**Autothrottle:**

A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices

Zibo Wang\textsuperscript{12}, Pinghe Li\textsuperscript{3}, Chieh-Jan Mike Liang\textsuperscript{1}, Feng Wu\textsuperscript{2}, Francis Y. Yan\textsuperscript{1}

\textsuperscript{1} Microsoft Research
\textsuperscript{2} University of Science and Technology of China
\textsuperscript{3} ETH Zurich
Cloud applications are shifting toward microservices
What microservice applications look like
A client request traverses many services
Different requests have different trajectories
Inadequate CPU allocations => high application latency

end-to-end latency: 800ms
SLO: 200ms

SLO violation
Excessive CPU allocations => waste of resources

end-to-end latency: 50ms
SLO: 200ms

waste CPU
Minimizing CPU allocation while meeting SLO

- Search space grows exponentially with number of services
- Mapping from CPU allocations to latency is unclear

end-to-end latency: 190ms
SLO: 200ms
Existing approach: service-level allocation

- Example: Kubernetes’ default heuristics

<table>
<thead>
<tr>
<th>Parameter: target CPU utilization = 50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: CPU usage = 2 cores</td>
</tr>
<tr>
<td>Output: CPU allocation = 2 / 50% = 4 cores</td>
</tr>
</tbody>
</table>

✔ Low overhead
✔ Fast reaction
✖ No global visibility
Existing approach: application-level allocation

- Example: Sinan (ASPLOS ’21)

ML-based allocation server

SLO

RPS, latency

Client → Nginx

CPU usage

Media

User

Unique Id

Url Shorten

Text

User Mention

Compose Post

Media Filter

Home Timeline

User Timeline

Text Filter

Social Graph

Compose Post

Redis

Home Timeline

Post Storage

MongoDB

Redis

Post Storage

Memcached

Social Graph

MongoDB

User Timeline

User Mention

Compress Post

Write Home Timeline

ML-based allocation server

✔ Global visibility

✖ Less responsive

✖ High (re-)training overhead
How to obtain the best of both worlds?

<table>
<thead>
<tr>
<th></th>
<th>Service-level</th>
<th>Application-level</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low overhead</td>
<td>✔️</td>
<td>✖️</td>
<td>✔️</td>
</tr>
<tr>
<td>Fast reaction</td>
<td>✔️</td>
<td>✖️</td>
<td>✔️</td>
</tr>
<tr>
<td>Global visibility</td>
<td>✖️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Our bi-level approach to resource management
Our bi-level approach to resource management

Use locally available metrics to perform CPU allocation

✔ Low overhead
✔ Fast reaction
Our bi-level approach to resource management

Monitor RPS, end-to-end latencies, and SLO violations
✔ Global visibility
Our bi-level approach to resource management

Performance target
- Periodically determined by the global controller
- Enabling local controllers to remain autonomous
Implementing bi-level approach with **Autothrottle**

**Description:***

- **SLO** flows from the Client to the Tower, which then sends it to the CPU allocation throttle ratio.
- The CPU allocation throttle ratio is then sent back to the Tower.
- The Tower sends the CPU allocation to the Captain, which allocates the CPU usage and throttle.
- The Captain sends the allocation and throttle back to the Tower.
- The Tower sends the allocation and throttle to the CPU scheduler.
- The CPU scheduler sends the allocation and throttle to the Media, User, and Unique Id modules.
- The User Timeline and Home Timeline modules receive the allocation and throttle from the CPU scheduler.

**Key Components:**
- **Tower**
- **CPU allocation throttle ratio**
- **Captain**
- **CPU scheduler**
- **Client**
- **Nginx**
- **Media**
- **User**
- **Unique Id**
- **Url Shorten**
- **Text**
- **User Mention**
- **Home Timeline**
- **User Timeline**
- **Write User Timeline**
- **Compose Post**
- **Text Filter**
- **Media Filter**

**Technologies:**
- MongoDB
- Memcached
- Redis

**Throttle Ratio Components:**
- **RPS**, **latency**, **SLO**, **CPU allocation**, **throttle ratio**

**Colors:**
- Blue
- Green
- Yellow
- Orange
- Red
- Purple
- Brown

**Notes:**
- Implementing bi-level approach with Autothrottle for optimized CPU allocation and usage.
Interface: throttle ratio
Interface: throttle ratio

- Example: Linux CFS (Completely Fair Scheduler)

![Diagram showing throttle ratio and periods]

**throttle ratio = \( \frac{1\ \text{period}}{3\ \text{periods}} \)**
Throttle ratio has a higher correlation with latency
Service-level: fast and lightweight Captains
Service-level: fast and lightweight Captains

- Closed-loop control based on throttle ratio target
- Collect data every 100ms, adjust allocation every 1s
Application-level: online learning Tower

Client → Nginx

SLO

RPS, latency

Tower

CPU allocation
throttle ratio

CPU scheduler

Captain

allocation

throttle, usage

Media

User

Unique Id

Url Shorten

Text

User Mention

Compose Post

Media Filter

Compose Pos

User Timeline

Home Timeline

Write User Timeline

Social Graph

MongoDB

Home Timeline

Redis

Post Storage

MongoDB

User Timeline

Redis

User Memcached

User MongoDB

User Timeline

Redis

User Memcached

User MongoDB

User Mention

Text Filter
Application-level: online learning Tower

- Determine the best throttle targets for Captains to achieve
- Lightweight online learning: contextual bandit algorithm
  - One step per minute, each step runs in ~100ms

Cost: computed with CPU allocation, end-to-end latency, and SLO
Evaluation methodology

- **Testbed:** 5 Azure VMs, 160 CPU cores in total
- **4 workload traces**
  - with patterns commonly observed in production environments
  - e.g. Puffer’s streaming requests, Google’s cluster usage, and Twitter tweets
- **3 benchmark applications**
  - Train-Ticket
  - Hotel-Reservation from DeathStarBench
  - **Social-Network** used in Sinan
Less allocation is better

Evaluation results

Kubernetes’ algorithm wastes up to 35% CPU
- It uses the same CPU utilization for all services

Sinan wastes even more CPU
- Its search space is too large to explore
Large-scale evaluation on a 512-core cluster

Kubernetes’ algorithm wastes up to 39% CPU

Less allocation is better
A 21-day comparison
A 21-day comparison

Autothrottle: automatic exploration
K8s-CPU: manual parameter tuning
A 21-day comparison

SLO = 200ms
A 21-day comparison

K8s-CPU violated SLO in 71 of 480 hours
A 21-day comparison

K8s-CPU violated SLO in 71 of 480 hours
A 21-day comparison

Red boxes mark K8s-CPU's SLO violations

K8s-CPU violated SLO in 71 of 480 hours
A 21-day comparison

Autothrottle saves an average of 12 cores and up to 35 cores

K8s-CPU violated SLO in 71 of 480 hours
A 21-day comparison

- Tower: global visibility
- Throttle ratio: higher correlation with latency
- Captains: low overhead and fast reaction
Conclusion

- Results show a CPU saving up to 26% while satisfying SLO
- Open-sourced at https://github.com/microsoft/autothrottle