Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices

Zhipeng Jia, Emmett Witchel
University of Texas at Austin
Motivation: Two Trends in Cloud Computing

Serverless functions / Function as a service (FaaS)
- User provides *stateless* functions, that are executed on cloud provider’s infrastructure
- Benefits: elasticity, and pay-as-you-go billing

Microservices
- Organize online applications with *single-purpose*, *loosely-coupled* microservices
- Benefits: composable software design
Motivation: Serverless Microservices

- Microservices are mostly implemented as RPC servers
- Stateless RPC handlers naturally fit in the FaaS paradigm

But not performant!!

<table>
<thead>
<tr>
<th></th>
<th>RPC servers</th>
<th>AWS Lambda</th>
</tr>
</thead>
<tbody>
<tr>
<td>median latency</td>
<td>2.34ms</td>
<td>26.94ms (11.5x)</td>
</tr>
<tr>
<td>tail latency</td>
<td>6.48ms</td>
<td>160.8ms (24.8x)</td>
</tr>
</tbody>
</table>

SocialNetwork microservices from DeathStarBench [ASPLOS ‘19] Running under light load (100 QPS)
RPC trace from SocialNetwork microservices
RPC trace from SocialNetwork microservices

**Observation 1:**

μs-scale execution time
Observation 2: High invocation rate (can be >100K/s)

RPC trace from SocialNetwork microservices
Performance Goals for Nightcore

- Observation 1: *us-scale* execution time
- Observation 2: high invocation rate (>100K/s)

<table>
<thead>
<tr>
<th></th>
<th>Function Execution Time</th>
<th>Invocation Latency</th>
<th>Invocation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current FaaS</td>
<td>&gt;100ms</td>
<td>1-10s of ms</td>
<td>&lt;10K/min</td>
</tr>
<tr>
<td>runtime</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current FaaS workloads: video processing, distributed compilation, data analytics, etc.

*Invocation latency: duration between function request and the start of function execution*
Performance Goals for Nightcore

- Observation 1: **us-scale** execution time
- Observation 2: high invocation rate (>100K/s)

<table>
<thead>
<tr>
<th></th>
<th>Function Execution Time</th>
<th>Invocation Latency</th>
<th>Invocation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current FaaS runtime</td>
<td>&gt;100ms</td>
<td>1-10s of ms</td>
<td>&lt;10K/min</td>
</tr>
<tr>
<td>FaaS runtime for microservices</td>
<td>100s of μs</td>
<td>&lt;100μs</td>
<td>&gt;100K/s</td>
</tr>
</tbody>
</table>

Invocation latency: duration between function request and the start of function execution
Nightcore’s Goals are Challenging Because We Are Vulnerable to *Killer Microseconds*

Microsecond-scale I/O means tension between performance and productivity that will need new latency-mitigating ideas, including in hardware.

**BY LUIZ BARROSO, MIKE MARTY, DAVID PATTERSON, AND PARTHASARATHY RANGANATHAN**

**Attack of the Killer Microseconds**

Microsecond-scale events:

- Networking
- TCP/IP stack
- RPC protocol
- Context switch
- Thread scheduling
- ……

Where hides our *killer microseconds*?
Nightcore Design

Hunting for “the killer microseconds” in the regime of FaaS
Nightcore’s Techniques

- Optimizing locality of internal function calls
- High optimizations for local I/Os
  - Low-latency message channels
  - Event-driven concurrency
- Managing per-microservice concurrency to mitigate load variation
Nightcore’s Techniques

- Optimizing locality of internal function calls
- High optimizations for local I/Os
  - Low-latency message channels
  - Event-driven concurrency
- Managing per-microservice concurrency to mitigate load variation
High-Level Design of a FaaS Runtime

Function invocation requests

Frontend (e.g. API gateway)

Dispatch to backends

Backend (e.g. VM)
Execution environment (e.g. container)

…...

More backends …...

Separation of frontend and backend

- Adopted by Apache OpenWhisk and OpenFaaS
- Scaling the system by adding backends
RPC trace from SocialNetwork microservices

NGINX frontend

New Tweet

Compose-post

UploadMedia

UploadUserWithUserId

UploadCreator

UploadUniqueId

UploadText

UploadUrls

UploadUserMention

StorePost

WriteUserTimeline

FanoutHomeTimelines

GetFollows

Stateful service

Stateless service (running on FaaS)
RPC trace from SocialNetwork microservices
Observation: High Ratio of Internal Calls

Function calls that are internal w.r.t. FaaS system

Frequent in microservices

<table>
<thead>
<tr>
<th>Microservice workloads</th>
<th>Social Network</th>
<th></th>
<th>Movie Reviewing</th>
<th>Hotel Reservation</th>
<th>Hipster Shop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>write</td>
<td>mixed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>% of internal fn calls</td>
<td>66.7%</td>
<td>62.3%</td>
<td>69.2%</td>
<td>79.2%</td>
<td>85.1%</td>
</tr>
</tbody>
</table>
Optimizing Locality for Internal Function Calls

Internal function calls always go through frontend

Skip frontend for internal function calls
Overview of Nightcore

Worker server

Gateway

Per-Fn dispatching queues

Fn₁
Fn₂
......
Fnₙ

Nightcore’s Engine

Nightcore’s runtime library

Launcher

Worker threads

Fn worker

Fn container

(more function containers)

fast path for internal function call

VM or Bare metal machine
Docker container
Process
User-provided function code
Overview of Nightcore

Worker server

Gateway

Per-Fn dispatching queues

\( F_{n_1} \)

\( F_{n_2} \)

......

\( F_{n_N} \)

Nightcore’s Engine

Nightcore’s runtime library

Worker threads

Fn worker

Launcher

Fn container

(more function containers)

Docker container

Process

User-provided function code

VM or Bare metal machine

Frontend and Backend
Overview of Nightcore

Worker server

Gateway

Per-Fn dispatching queues

\( \text{Fn}_1 \):


\( \text{Fn}_2 \):


......

\( \text{Fn}_N \):


Nightcore’s Engine

Nightcore’s runtime library

Worker threads

Launcher

Fnc container

(more function containers)

\text{fast path for internal function call}

User-provided function code

VM or Bare metal machine

Docker container

Process

User-provided function code

Fast path for internal function calls
Function Containers
Execution environments for serverless functions

Worker server

Per-Fn dispatching queues

Fn₁:  
Fn₂:  
......
Fnₙ:  

Nightcore’s Engine

Nightcore’s runtime library

Launcher

Worker threads

Fn worker

Docker container

VM or Bare metal machine

Docker container

Process

User-provided function code
Function Containers

Execution environments for serverless functions

Launcher launches new function workers, and worker threads
Nightcore’s Engine

The main Nightcore process running on each worker server.

Worker server

- Gateway
- Per-Fn dispatching queues
  - \( \text{Fn}_1 \)
  - \( \text{Fn}_2 \)
  - \( \ldots \)
  - \( \text{Fn}_N \)
- Nightcore’s runtime library
- Launcher
- Fn container
- Worker threads
- Fn worker

(more function containers)

VM or Bare metal machine
Docker container
Process
User-provided function code

Nightcore’s Engine

Fast path for internal function call
Nightcore’s Engine

The main Nightcore process running on each worker server

Gateway

Per-Fn dispatching queues

\[ \text{Fn}_1, \text{Fn}_2, \ldots, \text{Fn}_N \]

Nightcore’s runtime library

Worker threads

Launcher

Fn container

Worker server

(more function containers)

Receive external function requests from Gateway

Docker container

VM or Bare metal machine

User-provided function code
Nightcore’s Engine

The main Nightcore process running on each worker server

Dispatch function requests to worker threads
Nightcore’s Engine

The main Nightcore process running on each worker server

Worker server

Gateway

Per-Fn dispatching queues

\[ \text{Fn}_1, \text{Fn}_2, \ldots, \text{Fn}_N \]

Nightcore’s runtime library

Launcher

Worker threads

Fn worker

Fn container

Fast path for internal function calls

Fast path for internal function calls

VM or Bare metal machine

Docker container

Process

User-provided function code
Internal Function Request

① $F_{n_y}$ invoked via Nightcore's runtime API
Internal Function Request

1. \( F_n \) invoked via Nightcore's runtime API
2. \( Req \) sent to Nightcore's engine
Internal Function Request

1. $F_{n_y}$ invoked via Nightcore’s runtime API
2. $R_{eq_y}$ sent to Nightcore’s engine
3. Place $req_y$ in the dispatching queue
Internal Function Request

1. $\text{Fn}_y$ invoked via Nightcore’s runtime API
2. $\text{Req}_y$ sent to Nightcore’s engine
3. Place $\text{req}_y$ in the dispatching queue
4. Dispatch $\text{Req}_y$ to worker of $\text{Fn}_y$
Internal Function Request

1. $F_n$ invoked via Nightcore’s runtime API
2. $R_{eq}$ sent to Nightcore’s engine
3. Place $req_y$ in the dispatching queue
4. Dispatch $Req_y$ to worker of $F_n$
5. Worker thread executes code of $F_n$
Internal Function Request

① $F_n_y$ invoked via Nightcore’s runtime API
② Req$_y$ sent to Nightcore’s engine
③ Place req$_y$ in the dispatching queue
④ Dispatch Req$_y$ to worker of $F_n_y$
⑤ Worker thread executes code of $F_n_y$
⑥ Execution of req$_y$ completed
**Internal Function Request**

1. $\text{Fn}_y$ invoked via Nightcore’s runtime API
2. Req$_y$ sent to Nightcore’s engine
3. Place req$_y$ in the dispatching queue
4. Dispatch Req$_y$ to worker of $\text{Fn}_y$
5. Worker thread executes code of $\text{Fn}_y$
6. Execution of req$_y$ completed
7. Send output back to worker of $\text{Fn}_x$
Internal Function Request

1. \( F_n \) invoked via Nightcore’s runtime API
2. \( Rq \) sent to Nightcore’s engine
3. Place \( req_y \) in the dispatching queue
4. Dispatch \( Req_y \) to worker of \( F_n \)
5. Worker thread executes code of \( F_n \)
6. Execution of \( req_y \) completed
7. Send output back to worker of \( F_n \)
8. Execution flow returns back to code of \( F_n \)
Internal Function Request

① $\text{Fn}_y$ invoked via Nightcore’s runtime API
② $\text{Req}_y$ sent to Nightcore’s engine
③ Place $\text{req}_y$ in the dispatching queue
④ Dispatch $\text{Req}_y$ to worker of $\text{Fn}_y$
⑤ Worker thread executes code of $\text{Fn}_y$
⑥ Execution of $\text{req}_y$ completed
⑦ Send output back to worker of $\text{Fn}_x$
⑧ Execution flow returns back to code of $\text{Fn}_x$
Nightcore’s Techniques

- Optimizing locality of internal function calls
- High optimizations for local I/Os
  - Low-latency message channels
  - Event-driven concurrency
- Managing per-microservice concurrency to mitigate load variation
Nightcore’s Low-Latency Message Channel

We need IPC primitive for function worker I/Os

- One straightforward option — a feature-rich RPC framework like gRPC
- But wait, RPC protocols have μs-scale overheads (*killer microseconds*!)

Nightcore builds its own message channels for best performance

- Built on top of OS pipes
- Transmit fixed-size 1KB messages

Deliver messages in **3.4μs**

- In contrast, gRPC over Unix sockets takes 13μs for sending 1KB RPC payloads
Nightcore’s Low-Latency Message Channel

Why choosing 1KB as the message size?

Distribution of RPC sizes across microservices in DeathStarBench
Event-Driven Concurrency for Best Efficiency

Small number of I/O threads

4 threads are sufficient for an invocation rate of 100K/s
Nightcore’s Techniques

- Optimizing locality of internal function calls
- High optimizations for local I/Os
  - Low-latency message channels
  - Event-driven concurrency
- Managing per-microservice concurrency to mitigate load variation
Internal Load Variations within Microservices

Timeline of CPU utilization
Running SocialNetwork microservices at a fixed request rate

Why this happens?
Stage-based nature of microservices → Complex internal load dynamics
New Tweet

NGINX frontend

media
UploadMedia (320μs)

user
UploadUserWithUserId (300μs)

unique-id
UploadUniqueId (330μs)

user-mention
UploadUserMention (690μs)

compose-post
UploadMedia (140μs)

compose-post
UploadCreator (130μs)

compose-post
UploadUniqueId (140μs)

compose-post
UploadUserMention (130μs)

url-shorten
UploadUrls (590μs)

write-home-timeline
FanoutHomeTimelines (640μs)

post-storage
StorePost (260μs)

user-timeline
WriteUserTimeline (650μs)

social-graph
GetFollows (230μs)

compose-post
UploadUrls (140μs)

compose-post
UploadText (1710μs)

UploadText (3640μs)

text

Rate of 100 QPS

Stateful service

Stateless service (running on FaaS)

What is the load for each ???

RPC trace from SocialNetwork microservices
Internal Load Variations within Microservices

Why this happens?

Stage-based nature of microservices → Complex internal load dynamics

Overusing concurrency for bursty load → Worse overall performance

Timeline of CPU utilization
Running SocialNetwork microservices at a *fixed* request rate
Nightcore’s Managed Concurrency

Per-function concurrency target

- Limiting concurrent execution
  → Prevent overuse of concurrency
- Dynamically computed with input load

\[
(\text{concurrency target}) = (\text{invocation rate}) \times (\text{function execution time})
\]

**Computed by exponential weight average**

Timeline of CPU utilization
Running SocialNetwork microservices at a **fixed** request rate

“Flatten the curve”
Nightcore’s Managed Concurrency

Adaptive to load changes
Finally, Do We Achieve Our Performance Goals?

<table>
<thead>
<tr>
<th>FaaS Systems</th>
<th>Invocation Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50th</td>
</tr>
<tr>
<td>AWS Lambda</td>
<td>10.4ms</td>
</tr>
<tr>
<td>OpenFaaS</td>
<td>1.09ms</td>
</tr>
<tr>
<td>Nightcore <em>(external function calls)</em></td>
<td>285μs</td>
</tr>
<tr>
<td>Nightcore <em>(internal function calls)</em></td>
<td>39μs</td>
</tr>
</tbody>
</table>
Evaluation

A nightcore edit is a cover track that speeds up the pitch and time of its source material by 10–30%.
Benchmark Workloads

DeathStarBench [ASPLOS ’19]
- SocialNetwork
- MovieReviewing
- HotelReservation

Google’s HipsterShop microservices

<table>
<thead>
<tr>
<th></th>
<th>Ported services</th>
<th>RPC framework</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social Network</td>
<td>11</td>
<td>Apache Thrift</td>
<td>C++</td>
</tr>
<tr>
<td>Movie Reviewing</td>
<td>12</td>
<td>Apache Thrift</td>
<td>C++</td>
</tr>
<tr>
<td>Hotel Reservation</td>
<td>11</td>
<td>gRPC</td>
<td>Go</td>
</tr>
<tr>
<td>HipsterShop</td>
<td>13</td>
<td>gRPC</td>
<td>Go, Node.js, Python</td>
</tr>
</tbody>
</table>

Systems for Comparison

RPC servers — non-serverless deployment of microservices

OpenFaaS — FaaS system deployed in the same way as Nightcore
Single-Server Experiment

OpenFaaS and Nightcore: one worker VM runs all functions

RPC servers: one VM runs all RPC servers
X-axis: throughput (QPS)
Upper chart: median latency
RPC servers —the ordinary choice for microservices
OpenFaaS — microservices on FaaS, but a worse choice
Nightcore — let FaaS shine for microservices
Nightcore v.s. RPC servers

1.27x to 1.59x higher throughput
up to 34% reduction in tail latency
Performance Evaluation of Nightcore Designs

1/3 throughput of RPC servers
Performance Evaluation of Nightcore Designs

- Throughput close to RPC servers
- Much better tail latency
Performance Evaluation of Nightcore Designs

slightly better than RPC servers
Performance Evaluation of Nightcore Designs

1.33x higher throughput than RPC servers
Weak Scaling of Nightcore

Note: $N$ servers run $N$ times of the request load of 1 server

Similar median latency with more servers
Weak Scaling of Nightcore

Note: $N$ servers run $N$ times of the request load of 1 server

Similar (or better) tail latency with more servers

Except MovieReviewing with 8 servers
But we see a similar spike in tail latencies when using 8 RPC servers
## Comparison (8 Servers)

RPC servers as the baseline (1.0x)

<table>
<thead>
<tr>
<th></th>
<th>Throughput (higher is better)</th>
<th>Tail Latency (lower is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenFaaS</td>
<td>Nightcore</td>
</tr>
<tr>
<td>SocialNetwork</td>
<td>0.29x</td>
<td>1.33x</td>
</tr>
<tr>
<td>MovieReviewing</td>
<td>0.30x</td>
<td>1.36x</td>
</tr>
<tr>
<td>HotelReservation</td>
<td>0.28x</td>
<td>2.93x</td>
</tr>
<tr>
<td>HipsterShop</td>
<td>0.38x</td>
<td>1.87x</td>
</tr>
</tbody>
</table>
## Comparison (8 Servers)

**RPC servers as the baseline (1.0x)**

<table>
<thead>
<tr>
<th>Service</th>
<th>OpenFaaS</th>
<th>Nightcore</th>
<th>OpenFaaS</th>
<th>Nightcore</th>
</tr>
</thead>
<tbody>
<tr>
<td>SocialNetwork</td>
<td>0.29x</td>
<td>1.33x</td>
<td>3.40x</td>
<td>0.34x</td>
</tr>
<tr>
<td>MovieReviewing</td>
<td>0.30x</td>
<td>1.36x</td>
<td>4.44x</td>
<td>0.98x</td>
</tr>
<tr>
<td>HotelReservation</td>
<td>0.28x</td>
<td>2.93x</td>
<td>0.96x</td>
<td>1.06x</td>
</tr>
<tr>
<td>HipsterShop</td>
<td>0.38x</td>
<td>1.87x</td>
<td>1.80x</td>
<td>0.31x</td>
</tr>
</tbody>
</table>

OpenFaaS v.s. RPC servers

- **28% to 38% of throughput increase**
- **Tail latency increase by up to 4.4x**
## Comparison (8 Servers)

RPC servers as the baseline (1.0x)

<table>
<thead>
<tr>
<th></th>
<th>Throughput (higher is better)</th>
<th>Tail Latency (lower is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenFaaS</td>
<td>Nightcore</td>
</tr>
<tr>
<td>SocialNetwork</td>
<td>0.29x</td>
<td>1.33x</td>
</tr>
<tr>
<td>MovieReviewing</td>
<td>0.30x</td>
<td>1.36x</td>
</tr>
<tr>
<td>HotelReservation</td>
<td>0.28x</td>
<td>2.93x</td>
</tr>
<tr>
<td>HipsterShop</td>
<td>0.38x</td>
<td>1.87x</td>
</tr>
</tbody>
</table>

Nightcore v.s. RPC servers

- **1.4x to 2.9x** higher throughput
- up to **69%** reduction in tail latency
Conclusion

Nightcore is a FaaS runtime for μs-scale microservices

Nightcore includes diverse techniques to eliminate μs-scale overheads

Nightcore achieves $1.4x–2.9x$ higher throughput than containerized RPC servers, and up to $69\%$ reduction in tail latency

Nightcore is open source at [github.com/ut-osa/nightcore](https://github.com/ut-osa/nightcore)

“Make it fast, rather than general or powerful”
(Butler W. Lampson, *Hints for Computer System Design*)