Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale

Akshitha Sriraman*

Abhishek Dhanotia

The 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)
Rapid Increase in Modern Web Services

Stringent Service-Level Objectives

Point of diminishing returns

End of Dennard scaling

$\mu$service SLOs + end of Dennard scaling $\rightarrow$ increase in custom HW
But.... What Should we Accelerate?

EVERYTHING?!
Key (?) Acceleration Opportunities

Too many custom HW -> EXPENSIVE

Feed microservice

Accelerating ML inference

What is end-to-end Feed’s throughput increase? < 1.5x!
Acceleration Opportunities at Facebook

Which µservice operations consume the most CPU cycles?

Do µservices have common overheads that can inspire future HW designs?
Contribution 1: Where Did My Cycles Go?

Leaf function: `memcpy()`

μservice functionalities:
- ML inference

Orchestration

High & common orchestration overheads: Worth accelerating
Contribution 2: Accelerometer

Accelerometer: Projects perf. bounds early in the HW design phase
Contribution 3: Validating Accelerometer

Validating in production: Three retrospective case studies
Contribution 4: Applying Accelerometer

How can you apply accelerometer to make good HW investments?
Introduction
Post Dennard scaling: What to accelerate?

Characterization
Where do data center cycles go?

Accelerometer
Analytical model for HW acceleration

Validation
Production case studies

Application
How we use Accelerometer
Introduction
Post Dennard scaling: What to accelerate?

Characterization
Where do data center cycles go?

Accelerometer
Analytical model for HW acceleration

Validation
Production case studies

Application
How we use Accelerometer

Leaf function
Service functionality
# Leaf Function Characterization

<table>
<thead>
<tr>
<th>Leaf category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Copy, allocation, free, compare</td>
</tr>
<tr>
<td>Kernel</td>
<td>Scheduling, interrupt handling, network comm., mem. mgmt.</td>
</tr>
<tr>
<td>Hashing</td>
<td>SHA</td>
</tr>
<tr>
<td>Synchronization</td>
<td>C++ atomics, mutex, spin locks, CAS</td>
</tr>
<tr>
<td>ZSTD</td>
<td>(De)compression</td>
</tr>
<tr>
<td>Math</td>
<td>Intel’s MKL, AVX</td>
</tr>
<tr>
<td>SSL</td>
<td>En(de)cription</td>
</tr>
<tr>
<td>C Libraries</td>
<td>Search algorithms, array &amp; string compute</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Assorted functions</td>
</tr>
</tbody>
</table>

### Memory and kernel leaf functions dominate

<table>
<thead>
<tr>
<th>Category</th>
<th>Web</th>
<th>Feed1</th>
<th>Feed2</th>
<th>Ads1</th>
<th>Ads2</th>
<th>Cache1</th>
<th>Cache2</th>
<th>Google [Kanev’15]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>37</td>
<td>7</td>
<td>32</td>
<td>31</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feed1</td>
<td>8</td>
<td>32</td>
<td>77</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feed2</td>
<td>20</td>
<td>4</td>
<td>5</td>
<td>13</td>
<td>37</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ads1</td>
<td>28</td>
<td>11</td>
<td>3</td>
<td>31</td>
<td>17</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ads2</td>
<td>28</td>
<td>12</td>
<td>5</td>
<td>5</td>
<td>42</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache1</td>
<td>26</td>
<td>1</td>
<td>22</td>
<td>19</td>
<td>3</td>
<td>6</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>Cache2</td>
<td>19</td>
<td>44</td>
<td>1</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Google</td>
<td>13</td>
<td>19</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>62</td>
</tr>
</tbody>
</table>

**FB microservices**
μService Functionality Characterization

Facebook's production microservices

Orchestration overheads dominate across microservices
### μService Functionality Breakdown

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>(In)secure IO</td>
<td>IO send/receive</td>
</tr>
<tr>
<td>IO pre/post processing</td>
<td>Copy, alloc., etc before/after IO</td>
</tr>
<tr>
<td>(De)compression</td>
<td>(De)compression logic</td>
</tr>
<tr>
<td>(De)serialization</td>
<td>RPC De(serialization)</td>
</tr>
<tr>
<td>Feature extraction</td>
<td>Feature vector creation in ML services</td>
</tr>
<tr>
<td>Prediction/Ranking</td>
<td>ML inference</td>
</tr>
<tr>
<td>Core app. logic</td>
<td>Core business logic</td>
</tr>
<tr>
<td>Logging</td>
<td>Creating, reading, updating logs</td>
</tr>
<tr>
<td>Thread pool mgmt.</td>
<td>Creating, deleting, synchronizing threads</td>
</tr>
</tbody>
</table>

#### Examples of Cycles Spent

- **Web**: 21, 17, 11, 10, 18, 23
- **Feed1**: 7, 15, 13, 57, 3, 5
- **Feed2**: 4, 8, 9, 28, 33, 6, 4, 7
- **Ads1**: 7, 6, 4, 14, 52, 7, 11
- **Ads2**: 4, 4, 24, 58, 7, 9
- **Cache1**: 38, 6, 15, 4, 31, 6
- **Cache2**: 52, 8, 3, 3, 35, 3
- **Google's services [Kanev '15]**: 4, 6, 5, 85

**Common orchestration overheads dominate across microservices**
Characterization Takeaways

Accelerating orchestration can improve perf. across global fleet
Investing in HW Acceleration

Offload-induced overheads
Service execution
Orchestration
Core app. logic

Risky @scale due to perf. bounds from offload overheads
Accelerometer: Analytical Model

Acceleration strategy

- On-chip
- Off-chip
- Remote

Threading design for offload

- Synchronous
- Asynchronous

Models throughput speedup and per-request latency reduction
Synchronous Offload

Unaccelerated:

\[(1 - \alpha) C \quad o_0\]

To accelerate:

\[\alpha C\]

Host cycles

Sync. thread returns

Interface

Accelerator

Speedup = \[
\frac{C}{(1 - \alpha)C + \alpha C/A + (o_0 + Q + L) n}
\]

Accelerator cycles critically affect speedup & latency reduction
Sync. Offload with Thread Oversubscription

Unaccelerated:

\[(1 - \alpha) C\]  

\[o_0\]  

offload  

\[\text{Interface}\]  

\[\text{Accelerator}\]  

\[\alpha C/A\]  

\[o_1\]  

\[o_1\]  

Thread runs

Host cycles

Thread returns

*Context switch penalties impact speedup & latency reduction*
Asynchronous Offload

Unaccelerated: 

\[(1 - \alpha) C_o\]

Accelerator cycles do not critically affect speedup
Introduction
Post Dennard scaling: What to accelerate?

Characterization
Where do data center cycles go?

Accelerometer
Analytical model for HW acceleration

Validation
Production case studies

Application
How we use Accelerometer
Validating Accelerometer: Encryption

(1 - α) C o₀ = 0  \[ \text{offload} \]

L = 2530 cycles

\[ \text{Async. opr.} \]

Host cycles
C = 2.3 \times 10^9 \text{cycles}

\[ \alpha C; \alpha = 0.19 \]

\[ \text{Interface} \]

\[ \text{Accelerator} \]

Estimated speedup = 8.6%

Real speedup = 7.5%

Accelerometer estimates speedup with <= 3.7% error
Introduction
Post Dennard scaling: What to accelerate?

Characterization
Where do data center cycles go?

Accelerometer
Analytical model for HW acceleration

Validation
Production case studies

Application
How we use Accelerometer
Applying Accelerometer: Compression

Accelerometer can identify good accelerator investments
Introduction
Post Dennard scaling: What to accelerate?

Characterization
Where do data center cycles go?

Accelerometer
Analytical model for HW acceleration

Validation
Production case studies

Application
How we use Accelerometer
Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale

Akshitha Sriraman*
Abhishek Dhanotia

The 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)