving on to successive requests. Since asynchronous models capitalize on the available threads, thread pool sizes are sufficient at various loads, markedly lower. We note the following differences: trends follow the synchronous models, but latencies are services as they match null RPCs, SIP outperforms SDB by 1.42×, which is a significant improvement. For null RPCs, SIP outperforms SDB by 1.57×, which is even more pronounced.

We find four threads enough to sustain high loads. Larger pool sizes that achieve the best tails for each load level. However, SDB outperforms SDP, since SDP tends for CPU (in contrast to SDP at high loads). This is because ADP incurs more context switches, cache misses, and work sockets or CPU resources. In contrast, SIB, SDB, and cache misses. SIB in-line threads contend less as they block, rather than poll. SDB and SDP exhibit similar latency for that load—No threading model is always the best.

Asynchronous models outperform set algebra. ADB and ADP both show comparable performance, with ADP outperforming ADB by 1.1×. However, AIB and AIP show significant improvements, with AIP outperforming SIB by 1.4× and AIB outperforming SIB by 1.2×.

Comparison to the state-of-the-art adaptation techniques [43, 76, 97]. We find that ADB scales much better than SIP. AIP scales much better than SIP. ADP scales worse than SDP. AIB scales much better than SIP. ADP incurs more context switches, cache misses, and work sockets or CPU resources. In contrast, SIB, SDB, and cache misses. SIB in-line threads contend less as they block, rather than poll. SDB and SDP exhibit similar latency for that load—No threading model is always the best.

We next compare ADB, ADP, AIP, and AIB. We find that ADB scales much better than SIP. AIP scales much better than SIP. ADP scales worse than SDP. AIB scales much better than SIP. ADP incurs more context switches, cache misses, and work sockets or CPU resources. In contrast, SIB, SDB, and cache misses. SIB in-line threads contend less as they block, rather than poll. SDB and SDP exhibit similar latency for that load—No threading model is always the best.

We compare these tests as they match the reported trends. We notice similar trends as on our primary Haswell platform (Intel Xeon “Skylake” vs. “Haswell”).

Figure 13: Async. thread pools for best tails: Big pools contend. Figure 12: Async. thread pools for best tails: Big pools contend.

Normalized increase over ADB

<table>
<thead>
<tr>
<th>HITMs</th>
<th>Context switch</th>
<th>Cache miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIB</td>
<td>1.1×</td>
<td>1.2×</td>
</tr>
<tr>
<td>AIP</td>
<td>1.4×</td>
<td>1.3×</td>
</tr>
<tr>
<td>ADB</td>
<td>1.5×</td>
<td>1.4×</td>
</tr>
<tr>
<td>ADP</td>
<td>1.3×</td>
<td>1.2×</td>
</tr>
</tbody>
</table>
Router

- Routes key-value stores to Memcached
- Replication-based protocol routing for fault-tolerance
  - SETs go to multiple leaves
  - GETs go to a single leaf
- More scalable – a subset of leaves are contacted
  - May face more threading overheads due to GET/SET asymmetry
Router: Operation

Front-End Microserver

Mid-Tier Microserver

Leaf Microserver 1

Leaf Microserver 2

SpookyHash

SET query: Name = Tom

Key
Value

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Tom</td>
</tr>
</tbody>
</table>

Memcached

μSuite HDSearch Router Set Alg. Recommend Taxonomy μTune Evaluation
Making Router a Benchmark

• Query set:
  – Set of {key, value} pairs from a Twitter data set [Ferdman ‘12]
  – GET:SET distributions mimic YCSB’s workload A (50:50)
Set Algebra

• Document retrieval for web search
  – Set intersections on posting lists

• Inverted index:
  – Map of term to all doc IDs containing term

• Large variability in leaves’ compute
  – Helps study overheads with short & long requests

<table>
<thead>
<tr>
<th>ID</th>
<th>Term</th>
<th>Doc. IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>3</td>
<td>Butterfly</td>
<td>1, 2, 6, 7</td>
</tr>
<tr>
<td>3</td>
<td>Rainbow</td>
<td>2, 4, 5</td>
</tr>
<tr>
<td>4</td>
<td>Unicorn</td>
<td>2</td>
</tr>
</tbody>
</table>
Set Algebra: Operation

Search query: “rainbow unicorn”

Set union

Front-End Microserver

Mid-Tier Microserver

Leaf Microserver 1

Leaf Microserver 2

Inverted index

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Butterfly</td>
<td>1 3</td>
</tr>
<tr>
<td>Rainbow</td>
<td>3</td>
</tr>
<tr>
<td>Unicorn</td>
<td>1 3</td>
</tr>
</tbody>
</table>

Inverted index

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Butterfly</td>
<td>2 8</td>
</tr>
<tr>
<td>Unicorn</td>
<td>4</td>
</tr>
<tr>
<td>Rainbow</td>
<td>4 6</td>
</tr>
</tbody>
</table>

μSuite HDSearch Router Set Alg. Recommend Taxonomy μTune Evaluation
Making Set Algebra a Benchmark

- Data set: inverted index of documents
  - 4.3M documents from Wikipedia: 10 GB
  - Prepared sharded inverted index corpus
  - Test set: Synthetically created using Wikipedia’s word probabilities
  - Query: uniformly randomly selected set of <= 10 terms
Recommend

• Predicts user ratings for specific items
  – Uses collaborative filtering

• Mid-Tier does minimal work on the request path
  – Helps study unmasked OS and network effects
Recommend: Operation

Front-End Microserver

Search query: “User: Tom; Item: The Hobbit”

Mid-Tier Microserver

Leaf Microserver 1

Leaf Microserver 2

Average

Collaborative filtering

Collaborative filtering

μSuite | HDSearch | Router | Set Alg. | Recommend | Taxonomy | μTune | Evaluation
---|---|---|---|---|---|---|---
57
Making Recommend a Benchmark

• Dataset: \{user, item, rating\} tuples
  – MovieLens movie recommendation data set [Harper ‘15]
  – Prepared sharded sparse user-item rating matrix
  – Test set of \{user, item\} query pairs from MovieLens [Harper ‘15]
Characterizing the Threading Taxonomy

- SIP has lowest latency at low load
  - Avoid two kinds of thread wakeups
- SDP is best at intermediate loads
  - Avoids in-line polling thread contention
- SDB enables highest load
  - Single network thread, many workers

No single threading model is optimal at all loads
Comparison With State-of-the-Art Adaptation

- Few-to-Many (FM) parallelism [Haque ‘15]
  - Uses offline interval table to select thread pool sizes
- Integrating Polling and Interrupts (IPI) [Langendoen ‘96]
  - Polls when threads are blocked
  - Uses interrupts when blocked thread returns
- Time-window Based Detection (TBD) [Abdelzaher ‘99]
  - Track request arrivals in fixed observation time windows

μTune should outperform as it considers both threading models & pool sizes
Async. models are more performant although harder to program