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

Zhipeng Jia
The University of Texas at Austin
Austin, TX, USA
zjia@cs.utexas.edu

Emmett Witchel
The University of Texas at Austin
Austin, TX, USA
witchel@cs.utexas.edu

ABSTRACT

The microservice architecture is a popular software engineering approach for building flexible, large-scale online services. Serverless functions, or function as a service (FaaS), provide a simple programming model of stateless functions which are a natural substrate for implementing the stateless RPC handlers of microservices, as an alternative to containerized RPC servers. However, current serverless platforms have millisecond-scale runtime overheads, making them unable to meet the strict sub-millisecond latency targets required by existing interactive microservices.

We present Nightcore, a serverless function runtime with microsecond-scale overheads that provides container-based isolation between functions. Nightcore's design carefully considers various factors having microsecond-scale overheads, including scheduling of function requests, communication primitives, threading models for I/O, and concurrent function executions. Nightcore currently supports serverless functions written in C/C++, Go, Node.js, and Python. Our evaluation shows that when running latency-sensitive interactive microservices, Nightcore achieves 1.36×–2.93× higher throughput and up to 69% reduction in tail latency.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Cloud computing**; **n-tier architectures**.

KEYWORDS

Cloud computing, serverless computing, function-as-a-service, microservices

ACM Reference Format:
Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3445814.3446701

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASPLOS '21, April 19–23, 2021, Virtual, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446701

1 INTRODUCTION

The microservice architecture is a popular software engineering approach for building large-scale online services. It has been widely adopted by large companies such as Amazon, Netflix, LinkedIn, Uber, and Airbnb [1, 4, 5, 42, 47]. The microservice architecture enables a large system to evolve more rapidly because each microservice is developed, tested, and deployed independently [36, 49]. Microservices communicate with each other via pre-defined APIs, mostly using remote procedure calls (RPC) [70]. Hence, the dominant design pattern for microservices is that each microservice is an RPC server and they are deployed on top of a container orchestration platform such as Kubernetes [29, 54, 70].

Serverless cloud computing enables a new way of building microservice-based applications [10, 18, 44, 52], having the benefit of greatly reduced operational complexity (§2). Serverless functions, or function as a service (FaaS), provide a simple programming model of *stateless* functions. These functions provide a natural substrate for implementing *stateless* RPC handlers in microservices, as an alternative to containerized RPC servers. However, readily available FaaS systems have invocation latency overheads ranging from a few to tens of milliseconds [14, 55, 84] (see Table 1), making them a poor choice for latency-sensitive interactive microservices, where RPC handlers only run for hundreds of microseconds to a few milliseconds [70, 83, 100, 101] (see Figure 1). The microservice architecture also implies a high invocation rate for FaaS systems, creating a performance challenge. Taking Figure 1 as an example, one request that uploads a new social media post results in 15 *stateless* RPCs (blue boxes in the figure). Our experiments on this workload show that 100K RPCs per second is a realistic performance goal, achievable under non-serverless deployment using five 8-vCPU RPC server VMs. For a FaaS system to efficiently support interactive microservices, it should achieve at least two performance goals which are not accomplished by existing FaaS systems: (1) invocation latency overheads are well within 100μs; (2) the invocation rate must scale to 100K/s with a low CPU usage.

Some previous studies [62, 98] reduced FaaS runtime overheads to microsecond-scale by leveraging software-based fault isolation (SFI), which weakens isolation guarantees between different functions. We prefer the stronger isolation provided by containers because that is the standard set by containerized RPC servers and provided by popular FaaS systems such as Apache OpenWhisk [50] and OpenFaaS [37]. But achieving our performance goals while providing the isolation of containers is a technical challenge.

We present Nightcore, a serverless function runtime designed and engineered to combine high performance with container-based isolation. Any microsecond-or-greater-scale performance overheads

Table 1: Invocation latencies of a warm nop function.

FaaS systems	50th	99th	99.9th
AWS Lambda	10.4 ms	25.8 ms	59.9 ms
OpenFaaS [37]	1.09 ms	3.66 ms	5.54 ms
Nightcore (external)	285 μs	536 μs	855 μs
Nightcore (internal)	39 μs	107 μs	154 μs

can prevent Nightcore from reaching its performance goal, motivating a "hunt for the killer microseconds" [60] in the regime of FaaS systems.

Existing FaaS systems like OpenFaaS [37] and Apache OpenWhisk [50] share a generic high-level design: all function requests are received by a *frontend* (mostly an API gateway), and then forwarded to independent *backends* where function code executes. The frontend and backends mostly execute on separate servers for fault tolerance, which requires invocation latencies that include at least one network round trip. Although datacenter networking performance is improving, round-trip times (RTTs) between two VMs in the same AWS region range from 101μs to 237μs [25]. Nightcore is motivated by noticing the prevalence of internal function calls made during function execution (see Figure 1). An *internal function call* is one that is generated by the execution of a microservice, not generated by a client (in which case it would be an external function call, received by the gateway). What we call internal function calls have been called "chained function calls" in previous work [98]. Nightcore schedules internal function calls on the same backend server that made the call, eliminating a trip to through the gateway and lowering latency (§ 3.2).

Nightcore's support for internal function calls makes most communication local, which means its inter-process communications (IPC) must be efficient. Popular, feature-rich RPC libraries like gRPC work for IPC (over Unix sockets), but gRPC's protocol adds overheads of ~10μs [60], motivating Nightcore to design its own message channels for IPC (§ 3.1). Nightcore's message channels are built on top of OS pipes, and transmit fixed-size 1KB messages, because previous studies [83, 93] show that 1KB is sufficient for most microservice RPCs. Our measurements show Nightcore's message channels deliver messages in 3.4μs, while gRPC over Unix sockets takes 13μs for sending 1KB RPC payloads.

Previous work has shown microsecond-scale latencies in Linux's thread scheduler [60, 92, 100], leading dataplane OSes [61, 77, 87, 91, 92, 94] to build their own schedulers for lower latency. Nightcore relies on Linux's scheduler, because building an efficient, time-sharing scheduler for microsecond-scale tasks is an ongoing research topic [63, 77, 84, 91, 96]. To support an invocation rate of ≥100K/s, Nightcore's engine (§ 4.1) uses event-driven concurrency [23, 105], allowing it to handle many concurrent I/O events with a small number of OS threads. Our measurements show that 4 OS threads can handle an invocation rate of 100K/s. Furthermore, I/O threads in Nightcore's engine can wake function worker threads (where function code is executed) via message channels, which ensures the engine's dispatch suffers only a single wake-up delay from Linux's scheduler.

Existing FaaS systems do not provide concurrency management to applications. However, stage-based microservices create internal

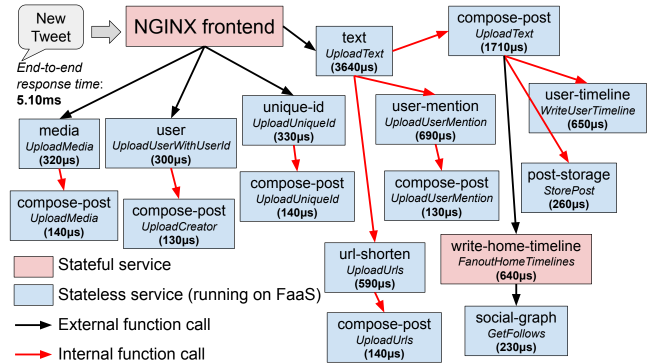


Figure 1: RPC graph of uploading new post in a microservice-based SocialNetwork application [70]. This graph omits stateful services for data caching and data storage.

load variations even under a stable external request rate [73, 105]. Previous studies [38, 73, 104, 105] indicate overuse of concurrency for bursty loads can lead to worse overall performance. Nightcore, unlike existing FaaS systems, actively *manages* concurrency providing dynamically computed targets for concurrent function executions that adjust with input load (§ 3.3). Nightcore's managed concurrency flattens CPU utilization (see Figure 4) such that overall performance and efficiency are improved, as well as being robust under varying request rates (§ 5.2).

We evaluate the Nightcore prototype on four interactive microservices, each with a custom workload. Three are from DeathStarBench [70] and one is from Google Cloud [29]. These workloads are originally implemented in RPC servers, and we port them to Nightcore, as well as OpenFaaS [37] for comparison. The evaluation shows that only by the carefully finding and eliminating microsecond-scale latencies can Nightcore use serverless functions to efficiently implement latency-sensitive microservices.

This paper makes the following contributions.

- Nightcore is a FaaS runtime optimized for microsecond-scale microservices. It achieves invocation latency overheads under 100μs and efficiently supports invocation rates of 100K/s with low CPU usage. Nightcore is publicly available at GitHub `ut-osa/nightcore`.
- Nightcore's design uses diverse techniques to eliminate microsecond-scale overheads, including a fast path for internal function calls, low-latency message channels for IPC, efficient threading for I/O, and function executions with dynamically computed concurrency hints (§3).
- With containerized RPC servers as the baseline, Nightcore achieves 1.36×–2.93× higher throughput and up to 69% reduction in tail latency, while OpenFaaS only achieves 29%–38% of baseline throughput and increases tail latency by up to 3.4× (§5).

2 BACKGROUND

Latency-Sensitive Interactive Microservices. Online services must scale to high concurrency, with response times small enough (a few tens of milliseconds) to deliver an interactive experience [58,

66, 106]. Once built with monolithic architectures, interactive online services are undergoing a shift to microservice architectures [1, 4, 5, 42, 47], where a large application is built by connecting loosely coupled, single-purpose microservices. On the one hand, microservice architectures provide software engineering benefits such as modularity and agility as the scale and complexity of the application grows [36, 49]. On the other hand, staged designs for online services inherently provide better scalability and reliability, as shown in pioneering works like SEDA [105]. However, while the interactive nature of online services implies an end-to-end service-level objectives (SLO) of a few tens of milliseconds, individual microservices face more strict latency SLOs – at the sub-millisecond-scale for leaf microservices [100, 110].

Microservice architectures are more complex to operate compared to monolithic architectures [22, 35, 36], and the complexity grows with the number of microservices. Although microservices are designed to be loosely coupled, their failures are usually very dependent. For example, one overloaded service in the system can easily trigger failures of other services, eventually causing cascading failures [3]. Overload control for microservices is difficult because microservices call each other on data-dependent execution paths, creating dynamics that cannot be predicted or controlled from the runtime [38, 48, 88, 111]. Microservices are often comprised of services written in different programming languages and frameworks, further complicating their operational problems. By leveraging fully managed cloud services (e.g., Amazon's DynamoDB [6], ElasticCache [7], S3 [19], Fargate [12], and Lambda [15]), responsibilities for scalability and availability (as well as operational complexity) are mostly shifted to cloud providers, motivating *serverless microservices* [20, 33, 41, 43–45, 52, 53].

Serverless Microservices. Simplifying the development and management of online services is the largest benefit of building microservices on serverless infrastructure. For example, scaling the service is automatically handled by the serverless runtime, deploying a new version of code is a push-button operation, and monitoring is integrated with the platform (e.g., CloudWatch [2] on AWS). Amazon promotes serverless microservices with the slogan "no server is easier to manage than no server" [44]. However, current FaaS systems have high runtime overheads (Table 1) that cannot always meet the strict latency requirement imposed by *interactive* microservices. Nightcore fills this performance gap.

Nightcore focuses on mid-tier services implementing *stateless* business logic in microservice-based online applications. These mid-tier microservices bridge the user-facing frontend and the data storage, and fit naturally in the programming model of serverless functions. Online data intensive (OLDI) microservices [100] represent another category of microservices, where the mid-tier service fans out requests to leaf microservices for parallel data processing. Microservices in OLDI applications are mostly *stateful* and memory intensive, and therefore are not a good fit for serverless functions. We leave serverless support of OLDI microservices as future work.

The programming model of serverless functions expects function invocations to be short-lived, which seems to contradict the assumption of service-oriented architectures which expect services to be long-running. However, FaaS systems like AWS Lambda allows clients to maintain long-lived connections to their API gateways [8], making a serverless function "service-like". Moreover, because AWS Lambda re-uses execution contexts for multiple function invocations [13], users' code in serverless functions can also cache reusable resources (e.g., database connections) between invocations for better performance [17].

Optimizing FaaS Runtime Overheads. Reducing start-up latencies, especially cold-start latencies, is a major research focus for FaaS runtime overheads [57, 64, 67, 89, 90, 98]. Nightcore assumes sufficient resources have been provisioned and relevant function containers are in warm states which can be achieved on AWS Lambda by using provisioned concurrency (AWS Lambda strongly recommends provisioned concurrency for latency-critical functions [40]). As techniques for optimizing cold-start latencies [89, 90] become mainstream, they can be applied to Nightcore.

Invocation latency overheads of FaaS systems are largely overlooked, as recent studies on serverless computing focus on data intensive workloads such as big data analysis [75, 95], video analytics [59, 69], code compilation [68], and machine learning [65, 98], where function execution times range from hundreds of milliseconds to a few seconds. However, a few studies [62, 84] point out that the millisecond-scale invocation overheads of current FaaS systems make them a poor substrate for microservices with microsecond-scale latency targets. For serverless computing to be successful in new problem domains [71, 76, 84], it must address microsecond-scale overheads.

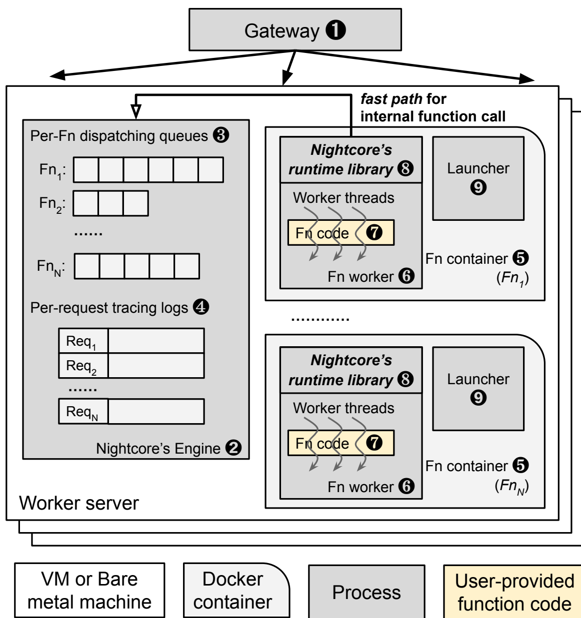
3 DESIGN

Nightcore is designed to run serverless functions with sub-millisecond-scale execution times, and to efficiently process internal function calls, which are generated during the execution of a serverless function (not by an external client). Nightcore exposes a serverless function interface that is similar to AWS Lambda: users provide stateless function handlers written in supported programming languages. The only addition to this simple interface is that Nightcore's runtime library provides APIs for fast internal function invocations.

3.1 System Architecture

Figure 2 depicts Nightcore's design which mirrors the design of other FaaS systems starting with the separation of *frontend* and *backend*. Nightcore's *frontend* is an API gateway for serving external function requests and other management requests (e.g., to register new functions), while the *backend* consists of a number of independent **worker servers**. This separation eases availability and scalability of Nightcore, by making the *frontend* API gateway fault tolerant and horizontally scaling *backend* worker servers. Each worker server runs a Nightcore engine process and function containers, where each function container has one registered serverless function, and each function has only one container on each worker server. Nightcore's engine directly manages function containers and communicates with worker threads within containers.

Internal Function Calls. Nightcore optimizes internal function calls locally on the same worker server, without going through the API gateway. Figure 2 depicts this fast path in Nightcore's runtime



❶ Gateway	• Accept external function requests • Load balance requests to worker servers
❷ Engine	• The main Nightcore process on each worker server, which communicates with Gateway ❶, launchers ❽, and worker threads inside Fn workers ❻ • Maintain per-function dispatching queues ❸ and per-request tracing logs ❹
❸ Dispatching queues	• Function requests queued here • Dispatch function requests to worker threads in Fn worker ❻
❹ Tracing logs	• Track life-cycle of all function invocations, by recording receive, dispatch, and completion timestamps
❺ Fn container	• Execution environment for individual functions • Consists of Fn worker ❻ and Launcher ❽ processes
❻ Fn worker process	• Multiple worker threads execute user-provided function code ❼, and call a runtime library ❽ for fast, internal function call • Implementation tailored to each supported programming language
❼ User-provided Fn code	• Stateless function code written in supported programming language (C/C++, Go, Python, or Node.js) • Executed on worker threads within Fn worker process ❻
❽ Runtime library	• Fast path for internal function call: talk directly with Engine to enqueue the function call ❸, entirely bypassing Gateway ❶
❾ Launcher	• Launch new Fn worker ❻ or worker threads

Figure 2: Architecture of Nightcore (§ 3.1).

library which executes inside a function container. By optimizing the locality of dependent function calls, Nightcore brings performance close to a monolithic design. At the same time, different microservices remain logically independent and they execute on different worker servers, ensuring there is no single point of failure. Moreover, Nightcore preserves the engineering and deployment benefits of microservices such as diverse programming languages and software stacks.

Nightcore's performance optimization for internal function calls assumes that an individual worker server is capable of running most function containers from a single microservice-based application [1]. We believe this is justified because we measure small working sets for stateless microservices. For example, when running SocialNetwork [70] at its saturation throughput, the 11 stateless microservice containers consume only 432 MB of memory, while the host VM is provisioned with 16 GB. As current datacenter servers have growing numbers of CPUs and increasing memory sizes (e.g., AWS EC2 VMs have up to 96 vCPUs and 192 GB of memory), a single server is able to support the execution of thousands of containers [98, 109]. When it is not possible to schedule function containers on the same worker server, Nightcore falls back to scheduling internal function calls on different worker servers through the gateway.

Gateway. Nightcore's gateway (Figure 2①) performs load balancing across worker servers for incoming function requests and forwards requests to Nightcore's engine on worker servers. The gateway also uses external storage (e.g., Amazon's S3) for saving function metadata and it periodically monitors resource utilizations on all worker servers, to know when it should increase capacity by launching new servers.

Engine. The engine process (Figure 2②) is the most critical component of Nightcore for achieving microsecond-scale invocation latencies, because it invokes functions on each worker server. Nightcore' engine responds to function requests from both the gateway and from the runtime library within function containers. It creates low-latency message channels to communicate with function workers and launchers inside function containers (§ 4.1). Nightcore's engine is event driven (Figure 5) allowing it to manage hundreds of message channels using a small number of OS threads. Nightcore's engine maintains two important data structures: (1) Per-function dispatching queues for dispatching function requests to function worker threads (Figure 2③); (2) Per-request tracing logs for tracking the life cycle of all inflight function invocations, used for computing the proper concurrency level for function execution (Figure 2④).

Function Containers. Function containers (Figure 2⑤) provide isolated environments for executing user-provided function code. Inside the function container, there is a launcher process, and one or more worker processes depending on the programming language implementation (see § 4.2 for details). Worker threads within worker processes receive function requests from Nightcore's engine, and execute user-provided function code. Worker processes also contain a Nightcore runtime library, exposing APIs for user-provided function code. The runtime library includes APIs for fast internal function calls without going through the gateway. Nightcore's internal function calls directly contact the dispatcher to enqueue the calls that are executed on the same worker server without having to involve the gateway.

Nightcore has different implementations of worker processes for each supported programming language. The notion of "worker

[1]Nightcore also needs to know which set of functions form a single application. In practice, this knowledge comes directly from the developer, e.g., Azure Functions allow developers to organize related functions as a single *function app* [34].

threads" is particularly malleable because different programming languages have different threading models. Futhermore, Nightcore's engine does not distinguish worker threads from worker processes, as it maintains communication channels with each individual worker thread. For clarity of exposition we assume the simplest case in this Section (which holds for the C/C++ implementation), where "worker threads" are OS threads (details for other languages in § 4.2).

Isolation in Nightcore. Nightcore provides container-level isolation between different functions, but does not guarantee isolation between different invocations of the same function. We believe this is a reasonable trade-off for microservices, as creating a clean isolated execution environment within tens of microseconds is too challenging for current systems. When using RPC servers to implement microservices, different RPC calls of the same service can be concurrently processed within the same process, so Nightcore's isolation guarantee is as strong as containerized RPC servers.

Previous FaaS systems all have different trade-offs between isolation and performance. OpenFaaS [51] allows concurrent invocations within the same function worker process, which is the same as Nightcore. AWS Lambda [13] does not allow concurrent invocations in the same container/MicroVM but allows execution environments to be re-used by subsequent invocations. SAND [57] has two levels of isolation–different applications are isolated by containers but concurrent invocations within the same application are only isolated by processes. Faasm [98] leverages the software-based fault isolation provided by WebAssembly, allowing a new execution environment to be created within hundreds of microseconds, but it relies on language-level isolation which is weaker than container-based isolation.

Message Channels. Nightcore's message channels are designed for low-latency message passing between its engine and other components, which carry fixed-size 1KB messages. The first 64 bytes of a message is the header which contains the message type and other metadata, while the remaining 960 bytes are message payload. There are three types of messages relevant to function invocations:

(1) *Dispatch*, used by engine for dispatching function requests to worker threads (④ in Figure 3).
(2) *Completion*, used by function worker threads for sending outputs back to the engine (⑥ in Figure 3), as well as by the engine for sending outputs of internal function calls (⑦ in Figure 3).
(3) *Invoke*, used by Nightcore's runtime library for initiating internal function calls (② in Figure 3).

When payload buffers are not large enough for function inputs or outputs, Nightcore creates extra shared memory buffers for exchanging data. In our experiments, these overflow buffers are needed for less than 1% of the messages for most workloads, though HipsterShop needs them for 9.7% of messages. When overflow buffers are required, they fit within 5KB 99.9% of the time. Previous work [83] has shown that 1KB is sufficient for more than 97% of microservice RPCs.

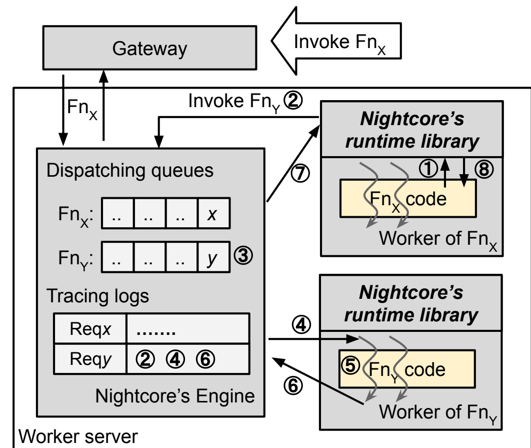


Figure 3: Diagram of an internal function call (§ 3.2).

3.2 Processing Function Requests

Figure 3 shows an example with both an external and internal function call. Suppose code of Fn_x includes an invocation of Fn_y. In this case, Fn_y is invoked via Nightcore's runtime API (①). Then, Nightcore's runtime library generates a unique ID (denoted by req_y) for the new invocation, and sends an internal function call request to Nightcore's engine (②). On receiving the request, the engine writes req_y's receive timestamp (also ②). Next, the engine places req_y in the dispatching queue of Fn_y ③. Once there is an idle worker thread for Fn_y and the concurrency level of Fn_y allows, the engine will dispatch req_y to it, and records req_y's dispatch timestamp in its tracing log (④). The selected worker thread executes Fn_y's code (⑤) and sends the output back to the engine (⑥). On receiving the output, the engine records request req_y's completion timestamp (also ⑥), and directs the function output back to Fn_x's worker (⑦). Finally, execution flow returns back to user-provided Fn_x ⑧.

3.3 Managing Concurrency for Function Executions (τ_k)

Nightcore maintains a pool of worker threads in function containers for concurrently executing functions, but deciding the size of thread pools can be a hard problem. One obvious approach is to always create new worker threads when needed, thereby maximizing the concurrency for function executions. However, this approach is problematic for microservice-based applications, where one function often calls many others. Maximizing the concurrency of function invocations with high fanout can have a domino effect that overloads a server. The problem is compounded when function execution time is short. In such cases, overload happens too quickly for a runtime system to notice it and respond appropriately.

To address the problem, Nightcore adaptively manages the number of concurrent function executions, to achieve the highest useful concurrency level while preventing instantaneous server overload. Following *Little's law*, the ideal concurrency can be estimated as the product of the average request rate and the average processing time. For a registered function Fn_k, Nightcore's engine maintains exponential moving averages of its invocation rate (denoted by λ_k)

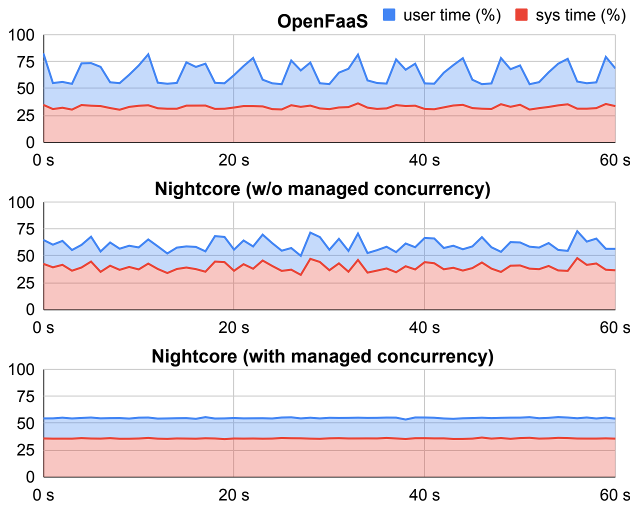


Figure 4: CPU utilization timelines of OpenFaaS, and Nightcore (w/o and with managed concurrency), running SocialNetwork microservices [70] at a fixed rate of 500 QPS for OpenFaaS and 1200 QPS for Nightcore.

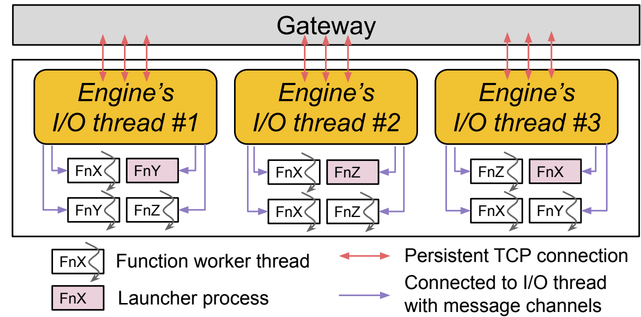


Figure 5: Event-driven I/O threads in Nightcore's engine (§ 4.1).

and function execution time (denoted by t_k). Both are computed from request tracing logs. Nightcore uses their product $\lambda_k \cdot t_k$ as the concurrency hint (denoted by τ_k) for function Fn_k.

When receiving an invocation request of Fn_k, the engine will only dispatch the request if there are fewer than τ_k concurrent executions of Fn_k. Otherwise, the request will be queued, waiting for other function executions to finish. In other words, the engine ensures the maximum concurrency of Fn_k is τ_k at any moment. Note that Nightcore's approach is adaptive because τ_k is computed from two exponential moving averages (λ_k and t_k), that change over time as new function requests are received and executed. To realize the desired concurrency level, Nightcore must also maintain a worker thread pool with at least τ_k threads. However, the dynamic nature of τ_k makes it change rapidly (see Figure 6), and frequent creation and termination of threads is not performant. To modulate the dynamic values of τ_k, Nightcore allows more than τ_k threads to exist in the pool, but only uses τ_k of them. It terminates extra threads when there are more than $2\tau_k$ threads.

Nightcore's managed concurrency is fully automatic, without any knowledge or hints from users. The concurrency hint (τ_k) changes frequently at the scale of microseconds, to adapt to load variation from microsecond-scale microservices (§ 5.2) . Figure 4 demonstrates the importance of managing concurrency levels instead of maximizing them. Even when running at a fixed input rate, CPU utilization varies quite a bit for both OpenFaaS and Nightcore when the runtime maximizes the concurrency. On the other hand, managing concurrency with hints has a dramatic "flatten-the-curve" benefit for CPU utilization.

4 IMPLEMENTATION

Nightcore's API gateway and engine consists of 8,874 lines of C++. Function workers are supported in C/C++, Go, Node.js, and Python, requiring 1,588 lines of C++, 893 lines of Go, 57 lines of JavaScript, and 137 lines of Python.

Nightcore's engine (its most performance-critical component) is implemented in C++. Garbage collection can have a significant impact for latency-sensitive services [107] and short-lived routines [27, 28]. Both OpenFaaS [37] and Apache OpenWhisk [50] are implemented with garbage-collected languages (Go and Scala, respectively), but Nightcore eschews garbage collection in keeping with its theme of addressing microsecond-scale latencies.

4.1 Nightcore's Engine

Figure 5 shows the event-driven design of Nightcore's engine as it responds to I/O events from the gateway and message channels. Each I/O thread maintains a fixed number of persistent TCP connections to the gateway for receiving function requests and sending back responses, while message channels are assigned to I/O threads with a round-robin policy. Individual I/O threads can only read from and write to their own TCP connections and message channels. Shared data structures including dispatching queues and tracing logs are protected by mutexes, as they can be accessed by different I/O threads.

Event-Driven IO Threads. Nightcore's engine adopts `libuv` [32], which is built on top of the `epoll` system call, to implement its event-driven design. `libuv` provides APIs for watching events on file descriptors, and registering handlers for those events. Each IO thread of the engine runs a `libuv` event loop, which polls for file descriptor events and executes registered handlers.

Message Channels. Nightcore's messages channels are implemented with two Linux pipes in opposite directions to form a full-duplex connection. Meanwhile, shared memory buffers are used when inline payload buffers are not large enough for function inputs or outputs (§ 3.1). Although shared memory allows fast IPC at memory speed, it lacks an efficient mechanism to notify the consumer thread when data is available. Nightcore's use of pipes and shared memory gets the best of both worlds. It allows the consumer to be eventually notified through a blocking read on the pipe, and at the same time, it provides the low latency and high throughput of shared memory when transferring large message payloads.

As the engine and function workers are isolated in different containers, Nightcore mounts a shared `tmpfs` directory between their

containers, to aid the setup of pipes and shared memory buffers. Nightcore creates named pipes in the shared `tmpfs`, allowing function workers to connect. Shared memory buffers are implemented by creating files in the shared `tmpfs`, which are `mmap`ed with the `MAP_SHARED` flag by both the engine and function workers. Docker by itself supports sharing IPC namespaces between containers [31], but the setup is difficult for Docker's cluster mode. Nightcore's approach is functionally identical to IPC namespaces, as Linux's System V shared memory is internally implemented by `tmpfs` [46].

Communications between Function Worker Threads. Individual worker threads within function containers connect to Nightcore's engine with a message channel for receiving new function requests and sending responses (④ and ⑥ in Figure 3). A worker thread can be either *busy* (executing function code) or *idle*. During the execution of function code, the worker thread's message channel is also used by Nightcore's runtime library for internal function calls (② and ⑦ in Figure 3). When a worker thread finishes executing function code, it sends a response message with the function output to the engine and enters the idle state. An idle worker thread is put to sleep by the operating system, but the engine can wake it by writing a function request message to its message channel. The engine tracks the busy/idle state of each worker so there is no queuing at worker threads, the engine only dispatches requests to idle workers.

Mailbox. The design of Nightcore's engine only allows individual I/O threads to write data to message channels assigned to it (shown as violet arrows in Figure 5). In certain cases, however, an I/O thread needs to communicate with a thread that does not share a message channel. Nightcore routes these requests using per-thread mailboxes. When an I/O thread drops a message in the mailbox of another thread, `uv_async_send` (using `eventfd` [24] internally) is called to notify the event loop of the owner thread.

Computing Concurrency Hints (τ_k). To properly regulate the amount of concurrent function executions, Nightcore's engine maintains two exponential moving averages λ_k (invocation rate) and t_k (processing time) for each function Fn_k (§ 3.3). Samples of invocation rates are computed as $1/$(interval between consecutive Fn_k invocations), while processing times are computed as intervals between dispatch and completion timestamps, excluding queueing delays (the interval between receive and dispatch timestamps) from sub-invocations. Nightcore uses a coefficient $\alpha = 10^{-3}$ for computing exponential moving averages.

4.2 Function Workers

Nightcore executes user-provided function code in its function worker processes (§ 3.1). As different programming languages have different abstractions for threading and I/O, Nightcore has different function worker implementations for them.

Nightcore's implementation of function workers also includes a runtime library for fast internal function calls. Nightcore's runtime library exposes a simple API `output := nc_fn_call(fn_name, input)` to user-provided function code for internal function calls. Furthermore, Nightcore's runtime library provides Apache Thrift [9] and gRPC [30] wrappers for its function call API, easing porting of existing Thrift-based and gRPC-based microservices to Nightcore.

C/C++. Nightcore's C/C++ function workers create OS threads for executing user's code, loaded as dynamically linked libraries. These OS threads map to "worker threads" in Nightcore's design (§ 3.1 and Figure 2). To simplify the implementation, each C/C++ function worker process only runs one worker thread, and the launcher will fork more worker processes when the engine asks for more worker threads.

Go. In Go function workers, "worker threads" map to goroutines, the user-level threads provided by Go's runtime, and the launcher only forks one Go worker process. Users' code are compiled together with Nightcore's Go worker implementation, as Go's runtime does not support dynamic loading of arbitrary Go code [2]. Go's runtime allows dynamically setting the maximum number of OS threads for running goroutines (via `runtime.GOMAXPROCS`), and Nightcore's implementation sets it to $\lceil$worker goroutines/8$\rceil$.

Node.js and Python. Node.js follows an event-driven design where all I/O is asynchronous without depending on multi-threading, while Python is the same when using the asyncio [11] library for I/O. In both cases, Nightcore implements its message channel protocol within their event loops. As there are no parallel threads [3] inside Node.js and Python function workers, launching a new "worker thread" simply means creating a message channel, while the engine's notion of "worker threads" becomes event-based concurrency [23]. Also, `nc_fn_call` is an asynchronous API in Node.js and Python workers, rather than being synchronous in C/C++ and Go workers. For Node.js and Python functions, the launcher only forks one worker process.

5 EVALUATION

We conduct all of our experiments on Amazon EC2 C5 instances in the `us-east-2` region, running Ubuntu 20.04 with Linux kernel 5.4.41. We enable hyperthreading, but disable transparent huge pages.

5.1 Methodology

Microservice Workloads. Nightcore is designed to optimize microservice workloads, so we evaluate it on the four most realistic, publicly available, interactive microservice code bases: SocialNetwork, MovieReviewing, HotelReservation, and HipsterShop. The first three are from DeathStarBench [70], while HipsterShop is a microservice demo from Google Cloud Platform [29]. The workloads are summarized in Table 2.

For the SocialNetwork workload, we tested two load patterns: (1) a pure load of ComposePost requests (shown in Figure 1) (denoted as "write"); (2) a mixed load (denoted as "mixed"), that is a combination of 30% CompostPost, 40% ReadUserTimeline, 25% ReadHomeTimeline, and 5% FollowUser requests.

HipsterShop itself does not implement data storage, and we modify it to use MongoDB for saving orders and Redis for shopping carts. We also add Redis instances for caching product and ad lists.

[2] Go partially supports dynamic code loading via a plugin [39], but it requires the plugin and the loader be compiled with a same version of the Go toolchain, and all their dependency libraries have exactly the same versions.

[3] Node.js supports worker threads [56] for running CPU-intensive tasks, but they have worse performance for I/O-intensive tasks.

Table 2: Microservice workloads in evaluation (from [29, 70])

	Ported services	RPC framework	Languages
SocialNetwork	11	Thrift [9]	C++
MovieReviewing	12	Thrift	C++
HotelReservation	11	gRPC [30]	Go
HipsterShop	13	gRPC	Go, Node.js, Python

Table 3: Percentage of internal function calls (§ 5.1).

SocialNetwork		Movie	Hotel	Hipster
write	mixed	Reviewing	Reservation	Shop
66.7%	62.3%	69.2%	79.2%	85.1%

HipsterShop includes two services written in Java and C#, which are not languages supported by Nightcore. Thus we re-implement their functionality in Go and Node.js.

For each benchmark workload, we port their stateless mid-tier services to Nightcore, as well as OpenFaaS [37] for comparison. For other stateful services (e.g., database, Redis, NGINX, etc.), we run them on dedicated VMs with sufficiently large resources to ensure they are not bottlenecks. For Nightcore and OpenFaaS, their API gateways also run on a separate VM. This configuration favors OpenFaaS because its gateway is used for both external and internal function calls and is therefore more heavily loaded than Nightcore's gateway.

We use `wrk2` [26] as the load generator for all workloads. In our experiments, the target input load (queries per second (QPS)) is run for 180 seconds, where the first 30 seconds are used for warming up the system, and 50th and 99th percentile latencies of the next 150 seconds are reported. Following this methodology, variances of measured latencies are well within 10% before reaching the saturation throughput.

Internal Function Calls. One major design decision in Nightcore is to optimize internal function calls (§3), so we quantify the percentage of internal function calls in our workloads in Table 3. The results show that internal function calls dominate external calls, sometimes by more than a factor of 5×.

Cold-Start Latencies. There are two components of cold-start latencies in a FaaS system. The first arises from provisioning a function container. Our prototype of Nightcore relies on unmodified Docker, thus does not include optimizations. However, state-of-the-art techniques such as Catalyzer [67] achieve startup latencies of 1–14ms. These techniques can be applied to Nightcore as they become mainstream. The second component of cold-start latency is provisioning the FaaS runtime inside the container. Our measurement shows that Nightcore's function worker process takes only 0.8ms to be ready for executing user-provided function code.

Systems for Comparison. We compare Nightcore with two other systems: (1) RPC servers running in Docker containers which are originally used for implementing stateless microservices in the evaluation workloads; (2) OpenFaaS [37], a popular open source FaaS system, where we use the OpenFaaS watchdog [51] in HTTP mode to implement function handlers, and Docker for running function containers.

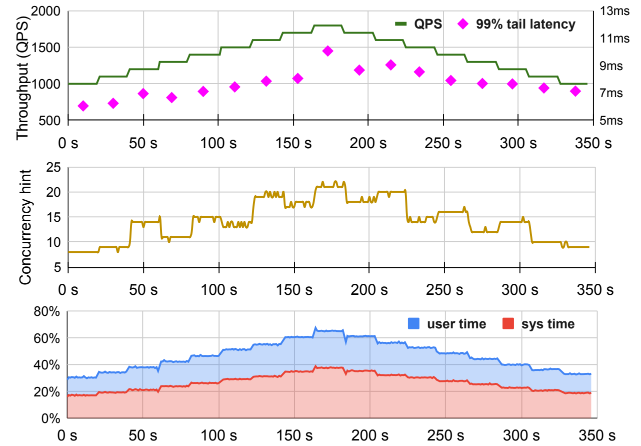


Figure 6: Nightcore running SocialNetwork (write) with load variations. The upper chart shows tail latencies under changing QPS, the middle chart shows how the concurrency hint (τ_k) of the post microservice changes with time, and the lower chart is a timeline of CPU utilization.

We also tested the SocialNetwork application on AWS Lambda. Even when running with a light input load and with provisioned concurrency, Lambda cannot meet our latency targets. Executing the "mixed" load pattern shows median and 99% latencies are 26.94ms and 160.77ms, while they are 2.34ms and 6.48ms for containerized RPC servers. These results are not surprising given the measurements in Table 1.

5.2 Benchmarks

Single Worker Server. We start with evaluating Nightcore with one worker server. All systems use one `c5.2xlarge` EC2 VM, which has 8 vCPUs and 16GiB of memory. For Nightcore and OpenFaaS, this VM is its single worker server, that executes all serverless functions. On the worker VM, Nightcore's engine uses two I/O threads. For RPC servers, this VM runs all ported stateless microservices, such that all inter-service RPCs are local.

Figure 7 demonstrates experimental results on all workloads. For all workloads, results show the trend that OpenFaaS' performance is dominated by containerized RPC servers, while Nightcore is superior to those RPC servers. OpenFaaS' performance trails the RPC servers because all inter-service RPCs flow through OpenFaaS's gateway and watchdogs, which adds significant latency and CPU processing overheads. On the other hand, Nightcore's performance is much better than OpenFaaS, because Nightcore optimizes the gateway out of most inter-service RPCs, and its event-driven engine handles internal function calls with microsecond-scale overheads.

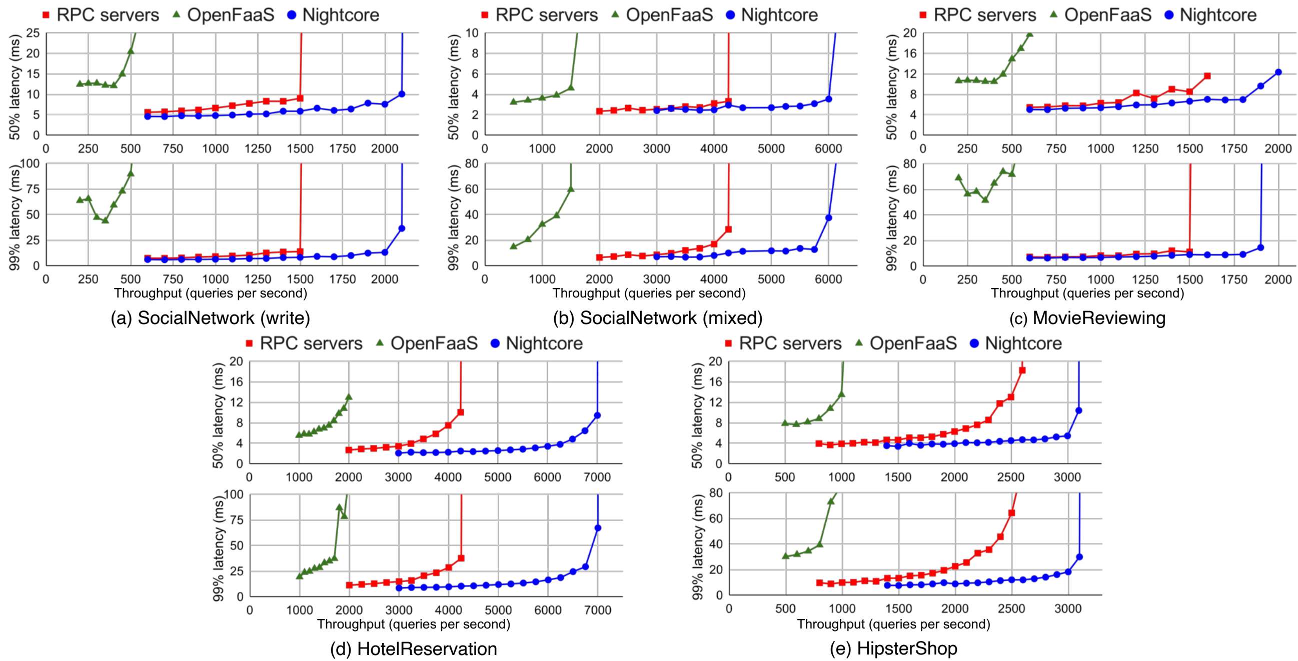


Figure 7: Comparison of Nightcore with RPC servers and OpenFaaS using one VM.

Table 4: Evaluation of Nightcore's scalability, where n worker servers run n times the base QPS input. For each workload, the base QPS is selected to be close to the saturation throughput when using one server.

	Base QPS	Median latency (ms)				99% tail latency (ms)			
		1 server	2 server	4 server	8 server	1 server	2 server	4 server	8 server
SocialNetwork (mixed)	2000	3.40	2.64	2.39	2.64	10.93	8.36	7.18	8.07
	2300	3.37	2.65	2.43	2.61	13.95	10.34	8.20	10.63
MovieReviewing	800	7.24	7.93	7.35	8.10	9.26	11.42	10.97	16.31
	850	7.24	7.54	7.57	8.57	9.31	11.18	12.24	25.01
HotelReservation	3000	3.48	3.29	3.08	4.32	18.27	15.98	14.98	18.09
	3300	5.56	4.43	5.50	4.43	31.92	22.66	22.54	20.83
HipsterShop	1400	6.05	5.70	6.23	5.68	19.68	17.42	19.10	15.02
	1500	7.95	7.51	8.32	7.06	25.39	23.74	23.81	20.53

Table 5: Comparison of Nightcore with other systems using 8 VMs. Median and 99% tail latencies are shown in milliseconds. For each workload, the saturation throughput of the RPC servers is the baseline QPS (1.00x in the table) for comparison.

	SocialNetwork (mixed)			MovieReviewing			HotelReservation			HipsterShop		
	QPS	median	tail	QPS	median	tail	QPS	median	tail	QPS	median	tail
RPC servers	1.00x	3.21	23.98	1.00x	14.45	25.57	1.00x	5.54	19.73	1.00x	10.68	48.13
	1.17x	110.01	>1000	1.20x	30.80	>1000	1.22x	10.43	43.46	1.17x	15.61	80.89
OpenFaaS	0.29x	4.57	81.60	0.30x	10.06	113.47	0.28x	5.80	18.96	0.29x	9.29	32.13
	0.33x	6.72	368.38	0.40x	13.32	>1000	0.33x	16.21	103.81	0.38x	24.93	86.59
Nightcore	1.33x	2.64	8.07	1.28x	8.10	16.31	2.67x	4.32	18.09	1.87x	5.68	15.02
	1.53x	2.61	10.63	1.36x	8.57	25.01	2.93x	4.43	20.83	2.00x	7.06	20.53

Compared to RPC servers, Nightcore achieves 1.27× to 1.59× higher throughput and up to 34% reduction in tail latencies, showing that Nightcore has a lower overhead for inter-service communications than RPC servers, which we will discuss more in § 5.3.

We also tested Nightcore under variable load to demonstrate how its ability to manage concurrency (§ 3.3) adapts to changing loads. Figure 6 shows the results with a corresponding timeline of CPU utilization, indicating that Nightcore can promptly change

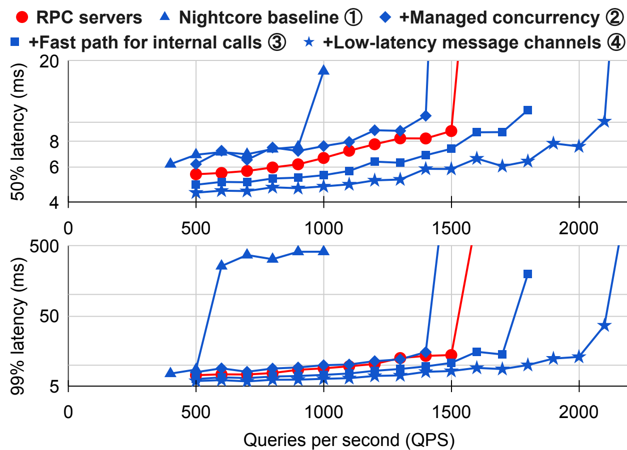


Figure 8: Comparison of Nightcore with RPC servers, running SocialNetwork (write) using one VM. Designs of Nightcore are progressively added to show their effectiveness (§ 5.3).

Table 6: Breakdowns of stacktrace samples, when running SocialNetwork (write) at 1200 QPS on one VM. Unix sockets are used by Thrift RPC servers for inter-thread synchronizations.

	RPC servers	Nightcore
do_idle	41.6%	60.4%
user space	18.3%	14.8%
irq / softirq		
– netrx	7.1%	6.8%
– others	2.0%	1.6%
syscall		
– tcp socket	20.7%	7.6%
– poll / epoll	2.5%	1.1%
– futex	2.2%	0.1%
– pipe	0%	3.7%
– unix socket	1.1%	0%
– others	3.1%	3.1%
uncategorized	1.4%	0.8%

its concurrency level under increasing loads. When the input load reaches its maximum (of 1800 QPS), the tail latency also reaches its maximum (of 10.07 ms).

Multiple Worker Servers. We also evaluate Nightcore in a distributed setting, where multiple VMs are used as Nightcore's worker servers. We use `c5.xlarge` EC2 instances for worker VMs, which have 4 vCPUs and 8GiB of memory.

Table 4 evaluates Nightcore's scalability, where up to 8 VMs are used for worker servers and the input load is scaled with the number of VMs. The similar (or decreasing) median and tail latencies show that Nightcore's scalability is nearly linear–e.g., increasing the input load 8× and providing 8 servers does not change median and tail latencies significantly. The only exception is MovieReviewing, where the tail latency of running on 8 server is 2.7× worse than 1 server. However, we observe that scaling this workload with RPC servers also suffers increased tail latencies.

Next we compare RPC servers and OpenFaaS to Nightcore with 8 worker VMs. For RPC servers, 8 VMs run stateless services, where each VM runs one replica of each service and load balancing is transparently supported by RPC client libraries. For OpenFaaS and Nightcore, 8 VMs run their function handlers. Table 5 summarizes the experimental results, demonstrating that Nightcore achieves 1.36× to 2.93× higher throughput and up to 69% reduction in tail latency than an RPC server (while OpenFaaS consistently underperforms an RPC server). The advantage of Nightcore over RPC servers is more significant in the distributed setting, because there are inter-host RPCs in the case of replicated RPC servers, while there is no network traffic among Nightcore's worker VMs.

5.3 Analysis

Evaluating Nightcore's Design. We quantify the value of Nightcore's design decisions in Figure 8. The figure shows the performance effect of adding managed concurrency for function executions (§ 3.3), a fast path for internal function call (§ 3.2), and low-latency message channels as IPC primitives (§ 3.1). The Nightcore baseline ① maximizes concurrent function executions (i.e., concurrency management is disabled), all internal function calls go through the gateway, and Nightcore's message channels are replaced with TCP sockets. This baseline Nightcore design can achieve only one third the throughput of RPC servers while meeting tail latency targets (①). When we add managed concurrency (②), Nightcore's performance comes close to RPC servers, as tail latencies are significantly improved. Optimizing the gateway out of the processing path for internal function calls (③) brings Nightcore's performance above the RPC servers. Finally, Nightcore's low-latency message channels boost performance further (④), resulting in 1.33× higher throughput than RPC servers.

Communication Overheads. Microservice-based applications are known to have a high communication-to-computation ratio [70, 83, 101]. When using RPC servers to implement microservices and running them in containers, inter-service RPCs pass through network sockets virtualized by the container runtime, via overlay networks [113]. Container overlay networks allow a set of related containers running on multiple host machines to use independent IP addresses, without knowing if other containers reside on the same host. While this approach works for the general case, even containers on the same host pay the processing costs of the full network stack.

On the other hand, Nightcore keeps most inter-service calls on the same host and uses Linux pipes for intra-host communications. Eliminating most inter-host networking explains Nightcore's performance advantage over containerized RPC servers in the distributed setting. But Nightcore also has a noticeable advantage over containerized RPC servers for intra-host communications, shown in Figure 7. To further understand this advantage, we collect stacktrace samples for both Nightcore and containerized RPC servers running with a single VM, and Table 6 summarizes the results. For RPC servers, TCP-related system calls and netrx softirq consume 47.6%

of non-idle CPU time, both of which are used for inter-service communications. In contrast, Nightcore spends much less CPU time in TCP-related system calls, because only communication with services running on other hosts (e.g., database and Redis) uses TCP sockets. Both systems spend roughly the same amount of CPU time in netrx softirqs, which is caused only by inter-host networking.

5.4 Discussion

A goal for Nightcore is to avoid modifying Linux, because we want Nightcore to be easier to adopt for existing microservice workloads. Nightcore therefore relies on existing OS abstractions to achieve its performance goals, creating a challenge to efficiently use the operating systems' existing I/O abstractions and to find individual "killer microseconds."

In our experience with Nightcore, we find there is no single dominant "killer microsecond." There are multiple factors with significant contributions, and all must be addressed. Profiling the whole system for microsecond-scale optimization opportunities is challenging given the overheads introduced by profiling itself. In Nightcore, we implement low-overhead statistics collectors, and use eBPF programs [16] for kernel-related profiling.

6 RELATED WORK

Microservices. The emergence of microservices for building large-scale cloud applications has prompted recent research on characterizing their workloads [70, 99, 102, 112], as well as studying their hardware-software implications [70, 99, 100]. Microservices have a higher communication-to-computation ratio than traditional workloads [70] and frequent microsecond-scale RPCs, so prior work has studied various software and hardware optimization opportunities for microsecond-scale RPCs, including transport layer protocols [78, 79], a taxonomy of threading models [100], heterogeneous NIC hardware [85], data transformations [93], and CPU memory controllers [101]. The programming model of serverless functions maps inter-service RPCs to internal function calls, allowing Nightcore to avoid inter-host networking and transparently eliminate RPC protocol overheads. X-Containers [97] is a recently proposed LibOS-based container runtime, that improves the efficiency of inter-service communications for mutually trusting microservices. For comparison, Nightcore still relies on the current container mechanism (provided by Docker), which does not require microservices to trust each other.

Serverless Computing. Recent research on serverless computing has mostly focused on data intensive workloads [59, 65, 68, 69, 75, 95, 98], leading invocation latency overheads to be largely overlooked. SAND [57] features a local message bus as the fast path for chained function invocations. However, SAND only allows a single, local call at the end of a function, while Nightcore supports arbitrary calling patterns (e.g., Figure 1). Faasm [98]'s chained function calls have the same functionality as Nightcore's internal function calls, but they are executed within the same process, relying on WebAssembly for software-based fault isolation. One previous work [62] also notices that FaaS systems have to achieve microsecond-scale overheads for efficient support of microservices, but they demonstrate only a proof-of-concept FaaS runtime that relies on Rust's memory safety for isolation and lacks end-to-end evaluations on realistic microservices.

System Supports for Microsecond-Scale I/Os. Prior work on achieving microsecond-scale I/O has been spread across various system components, ranging from optimizing the network stack [74, 78, 79]; designs for a dataplane OS [61, 77, 87, 91, 92, 94]; thread scheduling for microsecond-scale tasks [63, 77, 91, 96, 100]; and novel filesystems leveraging persistent memory [81, 86, 108]. Additionally, the efficiency of I/O is also affected by the user-facing programming model [72, 105] and the underlying mechanism for concurrency [80, 103]. A recent paper from Google [60] argues that current systems are not tuned for microsecond-scale events, as various OS building blocks have microsecond-scale overheads. Eliminating these overheads requires a tedious hunt for the "killer microseconds." Inspired by this work, the design of Nightcore eliminates many of these overheads, making it practical for a microsecond-scale serverless system.

7 CONCLUSION

Optimizing Nightcore justifies one of Lampson's early hints [82]: "make it fast, rather than general or powerful", because fast building blocks can be used more widely. As computing becomes more granular [84], we anticipate more microsecond-scale applications will come to serverless computing. Designing and building this next generation of services will require careful attention to microsecond-scale overheads.

ACKNOWLEDGEMENTS

We thank our shepherd Christina Delimitrou and the anonymous reviewers for their insightful feedback. We also thank Zhiting Zhu, Christopher J. Rossbach, James Bornholt, Zhan Shi, and Wei Sun for their valuable comments on the early draft of this work. This work is supported in part by NSF grants CNS-2008321 and NSF CNS-1900457, and the Texas Systems Research Consortium.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact includes the prototype implementation of Nightcore, the DeathStarBench [21] and HipsterShop microservices [29] ported to Nightcore, and the experiment workflow to run these workloads on AWS EC2 instances.

A.2 Artifact Check-List (Meta-Information)

- **Program: Nightcore, Docker runtime, and `wrk2`**
- **Run-time environment: AWS EC2 instances**
- **Metrics: Latency and throughput**
- **Experiments: Our ports of DeathStarBench [21] and HipsterShop microservices [29] (included in this artifact)**
- **Disk space required: 2GB**
- **Time needed to prepare workflow: 1 hour**
- **Time needed to complete experiments: 3 hours**
- **Publicly available: Yes**
- **Code licenses: Apache License 2.0**
- **Archive: `10.5281/zenodo.4321760`**

A.3 Description

How to Access. The source code and benchmarks are host on GitHub `ut-osa/nightcore` and `ut-osa/nightcore-benchmarks`.

Hardware Dependencies. This artifact runs on AWS EC2 instances in `us-east-2` region.

Software Dependencies. This artifact requires experiment VMs running Ubuntu 20.04 with Docker installed.

We provide a pre-built VM image hosted on AWS `us-east-2` region (`ami-06e206d7334bff2ec`) with all necessary dependencies installed, which is used by experiment scripts in this artifact.

A.4 Installation

Setting up the Controller Machine. A controller machine in AWS `us-east-2` region is required for running scripts executing experiment workflows. The controller machine can use very small EC2 instance type, as it only provisions and controls experiment VMs, but does not affect experimental results. In our own setup, we use a `t3.micro` EC2 instance installed with Ubuntu 20.04 as the controller machine.

The controller machine needs `python3`, `rsync`, and AWS CLI version 1 installed. `python3` and `rsync` can be installed with apt. This documentation details the recommended way for installing AWS CLI version 1. Once installed, AWS CLI has to be configured with region `us-east-2` and access key (see this).

Then on the controller machine, clone our artifact repository with all git submodules:

```
git clone --recursive \
    https://github.com/ut-osa/nightcore-benchmarks.git
```

Finally, execute `scripts/setup_sshkey.sh` to setup SSH keys that will be used to access experiment VMs. Please read the notice in the script before executing it to check if this script works for your setup.

Setting up EC2 Security Group and Placement Group. Our VM provisioning script creates EC2 instances with security group `nightcore` and placement group `nightcore-experiments`. The security group includes firewall rules for experiment VMs (i.e., allowing the controller machine to SSH into them), while the placement group instructs AWS to place experiment VMs close together.

Executing `scripts/aws_provision.sh` on the controller machine creates these groups with correct configurations.

Building Docker Images. We also provide the script (`scripts/docker_images.sh`) for building Docker images relevant to experiments in this artifact. As we already pushed all compiled images to DockerHub, there is no need to run this script as long as you do not modify source code of Nightcore (in `nightcore` directory) and evaluation workloads (in `workloads` directory).

A.5 Experiment Workflow

Each sub-directory within `experiments` corresponds to one experiment. Within each experiment directory, a `config.json` file describes machine configuration and placement assignment of individual Docker containers (i.e. microservices) for this experiment.

The entry point of each experiment is the `run_all.sh` script. It first provisions VMs for experiments. Then it executes evaluation workloads with different QPS targets via `run_once.sh` script. `run_once.sh` script performs workload-specific setups, runs `wrk2` to measure latency distribution under the target QPS, and stores results in `results` directory. When everything is done, `run_all.sh` script terminates all provisioned experiment VMs.

VM provisioning is done by `scripts/exp_helper` with subcommand `start-machines`. By default, it creates on-demand EC2 instances. But it also supports the option to use Spot instances for cost saving. After EC2 instances are up, the script then sets up Docker engines on newly created VMs to form a Docker cluster in swarm mode.

A.6 Evaluation and Expected Result

For each experiment, the evaluation metric is the latency distribution under a specific QPS. We use `wrk2` as the benchmarking tool, and it outputs a detailed latency distribution, which looks like

```
 Latency Distribution (HdrHistogram - Recorded Latency)
 50.000%    2.21ms
 75.000%    3.29ms
 90.000%    5.13ms
 99.000%    9.12ms
 99.900%   12.28ms
 99.990%   17.45ms
 99.999%   20.32ms
100.000%   23.61ms
```

We report the 50% and 99% percentile values as median and tail latencies in the paper. `run_all.sh` script conducts evaluations on various QPS targets.

Experiment sub-directories ending with "`singlenode`" correspond to Nightcore results in Figure 7 of the main paper. Experiment sub-directories ending with "`4node`" correspond to Nightcore (4 servers) results in Table 4 of the main paper. Note that `run_all.sh` scripts run less data points than presented in the paper, to allow a fast validation. But all `run_all.sh` scripts can be easily modified to collect more data points.

We provide a helper script "`scripts/collect_results`" to print a summary of all experiment results. Meanwhile, "`expected_results_summary.txt`" gives the summary generated from our experiment runs. Details of our runs are stored in the "`expected_results`" directory within each experiment subdirectory. Note that these results are not the exact ones presented in the paper.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] [n.d.]. 4 Microservices Examples: Amazon, Netflix, Uber, and Etsy. https://blog.dreamfactory.com/microservices-examples/ [Accessed Jan, 2021].

[2] [n.d.]. Accessing Amazon CloudWatch logs for AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/monitoring-cloudwatchlogs.html [Accessed Dec, 2020].

[3] [n.d.]. Addressing Cascading Failures. https://landing.google.com/sre/sre-book/chapters/addressing-cascading-failures/ [Accessed Jan, 2021].

[4] [n.d.]. Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/ [Accessed Jan, 2021].

[5] [n.d.]. Airbnb's 10 Takeaways from Moving to Microservices. https://thenewstack.io/airbnbs-10-takeaways-moving-microservices/ [Accessed Jan, 2021].

[6] [n.d.]. Amazon DynamoDB | NoSQL Key-Value Database | Amazon Web Services. https://aws.amazon.com/dynamodb/ [Accessed Jan, 2021].

[7] [n.d.]. Amazon ElastiCache- In-memory data store and cache. https://aws.amazon.com/elasticache/ [Accessed Jan, 2021].

[8] [n.d.]. Announcing WebSocket APIs in Amazon API Gateway. https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/ [Accessed Dec, 2020].

[9] [n.d.]. Apache Thrift - Home. https://thrift.apache.org/ [Accessed Jan, 2021].

[10] [n.d.]. Architecture: Scalable commerce workloads using microservices. https://cloud.google.com/solutions/architecture/scaling-commerce-workloads-architecture [Accessed Jan, 2021].

[11] [n.d.]. asyncio — Asynchronous I/O. https://docs.python.org/3.8/library/asyncio.html [Accessed Jan, 2021].

[12] [n.d.]. AWS Fargate - Run containers without having to manage servers or clusters. https://aws.amazon.com/fargate/ [Accessed Jan, 2021].

[13] [n.d.]. AWS Lambda execution context - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html [Accessed Jan, 2021].

[14] [n.d.]. AWS Lambda FAQs. https://aws.amazon.com/lambda/faqs/ [Accessed Jan, 2021].

[15] [n.d.]. AWS Lambda – Serverless Compute - Amazon Web Servicesy. https://aws.amazon.com/lambda/ [Accessed Jan, 2021].

[16] [n.d.]. BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc [Accessed Jan, 2021].

[17] [n.d.]. Best practices for working with AWS Lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html [Accessed Dec, 2020].

[18] [n.d.]. Building serverless microservices in Azure - sample architecture. https://azure.microsoft.com/is-is/blog/building-serverless-microservices-in-azure-sample-architecture/ [Accessed Jan, 2021].

[19] [n.d.]. Cloud Object Storage | Store and Retrieve Data Anywhere | Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/ [Accessed Jan, 2021].

[20] [n.d.]. Coursera Case Study. https://aws.amazon.com/solutions/case-studies/coursera-ecs/ [Accessed Jan, 2021].

[21] [n.d.]. delimitrou/DeathStarBench: Open-source benchmark suite for cloud microservices. https://github.com/delimitrou/DeathStarBench [Accessed Jan, 2021].

[22] [n.d.]. Enough with the microservices. https://adamdrake.com/enough-with-the-microservices.html [Accessed Jan, 2021].

[23] [n.d.]. Event-based Concurrency (Advanced). http://pages.cs.wisc.edu/~remzi/OSTEP/threads-events.pdf [Accessed Jan, 2021].

[24] [n.d.]. eventfd(2) - Linux manual page. https://man7.org/linux/man-pages/man2/eventfd.2.html [Accessed Jan, 2021].

[25] [n.d.]. firecracker/network-performance.md at master · firecracker-microvm/firecracker. https://github.com/firecracker-microvm/firecracker/blob/master/docs/network-performance.md [Accessed Jan, 2021].

[26] [n.d.]. giltene/wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2 [Accessed Jan, 2021].

[27] [n.d.]. Go, don't collect my garbage. https://blog.cloudflare.com/go-dont-collect-my-garbage/ [Accessed Jan, 2021].

[28] [n.d.]. Go memory ballast: How I learnt to stop worrying and love the heap. https://blog.twitch.tv/en/2019/04/10/go-memory-ballast-how-i-learnt-to-stop-worrying-and-love-the-heap-26c2462549a2/ [Accessed Jan, 2021].

[29] [n.d.]. GoogleCloudPlatform/microservices-demo. https://github.com/GoogleCloudPlatform/microservices-demo [Accessed Jan, 2021].

[30] [n.d.]. gRPC - A high-performance, open source universal RPC framework. https://grpc.io/ [Accessed Jan, 2021].

[31] [n.d.]. IPC settings | Docker run reference. https://docs.docker.com/engine/reference/run/#ipc-settings---ipc [Accessed Jan, 2021].

[32] [n.d.]. libuv | Cross-platform asynchronous I/O. https://libuv.org/ [Accessed Jan, 2021].

[33] [n.d.]. Lyft Case Study. https://aws.amazon.com/solutions/case-studies/lyft/ [Accessed Jan, 2021].

[34] [n.d.]. Manage your function app. https://docs.microsoft.com/en-us/azure/azure-functions/functions-how-to-use-azure-function-app-settings [Accessed Jan, 2021].

[35] [n.d.]. Microservice Trade-Offs. https://martinfowler.com/articles/microservice-trade-offs.html [Accessed Jan, 2021].

[36] [n.d.]. Microservices - Wikipedia. https://en.wikipedia.org/wiki/Microservices [Accessed Jan, 2021].

[37] [n.d.]. OpenFaaS | Serverless Functions, Made Simple. https://www.openfaas.com/ [Accessed Jan, 2021].

[38] [n.d.]. Performance Under Load. https://medium.com/@NetflixTechBlog/performance-under-load-3e6fa9a60581 [Accessed Jan, 2021].

[39] [n.d.]. plugin - The Go Programming Language. https://golang.org/pkg/plugin/ [Accessed Jan, 2021].

[40] [n.d.]. Provisioned Concurrency for Lambda Functions. https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions/ [Accessed Jan, 2021].

[41] [n.d.]. Remind Case Study. https://aws.amazon.com/solutions/case-studies/remind/ [Accessed Jan, 2021].

[42] [n.d.]. Rewriting Uber Engineering: The Opportunities Microservices Provide. https://eng.uber.com/building-tincup-microservice-implementation/ [Accessed Jan, 2021].

[43] [n.d.]. Serverless and Microservices: a match made in heaven? https://pauldjohnston.medium.com/serverless-and-microservices-a-match-made-in-heaven-9964f329a3bc [Accessed Dec, 2020].

[44] [n.d.]. Serverless Microservices - Microservices on AWS. https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/serverless-microservices.html [Accessed Jan, 2021].

[45] [n.d.]. Serverless Microservices reference architecture. https://docs.microsoft.com/en-us/samples/azure-samples/serverless-microservices-reference-architecture/serverless-microservices-reference-architecture/ [Accessed Dec, 2020].

[46] [n.d.]. shm_overview(7) — Linux manual page. https://man7.org/linux/man-pages/man7/shm_overview.7.html [Accessed Jan, 2021].

[47] [n.d.]. Splitting Up a Codebase into Microservices and Artifacts. https://engineering.linkedin.com/blog/2016/02/q-a-with-jim-brikman--splitting-up-a-codebase-into-microservices [Accessed Jan, 2021].

[48] [n.d.]. "Stop Rate Limiting! Capacity Management Done Right" by Jon Moore. https://www.youtube.com/watch?v=m64SWl9bfvk [Accessed Jan, 2021].

[49] [n.d.]. Thoughts on (micro)services. https://luminousmen.com/post/thoughts-on-microservices [Accessed Jan, 2021].

[50] [n.d.]. Uncovering the magic: How serverless platforms really work! https://medium.com/openwhisk/uncovering-the-magic-how-serverless-platforms-really-work-3cb127b05f71 [Accessed Jan, 2021].

[51] [n.d.]. Watchdog - OpenFaaS. https://docs.openfaas.com/architecture/watchdog/ [Accessed Jan, 2021].

[52] [n.d.]. What are Microservices? | AWS. https://aws.amazon.com/microservices/ [Accessed Jan, 2021].

[53] [n.d.]. What is a serverless microservice? | Serverless microservices explained. https://www.cloudflare.com/learning/serverless/glossary/serverless-microservice/ [Accessed Dec, 2020].

[54] [n.d.]. Why should you use microservices and containers? https://developer.ibm.com/technologies/microservices/articles/why-should-we-use-microservices-and-containers/ [Accessed Jan, 2021].

[55] [n.d.]. Why so slow? - Binaris Blog. https://blog.binaris.com/why-so-slow/ [Accessed Jan, 2021].

[56] [n.d.]. Worker threads | Node.js v14.8.0 Documentation. https://nodejs.org/api/worker_threads.html [Accessed Jan, 2021].

[57] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[58] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1851182.1851192

[59] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[60] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (March 2017), 48–54. https://doi.org/10.1145/3015146

[61] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[62] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 645–650.

[63] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2020. Lightweight Preemptible Functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 465–477. https://www.usenix.org/conference/atc20/presentation/boucher

[64] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[65] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3357223.3362711

[66] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[67] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[68] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[69] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[70] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[71] Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Barnard Metzler. 2020. Serverless End Game: Disaggregation enabling Transparency. *arXiv preprint arXiv:2006.01251* (2020).

[72] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 135–148.

[73] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 519–531.

[74] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 489–502.

[75] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[76] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[77] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µSecond-Scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 345–359.

[78] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs Can Be General and Fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 1–16.

[79] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 863–880. https://www.usenix.org/conference/atc19/presentation/kogias-r2p2

[80] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. 2007. Events Can Make Sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA) *(ATC'07)*. USENIX Association, USA, Article 7, 14 pages.

[81] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

[82] Butler W. Lampson. 1983. Hints for Computer System Design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* (Bretton Woods, New Hampshire, USA) *(SOSP '83)*. Association for Computing Machinery, New York, NY, USA, 33–48. https://doi.org/10.1145/800217.806614

[83] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou. 2020. Dagger: Towards Efficient RPCs in Cloud Microservices With Near-Memory Reconfigurable NICs. *IEEE Computer Architecture Letters* 19, 2 (2020), 134–138. https://doi.org/10.1109/LCA.2020.3020064

[84] Collin Lee and John Ousterhout. 2019. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 149–154. https://doi.org/10.1145/3317550.3321447

[85] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 363–378. https://www.usenix.org/conference/atc19/presentation/liu-ming

[86] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[87] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 399–413. https://doi.org/10.1145/3341301.3359657

[88] Ben Maurer. 2015. Fail at Scale: Reliability in the Face of Rapid Change. *Queue* 13, 8 (Sept. 2015), 30–46. https://doi.org/10.1145/2838344.2839461

[89] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[90] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[91] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 361–377.

[92] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[93] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1203–1216. https://doi.org/10.1145/3373376.3378501

[94] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[95] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[96] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 145–160.

[97] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 121–135. https://doi.org/10.1145/3297858.3304016

[98] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[99] A. Sriraman and T. F. Wenisch. 2018. μSuite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 1–12.

[100] Akshitha Sriraman and Thomas F. Wenisch. 2018. µTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 177–194. https://www.usenix.org/conference/osdi18/presentation/sriraman

[101] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis. 2020. The NEBULA RPC-Optimized Architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 199–212.

[102] T. Ueda, T. Nakaike, and M. Ohara. 2016. Workload characterization for microservices. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10.

[103] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 268–281. https://doi.org/10.1145/945445.945471

[104] Matt Welsh and David Culler. 2003. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Seattle, WA) *(USITS'03)*. USENIX Association, USA, 4.

[105] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 230–243. https://doi.org/10.1145/502059.502057

[106] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/2018436.2018443

[107] Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, Haibing Guan, Sanhong Li, Chuansheng Lu, and Tongbao Zhang. 2020. Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 159–172. https://www.usenix.org/conference/atc20/presentation/wu-mingyu

[108] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[109] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 379–391. https://doi.org/10.1145/2465351.2465388

[110] Y. Zhang, D. Meisner, J. Mars, and L. Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 456–468.

[111] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 149–161. https://doi.org/10.1145/3267809.3267823

[112] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking Microservice Systems for Software Engineering Research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 323–324. https://doi.org/10.1145/3183440.3194991

[113] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 331–344.