Tail Latency in Node.js: Energy Efficient Turbo Boosting for Long Tail Requests in JavaScript

Wenzhi Cui
Daniel Richins
Yuhao Zhu
Vijay Janapa Reddi
Connecting People (2010s)
Connecting People (2010s)

Cloud Computing

1 billion Gmail users

>1 billion users
Connecting **Things** (2020s)
Connecting **Things** (2020s)

50 Billion Devices

“The Internet of Things” — Cisco
www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf
Thread-based Programming: Traditional Approach

Client Requests
Thread-based Programming: 
**Traditional Approach**

- Client Requests
- Blocking I/O
- Client Response
Thread-based Programming: 
**Traditional Approach**

Client Requests

Client Response

Blocking I/O

CPU
Thread-based Programming: Traditional Approach

Client Requests → Limited resources & thrashing

Client Response ← Blocking I/O

Limited resources & thrashing
Thread-based Programming

[Welsh et al. ’00]
Event-driven Programming: Emerging Approach
Event-driven Programming: \textbf{Emerging} Approach

<table>
<thead>
<tr>
<th>Head</th>
<th>Event Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tail</td>
<td></td>
</tr>
</tbody>
</table>
Event-driven Programming: Emerging Approach
Event-driven Programming: **Emerging** Approach

```javascript
fs.readFile('input.txt',
  function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
  }
);

console.log("Program continues…");
```
Event-driven Programming: 
Emerging Approach

Application Tasks

Head
Tail

Single-threaded event loop

```
fs.readFile('input.txt',
  function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
  }
);

console.log("Program continues...");
```
Event-driven Programming: Emerging Approach

Single-threaded event loop

Application Tasks

Head
Tail

Example code:
```javascript
fs.readFile('input.txt',
  function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
  }
);

console.log("Program continues…");
```
Event-driven Programming: Emerging Approach

Asynchronous I/O
- DB Access
- File I/O
- Network

Single-threaded event loop

Application Tasks
- Head
- Tail

Event-driven Programming

```javascript
fs.readFile('input.txt',
  function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
  }
);

console.log("Program continues...");
```
Event-driven Programming: Emerging Approach

Single-threaded event loop

Asynchronous I/O
- DB Access
- File I/O
- Network

Application Tasks

Event-driven Programming:

```javascript
fs.readFile('input.txt',
          function (err, data) {
              if (err) return console.error(err);
              console.log(data.toString());
          });

console.log("Program continues…");
```
Event-driven Programming: 
Emerging Approach

Single-threaded event loop

fs.readFile('input.txt',
function (err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});

console.log("Program continues...");
Event-driven Programming: Emerging Approach

Single-threaded event loop

Asynchronous I/O
DB Access
File I/O
Network

Application Tasks

Event Queue

fs.readFile('input.txt',
  function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
  }
);

console.log("Program continues...");
Thread-based Programming

Event-driven Programming

Graphs showing performance metrics for both thread-based and event-driven programming paradigms.

[Welsh et al. ’00]
The Changing Programming Language Landscape
The Changing Programming Language Landscape

<table>
<thead>
<tr>
<th>Repository Language</th>
<th>Active Repositories</th>
<th>Total Pushes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ruby</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shell</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VimL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Go</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perl</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## The Changing Programming Language Landscape

<table>
<thead>
<tr>
<th>Repository Language</th>
<th>Active Repositories</th>
<th>Total Pushes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ruby</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shell</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VimL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Go</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perl</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Event-driven Execution Model

Managed Language
Taming Tail Latencies in Event-Driven Web Services

Fraction of Requests vs. Request Latency

Tail
Experimental Setup

Wrk2: A customized Load Testing Tool, simulate real-world workloads

Intel i7-4790K, 4 physical cores with hyper-threading, 32 GB DRAM, 240GB SSD

(1 Gbps Network)
## Applications

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>I/O Type</th>
<th>#Requests</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etherpad lite</td>
<td>N/A</td>
<td>20K</td>
<td>Real time word processor.</td>
</tr>
<tr>
<td>Todo</td>
<td>Redis</td>
<td>40K</td>
<td>Online Task Manager.</td>
</tr>
<tr>
<td>Lighter</td>
<td>Disk</td>
<td>40K</td>
<td>Blogging Engine.</td>
</tr>
<tr>
<td>Let’s Chat</td>
<td>MongoDB</td>
<td>10K</td>
<td>Web-based Chat Application.</td>
</tr>
<tr>
<td>Client Manager</td>
<td>MongoDB</td>
<td>40K</td>
<td>Online Address book.</td>
</tr>
</tbody>
</table>

Github Repo: https://github.com/nodebenchmark/benchmarks
Etherpad: An Example
Etherpad: An Example

CDF

(1.85, 50%)
Etherpad: An Example

CDF

Latency (ms)

(1.85, 50%)

(7.30, 90%)
Etherpad: An Example

CDF

Latency (ms)

(1.85, 50%)

(7.30, 90%)

(36.91, 99.9%)
Etherpad: An Example

![CDF Graph with Latency (ms) and Tail Region](image)

- (1.85, 50%)
- (7.30, 90%)
- (36.91, 99.9%)
Tail latency (99.9%) is 9.1x longer than median request latency
## System Overview

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
</tr>
</thead>
</table>

System Overview

Step 1

Tools to Root-cause Tail in Node.js

Step 2

Step 3
System Overview

Step 1

- Req1
- Req2
- Event-driven runtime (libuv)
- File/Net JS libraries
- JavaScript runtime (V8)
- JS to C++ bindings

Tail Latency Reconstruction

Step 2

- Event-Dependency Graph (EDG)

Step 3

- Event Critical Path
- Event Data
System Overview

**Step 1**
- **Tail Latency Reconstruction**
- **File/Net JS libraries**
- **JS to C++ bindings**
- **Event-driven runtime (libuv)**
- **JavaScript runtime (V8)**

**Step 2**
- **Event Dependency Graph (EDG)**
- **Critical Path**

**Step 3**
- **Root-causing Tail in Node.js**
System Overview

Step 1

Tail Latency Reconstruction

Event-Driven Graph (EDG)

Event Critical Path

Step 2

Tail Latency Bottleneck Analysis

Step 3

Req1  Req2  ...
File/Net JS libraries
JS to C++ bindings
Event-driven runtime (libuv)
JavaScript runtime (V8)

IO (%)
Queue (%)
Exec (%)
GC (%)
JIT (%)
...
System Overview

Step 1

Tail Latency Reconstruction

- Req1
- Req2
- File/Net JS libraries
- JS to C++ bindings
- Event-driven runtime (libuv)
- JavaScript runtime (V8)

Event-Dependency Graph (EDG)

- e1
- e2
- e3
- e4
- e5

Event Critical Path

Step 2

Tail Latency Bottleneck Analysis

- I/O
- Queue
- Exec

Request Latency

- Req1
- Req2

IO (%)
Queue (%)
Exec (%)
GC (%)
JIT (%)

...
System Overview

Step 1

Tail Latency Reconstruction

- Req1
- Req2
- ... File/Net JS libraries
- JS to C++ bindings
- Event-driven runtime (libuv)
- JavaScript runtime (V8)

Event-Dependency Graph (EDG)

- e1
- e2
- e3
- e4
- e5

Event Critical Path

Event Data

Step 2

Tail Latency Bottleneck Analysis

- IO (%)
- Queue (%)
- Exec (%)
- GC (%)
- JIT (%)

Step 3

Tail Latency Optimization

Queue Boosting

Event Queue

VM Optimization

VM Tuning
System Overview

**Step 1**

Tail Latency Reconstruction

- Event-Dependency Graph (EDG)
- Critical Path

**Step 2**

Tail Latency Bottleneck Analysis

- Event-Queue
- Request Latency
- IO (%), Queue (%), Exec (%), GC (%), JIT (%)

**Step 3**

Tail Latency Optimization

- Queue Boosting
- Event Queue
- VM Optimization
- VM Tuning

**Static Instrumentation**

- Req1
- Req2
- 
- File/Net JS libraries
- JS to C++ bindings
- Event-driven runtime (libuv)
- JavaScript runtime (V8)

---

21
System Overview

Step 1

Tail Latency Reconstruction

Event-Dependency Graph (EDG)

Req1  Req2  ...
File/Net JS libraries
JS to C++ bindings
Event-driven runtime (libuv)
JavaScript runtime (V8)

Step 2

Tail Latency Bottleneck Analysis

Event-Dependency Graph (EDG)

Event Data

I/O
I/O
Exec
Exec
Queue
Queue

Req1  Req2

IO (%) Queue (%) Exec (%) GC (%) JIT (%) ...

Step 3

Tail Latency Optimization

I/O Boosting
Event Queue
VM Optimization
VM Tuning

Static Instrumentation

Dynamic Analysis & Optimization
Step 1: Latency Reconstruction

```javascript
var count = N;
fs.readdir("/data", function dir(err, files) {

});
```
Step 1: Latency Reconstruction

```javascript
var count = N;
fs.readdir("/data", function dir(err, files) {
    files.foreach(function file(f, index) {
        var fname = …;
        fs.readFile(fname, function read(err, data) {
            count -= 1;
            if (count == 0)
                sendResponse();
        });
    });
});
```
Step 1: Latency Reconstruction

```javascript
var count = N;
fs.readdir("/data", function dir(err, files) {
  files.foreach(function file(f, index) {
    var fname = ...;
    fs.readFile(fname, function read(err, data) {
      count -= 1;
      if (count == 0)
        sendResponse();
    });
  });
});
```
Step 1: Latency Reconstruction

```javascript
var count = N;
fs.readdir("/data", function dir(err, files) {
    files.forEach(function file(f, index) {
        var fname = ...;
        fs.readFile(fname, function read(err, data) {
            count -= 1;
            if (count == 0)
                sendResponse();
        });
    });
});
```
Step 1: Latency Reconstruction

```javascript
var count = N;
fs.readdir("/data", function dir(err, files) {
  files.forEach(function file(f, index) {
    var fname = ...
    fs.readFile(fname, function read(err, data) {
      count -= 1;
      if (count == 0)
        sendResponse();
    });
  });
});
```
Event Dependency Graph (EDG)
Event Dependency Graph (EDG)

![Event Dependency Graph (EDG)](image)

- Event: e1
- Event: e2
- Event: e3
- Event: e4
- Event: e5

Critical Path
Event Dependency Graph (EDG)

$$\text{Event-Dependency Graph (EDG)}$$

$$\text{Critical Path}$$

$$\text{Event}$$

$$\text{Latency}(R) = \sum_{i=1}^{N-1} T(e_i), e_i \in ECP(R)$$
How do we obtain the latency of each event?

\[
Latency(R) = \sum_{i=1}^{N-1} T(e_i), e_i \in ECP(R)
\]
Deconstructing Event Latency

Sever-side Latency
Deconstructing Event Latency

Server-side Latency

- I/O
- Queue.
- Exec.
Deconstructing Event Latency

Server-side Latency

- I/O
- Queue.
- Exec.

Compute Time
Deconstructing Event Latency

![Diagram showing components of server-side latency: I/O, Queue, Exec, Compute Time, GC Interrupt, IC Miss, JIT Engine, User Code]
Offline Instrumentation + Online Analysis

Instrument the Node.js runtime so that at runtime we could easily obtain: EDG & event latency info.

- Req1
- Req2
- ... File/Net JS libraries
- JS to C++ bindings
- Event-driven runtime (libuv)
- JavaScript runtime (V8)
Offline Instrumentation + Online Analysis

Instrument the Node.js runtime so that at runtime we could easily obtain: EDG & event latency info.

Identify root-causes of long tails at runtime.

Instrument the Node.js runtime so that at runtime we could easily obtain: EDG & event latency info.

Identify root-causes of long tails at runtime.
Step 2: EDG-based Bottleneck Analysis
**Step 2: EDG-based Bottleneck Analysis**

![Diagram showing latency analysis with categories: IO, Queue, Exec, and request types An, Bn, Cn, Dn, Avgn, At, Bt, Ct, Dt, Avgt.]
Step 2: EDG-based Bottleneck Analysis
Step 2: EDG-based Bottleneck Analysis

Etherpad: Queue and Exec. latency increases in tails, and I/O latency is not dominant for this particular application.
EDG-based Bottleneck Analysis

Request Type

Latency (ms)

Non-tail

Tail

IO
Queue
Exec

\( V_n \)
\( W_n \)
\( X_n \)
\( \text{Avg}_n \)
\( V_t \)
\( W_t \)
\( X_t \)
\( \text{Avg}_t \)
EDG-based Bottleneck Analysis

Client Manager: Queue and Exec. latency dominate in tails, but unlike Etherpad I/O plays a notable role in the requests.
EDG-based Bottleneck Analysis

On average, **queuing** and **native code execution** time contribute to ~80% of the tail latencies.

**Client Manager:** Queue and Exec. latency dominate in tails, but unlike Etherpad I/O plays a notable role in the requests.
Breakdown Within Compute
## Breakdown Within Compute

### Non-tail

<table>
<thead>
<tr>
<th>Application</th>
<th>IC</th>
<th>GC</th>
<th>JIT</th>
<th>Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etherpad Lite</td>
<td>0.3%</td>
<td>1.9%</td>
<td>0%</td>
<td>97.8%</td>
</tr>
<tr>
<td>Todo</td>
<td>1.8%</td>
<td>0.5%</td>
<td>0%</td>
<td>97.7%</td>
</tr>
<tr>
<td>Lighter</td>
<td>4.9%</td>
<td>4.7%</td>
<td>0%</td>
<td>90.4%</td>
</tr>
<tr>
<td>Let’s Chat</td>
<td>2.7%</td>
<td>3.9%</td>
<td>0%</td>
<td>93.4%</td>
</tr>
<tr>
<td>Client Manager</td>
<td>3.2%</td>
<td>3.1%</td>
<td>0%</td>
<td>93.7%</td>
</tr>
</tbody>
</table>
## Breakdown Within Compute

<table>
<thead>
<tr>
<th>Application</th>
<th>IC</th>
<th>GC</th>
<th>JIT</th>
<th>Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etherpad Lite</td>
<td>0.3%</td>
<td>1.9%</td>
<td>0%</td>
<td>97.8%</td>
</tr>
<tr>
<td>Todo</td>
<td>1.8%</td>
<td>0.5%</td>
<td>0%</td>
<td>97.7%</td>
</tr>
<tr>
<td>Lighter</td>
<td>4.9%</td>
<td>4.7%</td>
<td>0%</td>
<td>90.4%</td>
</tr>
<tr>
<td>Let’s Chat</td>
<td>2.7%</td>
<td>3.9%</td>
<td>0%</td>
<td>93.4%</td>
</tr>
<tr>
<td>Client Manager</td>
<td>3.2%</td>
<td>3.1%</td>
<td>0%</td>
<td>93.7%</td>
</tr>
</tbody>
</table>

Diagram showing Compute Time breakdown:

- Server-side Latency
- I/O
- Queue
- Exec
  - GC Interrupt
  - IC Miss
  - JIT Engine
  - User Code

28
### Breakdown Within Compute

#### Application Breakdown

<table>
<thead>
<tr>
<th>Application</th>
<th>Non-tail</th>
<th>Tail</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IC</td>
<td>GC</td>
<td>JIT</td>
</tr>
<tr>
<td><strong>Etherpad Lite</strong></td>
<td>0.3%</td>
<td>1.9%</td>
<td>0%</td>
</tr>
<tr>
<td><strong>Todo</strong></td>
<td>1.8%</td>
<td>0.5%</td>
<td>0%</td>
</tr>
<tr>
<td><strong>Lighter</strong></td>
<td>4.9%</td>
<td>4.7%</td>
<td>0%</td>
</tr>
<tr>
<td><strong>Let’s Chat</strong></td>
<td>2.7%</td>
<td>3.9%</td>
<td>0%</td>
</tr>
<tr>
<td><strong>Client Manager</strong></td>
<td>3.2%</td>
<td>3.1%</td>
<td>0%</td>
</tr>
</tbody>
</table>

#### Diagram:

- **Compute Time**
  - **Sever-side Latency**
    - **I/O**
    - **Queue**
    - **Exec**
      - **GC Interrupt**
      - **IC Miss**
      - **JIT Engine**
      - **User Code**
Breakdown Within Compute

<table>
<thead>
<tr>
<th>Application</th>
<th>IC</th>
<th>GC</th>
<th>JIT</th>
<th>Non-tail</th>
<th>IC</th>
<th>GC</th>
<th>JIT</th>
<th>Tail</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Etherpad Lite</em></td>
<td>0.3%</td>
<td>1.9%</td>
<td>0%</td>
<td>97.8%</td>
<td>7.0%</td>
<td>41.9%</td>
<td>0%</td>
<td>51.1%</td>
</tr>
<tr>
<td><em>Todo</em></td>
<td>1.8%</td>
<td>0.5%</td>
<td>0%</td>
<td>97.7%</td>
<td>0.6%</td>
<td>31.0%</td>
<td>0%</td>
<td>68.4%</td>
</tr>
<tr>
<td><em>Lighter</em></td>
<td>4.9%</td>
<td>4.7%</td>
<td>0%</td>
<td>90.4%</td>
<td>4.4%</td>
<td>45.3%</td>
<td>0%</td>
<td>50.3%</td>
</tr>
<tr>
<td><em>Let’s Chat</em></td>
<td>2.7%</td>
<td>3.9%</td>
<td>0%</td>
<td>93.4%</td>
<td>3.6%</td>
<td>48.4%</td>
<td>0%</td>
<td>48.0%</td>
</tr>
<tr>
<td><em>Client Manager</em></td>
<td>3.2%</td>
<td>3.1%</td>
<td>0%</td>
<td>93.7%</td>
<td>3.9%</td>
<td>70.3%</td>
<td>0%</td>
<td>25.8%</td>
</tr>
</tbody>
</table>
## Breakdown Within Compute

### Compute Time Components

- **Sever-side Latency**
  - I/O
  - Queue
  - Exec

- **GC Interrupt**
- **IC Miss**
- **JIT Engine**
- **User Code**

### Breakdown Table

<table>
<thead>
<tr>
<th>Application</th>
<th>Non-tail</th>
<th></th>
<th>Tail</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IC</td>
<td>GC</td>
<td>JIT</td>
<td>Native</td>
<td>IC</td>
<td>GC</td>
<td>JIT</td>
<td>Native</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Etherpad Lite</em></td>
<td>0.3%</td>
<td>1.9%</td>
<td>0%</td>
<td>97.8%</td>
<td>7.0%</td>
<td>41.9%</td>
<td>0%</td>
<td>51.1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Todo</em></td>
<td>1.8%</td>
<td>0.5%</td>
<td>0%</td>
<td>97.7%</td>
<td>0.6%</td>
<td>31.0%</td>
<td>0%</td>
<td>68.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Lighter</em></td>
<td>4.9%</td>
<td>4.7%</td>
<td>0%</td>
<td>90.4%</td>
<td>4.4%</td>
<td>45.3%</td>
<td>0%</td>
<td>50.3%</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Let’s Chat</em></td>
<td>2.7%</td>
<td>3.9%</td>
<td>0%</td>
<td>93.4%</td>
<td>3.6%</td>
<td>48.4%</td>
<td>0%</td>
<td>48.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Client Manager</em></td>
<td>3.2%</td>
<td>3.1%</td>
<td>0%</td>
<td>93.7%</td>
<td>3.9%</td>
<td>70.3%</td>
<td>0%</td>
<td>25.8%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Breakdown Within Compute

<table>
<thead>
<tr>
<th>Engine</th>
<th>Application</th>
<th>Non-tail</th>
<th>Tail</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IC</td>
<td>GC</td>
<td>JIT</td>
</tr>
<tr>
<td>Let’s Chat</td>
<td>2.7%</td>
<td>3.9%</td>
<td>0%</td>
</tr>
<tr>
<td>Client Manager</td>
<td>3.2%</td>
<td>3.1%</td>
<td>0%</td>
</tr>
</tbody>
</table>

We should focus optimization efforts on **Garbage Collection** and **Generated Native Code**
Step 3: Tail Latency Optimization

- Leveraging the **turbo boosting** capability of modern CPUs

- Key: wisely choose what to boost

- GC Boosting
  - Boost GC

- Queue Boosting
  - Boost when the system is “busy”
  - Use event queue stats as “hints”
# Optimization 1: VM Optimization (GC Boost)

<table>
<thead>
<tr>
<th>Application</th>
<th>Non-tail</th>
<th></th>
<th></th>
<th>Tail</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IC</td>
<td>GC</td>
<td>JIT</td>
<td>Native</td>
<td>IC</td>
<td>GC</td>
</tr>
<tr>
<td>Etherpad Lite</td>
<td>0.3%</td>
<td>1.9%</td>
<td>0%</td>
<td>97.8%</td>
<td>7.0%</td>
<td>41.9%</td>
</tr>
<tr>
<td>Todo</td>
<td>1.8%</td>
<td>0.5%</td>
<td>0%</td>
<td>97.7%</td>
<td>0.6%</td>
<td>31.0%</td>
</tr>
<tr>
<td>Lighter</td>
<td>4.9%</td>
<td>4.7%</td>
<td>0%</td>
<td>90.4%</td>
<td>4.4%</td>
<td>45.3%</td>
</tr>
<tr>
<td>Let's Chat</td>
<td>2.7%</td>
<td>3.9%</td>
<td>0%</td>
<td>93.4%</td>
<td>3.6%</td>
<td>48.4%</td>
</tr>
<tr>
<td>Client Manager</td>
<td>3.2%</td>
<td>3.1%</td>
<td>0%</td>
<td>93.7%</td>
<td>3.9%</td>
<td>70.3%</td>
</tr>
</tbody>
</table>
Optimization 1: VM Optimization (GC Boost)

- **Observations:**
  - GCs are infrequent, little overall energy overhead
  - IPC during GC is relatively high: ~1.3

<table>
<thead>
<tr>
<th>Application</th>
<th>IC</th>
<th>Non-tail</th>
<th></th>
<th>Tail</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IC</td>
<td>GC</td>
<td>JIT</td>
<td>Native</td>
<td>IC</td>
</tr>
<tr>
<td>Etherpad Lite</td>
<td>0.3%</td>
<td>1.9%</td>
<td>0%</td>
<td>97.8%</td>
<td>7.0%</td>
</tr>
<tr>
<td>Todo</td>
<td>1.8%</td>
<td>0.5%</td>
<td>0%</td>
<td>97.7%</td>
<td>0.6%</td>
</tr>
<tr>
<td>Lighter</td>
<td>4.9%</td>
<td>4.7%</td>
<td>0%</td>
<td>90.4%</td>
<td>4.4%</td>
</tr>
<tr>
<td>Let’s Chat</td>
<td>2.7%</td>
<td>3.9%</td>
<td>0%</td>
<td>93.4%</td>
<td>3.6%</td>
</tr>
<tr>
<td>Client Manager</td>
<td>3.2%</td>
<td>3.1%</td>
<td>0%</td>
<td>93.7%</td>
<td>3.9%</td>
</tr>
</tbody>
</table>
Optimization 1: VM Optimization (GC Boost)

- **Observations:**
  - GCs are infrequent, little overall energy overhead
  - IPC during GC is relatively high: ~1.3

- **Implementation**
  - User-space DVFS embedded in Google V8: enter boosting at GC prologues and exit boosting at GC epilogues
  - More benefits if we have access to fine-grained per-core DVFS mechanism
Optimization 2: Queue Boost
Optimization 2: Queue Boost

- More general compute acceleration: Boost when the system is “busy”
Optimization 2: Queue Boost

- More general compute acceleration: Boost when the system is “busy”
- How do you detect that?
Optimization 2: Queue Boost

- More general compute acceleration: Boost when the system is “busy”
- How do you detect that?
- Rely on two queue-related heuristics:
  - # of events in the queue
  - Processing time of the head-of-line event
Optimization 2: Queue Boost

- More general compute acceleration: Boost when the system is “busy”
- How do you detect that?
- Rely on two queue-related heuristics:
  - # of events in the queue
  - Processing time of the head-of-line event

Implementation:
- Periodic **Sampling:** Every 1 ms
- Dynamic **Thresholding**
  - Sample the average value of event number and per event processing time
  - Amplify the average value to decide a dynamic threshold by 2-3x
Evaluation

- **Our system**: normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
  - GC Tuning + GC Boost + Queue Boost
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
  - GC Tuning + GC Boost + Queue Boost

- **Baseline**
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
  - GC Tuning + GC Boost + Queue Boost

- **Baseline**
  - **Static Frequency:** 3.3GHz and 4.0GHz
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
  - GC Tuning + GC Boost + Queue Boost

- **Baseline**
  - **Static Frequency:** 3.3GHz and 4.0GHz
  - **Adrenaline** [HPCA 2015]
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
  - GC Tuning + GC Boost + Queue Boost

- **Baseline**
  - **Static Frequency:** 3.3GHz and 4.0GHz
  - **Adrenaline** [HPCA 2015]
  - **Rubik** [MICRO 2015]
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
  - GC Tuning + GC Boost + Queue Boost

- **Baseline**
  - **Static Frequency:** 3.3GHz and 4.0GHz
  - **Adrenaline** [HPCA 2015]
  - **Rubik** [MICRO 2015]
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
  - GC Tuning + GC Boost + Queue Boost

- **Baseline**
  - **Static Frequency:** 3.3GHz and 4.0GHz
  - **Adrenaline** [HPCA 2015]
  - **Rubik** [MICRO 2015]
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
  - GC Tuning + GC Boost + Queue Boost

- **Baseline**
  - **Static Frequency:** 3.3GHz and 4.0GHz
  - **Adrenaline** [HPCA 2015]
  - **Rubik** [MICRO 2015]
Evaluation

- **Our system:** normal operating frequency at 2.6 GHz, boosts to max 4.0 GHz during GC boost and Queue boost.

- **Different Variants:**
  - GC Boost
  - GC Boost with GC Parameter Tuning
  - Queue Boost
  - GC Tuning + GC Boost + Queue Boost

- **Baseline**
  - **Static Frequency:** 3.3GHz and 4.0GHz
  - **Adrenaline** [HPCA 2015]
  - **Rubik** [MICRO 2015]
Pareto-dominate existing solutions; 14-21% tail reduction with only 3-14% energy overhead over baseline.
Conclusions

Node.js uniquely combines event-driven programming model and managed language runtime, presenting new landscape and challenges to tail latency optimizations.
Conclusions

Node.js uniquely combines event-driven programming model and managed language runtime, presenting new landscape and challenges to tail latency optimizations.

Event-dependency graph (EDG) and event-critical path (ECP) critical to deconstruct tail latency in Node.js.
Conclusions

Node.js uniquely combines event-driven programming model and managed language runtime, presenting new landscape and challenges to tail latency optimizations.

Event-dependency graph (EDG) and event-critical path (ECP) critical to deconstruct tail latency in Node.js.

Intelligently leverage existing hardware features, turbo boosting in particular, to reduce latency with little to none energy overhead.