PYROSOME: A RUNTIME FOR EFFICIENT AND FINE-GRAINED MICROSERVICES AND SERVERLESS

Anonymous authors
Paper under double-blind review

Abstract

Developers have shifted from deploying applications on physical machines, to virtual machines, to containers, and now to serverless functions. Such shift of abstractions have also changed the way applications are structured. Today's cloud-native applications are naturally structured in a higher-level and more decomposed way. However, today's cloud and serverless platforms are still layered on top of the same inefficient legacy software infrastructure and abstractions as in the past. We argue that these legacy layers are now redundant, and we explore a clean-slate cloud services runtime targeted toward microservice- and serverless-era applications we call Pyrosome.

Pyrosome provides simple programming interfaces for executing code, storing and sharing data, and low-overhead communication without worrying about resource allocation and scheduling. It leverages low-overhead language-based sandboxing that avoids the full state and scheduling costs of operating system processes or containers. This allows implementation of a holistic scheduler that quickly redistributes load among cores, exploits parallelism in applications, and avoids tail latency with execution-time-aware sharding. The DeathStarBench microservices benchmarks show that Pyrosome speeds up microservice applications by as much as 4× and improves throughput by 10×. Additionally, we show Pyrosome balances load nearly instantly compared to standard microservice platforms.

1 Introduction

Over the last decade, cloud computing platforms have evolved from lower-level toward higher-level abstractions, from machine-level (virtual machines), to operating system-level (containers) and even higher-level programming abstractions like serverless [17, 23]. The evolution of cloud computing technologies and the need for efficient software development and maintenance has also led to fundamental changes in designing and deploying cloud applications, resulting in new cloud application paradigms such as microservices and serverless. There are two dimensions of paradigm shift for cloud applications. First, applications are leveraging high-level cloud-native abstractions, which frees developers from system-level resource management, scheduling and orchestration. Second, applications are being decomposed into finer granularity components. Decomposing applications into small services with well-defined interfaces, as with the microservies architecure, allows each service to be developed, maintained and scaled independently while matching organizational structures, which improves developer productivity especially for large-scale applications [13]. Some organizations have gone further to make services even smaller (e.g. the BBC's nanoservices platform [12]), with the expectation that smaller services limit the impact of failures, allow for more rapid iteration, and support flexible resource sharing.

However, today's cloud platforms, as the legacy from a decade of evolution of cloud technologies, is comprised of layers of software infrastructure and abstractions that are redundant and inefficient for fine-grained microservies and serverless applications. For example, the containerization layer, on top of which most of today's microservices and serverless platforms are built, adds additional isolation and communication costs. Microservices isolated in containers communicate with each other over the network, which can cost as much as one third of the total execution time [19]. The added overheads offset some of the benefits of microservices architecture, which means the benefits of microservices only outweigh the additional overheads for large complex applications. Smaller applications (especially ones composed of smaller microservices) are still better off implemented as monoliths to avoid these overheads. The high cost of cold starts (the process of creating and setting up new containers when capacity is under-provisioned) slows down resource reprovisioning and can impact the availability of serverless applications during scaling up. As a result, the instant scalability promise of serverless computing cannot be fully fulfilled either.

It's becoming a very compelling and real problem of microservices and serverless application deployment in the industry. For example, the Prime Video application was previously implemented as a distributed microservices architecture, but was forced to abandon the microservices architecture due to high costs and scaling bottlenecks. They moved back to the monolith architecture and as a result reduced costs by 90% [27].

A deep-rooted mismatch is that virtual machines and containers implement strong isolation to mitigate risks between untrusted users; however, this security is unnecessary for services from a single large application where only basic inter-service fault-isolation is required. In addition, many microservices and serverless applications are taking a cloud-

native approach that relies on high-level abstractions and interfaces provided by cloud platforms, and expect cloud platforms to hide the complexities of underlying hardware and software infrastructure. Thus the virtualized operating system abstraction provided by containerization is no longer needed.

In light of this, we propose a clean-slate design for a microservices deployment system that we call Pyrosome. We leverage the fact that in the following common use cases strong security isolation is not needed, to design Pyrosome as a PoC of a runtime with minimal isolation costs between mutually trusted microservices/functions and applications:

- Microservices and serverless functions of the same application should be mutually trusted, so there's no need for strong security isolation between them.
- Many microservices and serverless applications are actually deployed in trusted environment, on private infrastructure and private cloud computing platform, within which security is less of a concern.
- On public cloud, mutually trusted applications (e.g. applications from the same client) can share the same security sandbox (e.g. a VM) within which they don't need strong security isolation between each other. So Pyrosome runtime can be deployed inside such a security sandbox to support a group of mutually trusted applications.

Two key aspects of Pyrosome's design lead to these benefits. The first is that Pyrosome leverages lightweight language-level code isolation using software (V8) sandboxes and an in-process data cache. This lowers cross-sandbox/cross-service communication and data access costs. Additionally, since isolation and communication is low overhead, Pyrosome is able to support microservices of much finer granularity than today's standard container-based approach. It also ensures that Pyrosome can keep enough inexpensive sandboxes ready to eliminate the need for costly *cold starts* for container creation, so Pyrosome can instantly reprovision resources to handle sudden load changes. The second key aspect of Pyrosome's design demonstrates that in radically redesigning the cloud software infrastructure, cloud platforms can find opportunities for advanced optimizations for today's cloud-native applications. With low-overhead isolation, each service operation on Pyrosome can complete in microseconds or less. Thus, its scheduler works on a much finer timescale than conventional microservice resource schedulers (e.g. Kubernetes). With this in mind, Pyrosome's scheduler is designed to support several new optimizations that are largely inspired by recently fine-grained dispatch policies minimizing tail latency in simpler services like key-value stores [7, 14, 25, 32, 35]. Specifically, first, we propose a new execution-time-aware scheduling policy that exploits parallelism across and within services and avoiding inter-service interference while keeping CPU utilization high and balanced. Second, we propose *execution-time-aware sharding* that partitions latency-sensitive, short-running services from longer-running services and show its benefits in avoiding latency due to head-of-line blocking between services.

In the following sections we describe the design and implementation of Pyrosome, and we evaluate Pyrosome against existing container systems. Our evaluation shows that Pyrosome scales linearly and improves throughput by 10×, reduces median latency of a social media microservices application by 4×, and it can support services more than an order of magnitude more granular than today's services. Pyrosome can also handle substantial and sudden load increases with no impact on client-perceived latency.

2 Background

Microservices. Microservices is an application architecture that decomposes an application into small services with each running in a set of independent processes. This benefits software development and maintenance because each service can be developed and maintained independently reducing human communication costs. Microservices applications are usually deployed as a cluster of OS-level containers, and services are packaged using easily-deployable container images [43], which can contain a service and its software dependencies. Kubernetes [6] is a popular container-orchestration system used to deploy, maintain and scale container clusters. Though containers are generally purely OS-level mechanisms, they are also often deployed on top of virtual machines.

Serverless Computing. Serverless computing [1, 4, 30] is similar to microservices, and platforms for them should help by offering granular, pay-as-you-go billing where the cloud provider manages scaling, and serverless has helped in some domains [5, 10, 11, 15, 33, 36, 38, 41].

However, developers are not completely freed from provisioning and scaling concerns with serverless computing. Serverless functions are typically implemented in containers and the high cost of cold starts can impact the availability of serverless applications during scaling up. AWS provides Provisioned Concurrency [34] for developers to specify the expected peak load and reserve resources for it, pushing the burden of operations back to developers.

2.1 Pyrosome Design

Figure 1 shows the basic design of Pyrosome. Pyrosome works as a collective compute container for deployed services. Applications deploy a set of logical services to a machine running Pyrosome. Pyrosome as the software layer between hardware resources and applications, provides simple interfaces for applications that hides the complexities of managing

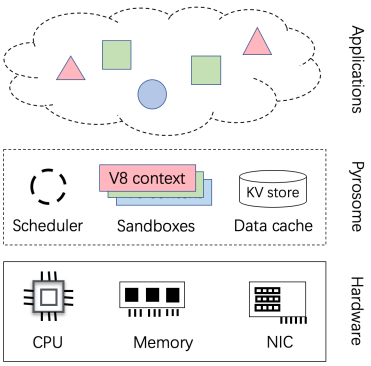


Figure 1: Pyrosome Overall Design

hardware resources. It can fluidly reprovision hardware resources on-demand, providing an automated resource pool for applications. It also provides shared access to local cached state for instances of stateful services. Pyrosome achieves efficiency and low-overhead by leveraging language-level sandboxing. Each service is isolated in a very light-weight V8 context. Note that Pyrosome's design is not limited to the V8 runtime, and can be implemented using any language runtime that provides light-weight isolation [21, 46].

Here, we elaborate on the core aspects of its design before describing the details of its implementation.

Simple high-level abstractions. Pyrosome should provide simple high-level programming abstractions for cloud application developers and hide the complexities of the underlying software infrastructure. Developers can then focused on application level logics.

Low-overhead Deployment and Flexible Modularization. Services should be logically isolated but not necessarily strongly physically isolated. High communication costs impede developers, forcing them to develop and scale their application in coarser units to amortize costs. Smaller services limit the impact of failures, allow for more rapid iteration, and support flexible resource sharing.

Instant and Transparent Resource Reprovisioning. Containers' high cold start results in wasteful overprovisioning and resource fragmentation; since containers are slow to create, operators must provision extra containers for each service to accommodate load changes [34]. In comparison, Pyrosome's isolated sandboxes can be created nearly instantaneously, and idle sandboxes (V8 Isolates) only occupy 3 MB of memory (compared to 35 MB for a container [9]). Furthermore, Pyrosome's design only requires one V8 Isolate per core, and each service on the same core has its own V8 Context, which is lighter-weight. Hence, Pyrosome can keep a Context per-service per-core, which avoids sandbox creation overheads altogether, letting it instantly shift any service's load to any core.

Low-overhead Runtime Level Scheduling and Optimization Containers and virtual machines depend on operating-system-level scheduling, which is problematic for inter-dependent fine-grained computations due to costly thread context switches and the kernel's lack of request-level visibility. Pyrosome efficiently schedules thousands of functions with better visibility because its runtime-level scheduler runs in the same process as services, letting it observe and schedule requests with low overhead. Pyrosome's low scheduling overhead lets services use CPU cores efficiently at much finer timescales.

Moreover, Pyrosome's runtime-level scheduler is able to implement advanced optimizations exploiting information collected at runtime for microservices and serverless workloads, which is beyond the capabilites of OS-level scheduling. We demonstrate this with a scheduling policy called *execution-time-aware sharding* that optimizes application tail latencies. Microservices and serverless applications vary in structure and per-invocation execution times [40]. Some applications are composed mostly of short-running functions [19, 22], but some services (e.g. machine learning) rely on long-running, compute-intensive services [26]. By observing these differences, Pyrosome can avoid problems like tail latency due to head-of-line blocking caused by long-running functions. Inspired by the Minos key-value store's size-aware sharding [14], *execution-time-aware sharding*, extends Minos' approach to generalized, opaque functions whose runtime varies rather than just basic get/put operations.

In Process Data Cache Microservices are usually stateful. To achieve high performance, not only should services communicate between each other with low overhead, services should also access data with low overhead. In current microservices approach, the state of services are usually stored in external databases or separate database services. That entails cross boundary overheads for data accesses. Our design provides data stores that resides in the same process with the services. Each service has its own datastore that is shared by all the instances of the service. Services can store ephemeral data in the datastore. For persistent data, the datastore can function as the cache between the service and external databases.

3 Implementation

Pyrosome is built on the Seastar framework [39]. Seastar's shared-nothing execution model enables Pyrosome to scale nearly-linearly across cores. Pyrosome leverages the V8 JavaScript engine as lightweight language level isolation for services. Other runtimes like Lucet and Wasmer could be

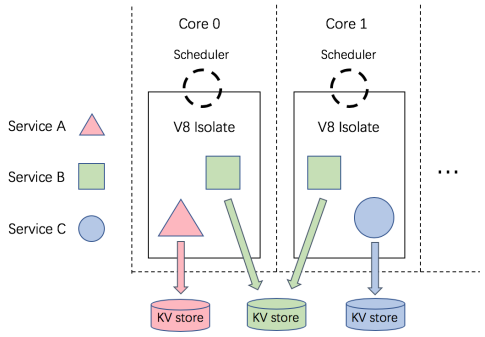


Figure 2: Pyrosome Basic Architecture

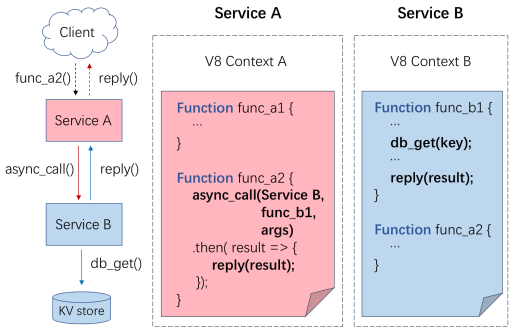


Figure 3: Programming Interfaces.

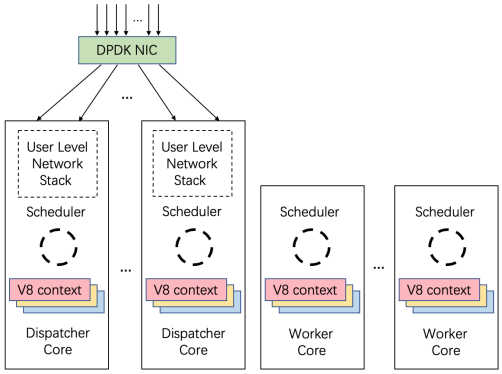


Figure 4: Pyrosome Networking Architecture

used instead to support WebAssembly to give developers more flexibility [21, 46].

In Pyrosome, each core processes incoming requests in a single-threaded event loop. Each core also hosts a V8 isolate, which is an instance of the V8 engine (Figure 2). Services on the same core are isolated in different V8 Contexts, which fault isolation so errors in one service don't cause other services to malfunction. They also provide an isolated execution environment with its own set of global variables, built-in objects and functions, so that services can be developed and maintained independently. There is also a scheduler on each core to schedule the execution of services. Each service has a key-value store that can be used to store state that must persist between invocations; and it also acts as a cache for external database accesses. Instances of the same service running on different cores share the same key-value store.

3.1 Programming Interfaces.

Pyrosome provides a list of very simple programming interfaces. `async_call()` and `reply()` allows a service to invoke and communicate with another service. The `.then()` callback of the caller service will be invoked to receive the replied result. `db_get()` and `db_set()` are provided for stateful services to read and write their key value stores.

Figure 3 shows the code structure of services on Pyrosome.

A service is a group of related functions that share the same V8 context. Each function in a service is addressed independently. For example, if a client wants to invoke `func_a2()` in `Service_A`, it can issue an HTTP request to address "/Service_A/func_a2" to Pyrosome. Services on Pyrosome can also call functions of other services asynchronously, through the `async_call()` interface. `async_call()` is implemented as a C++ binding that calls the scheduler to schedule and run the callee function. The caller can pass messages through `async_call()` to the callee function as the argument via shared memory. Complex objects can be serialized JSON strings; the callee must parse the JSON to retrieve the objects. This makes it possible to implement sophisticated APIs between services. The reply of a service call is sent through the `reply()` interface to the caller service. The `.then()` interface is used to implement the callback to be invoked once the replied result is received. A client-facing service also uses `reply()` to return result to the requesting client in an HTTP reply.

Stateful services can use `db_get()` and `db_set()` to read and write their key value stores. When instances of the same service are running on different cores, their writes could cause consistency problems. To ensure consistency, we implement compare and swap semantics for accesses to key value stores. A version number is attached to each record; when an instance of a service reads and then writes to its key value store, `db_set()` checks if the given version number matches the current version number of the record in the key value store to ensure no other instance on some other core has updated the record between the read and write. If the version numbers don't match, the `db_set()` returns an `Abort` status to inform the caller that the write has failed. Services should check the return status of `db_set()`, if failed they should read the updated record and redo the operation on the record and retry `db_set()`.

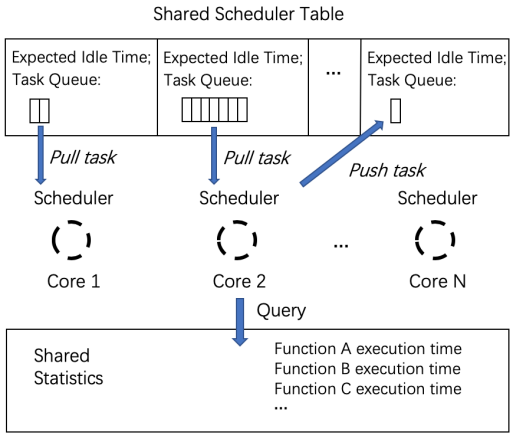


Figure 5: Scheduler Workflow

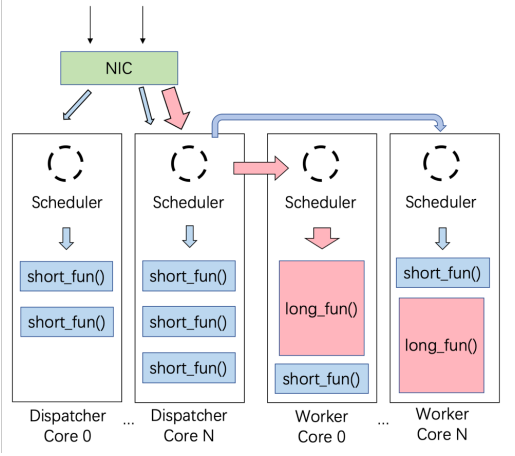


Figure 6: Execution-Time-Aware Sharding

3.2 Networking

In Seastar, all incoming network connections are randomly distributed among all cores in hardware by the network card. Pyrosome modifies Seastar so that cores are divided into *dispatcher cores* and *worker cores* (Figure 4); incoming network connections are only distributed to and processed by the dispatcher cores. We use DPDK to bypass the kernel networking stack; Pyrosome relies on Seastar's user-level network stack on dispatcher cores rather than the standard Linux TCP stack. Dispatcher cores can also run services. The worker cores do not receive client requests from the network; instead they only run services scheduled to them from the dispatcher cores. This design prevents long-running functions from blocking network packet processing. It also enables *execution-time aware sharding* which we detail below.

There are two levels of load balancing in Pyrosome. First, incoming HTTP requests are randomly distributed to the dispatcher cores (Figure 4). The second level of load balancing is done by the scheduler on each core, which we describe next.

3.3 Execution-Time-Aware Scheduling

Figure 5 shows the architecture of the Pyrosome scheduler. Each core has its own scheduler, and they share one scheduler table. The scheduler table tracks the task queue and states of each core, including whether the core is busy or idle and the expected time that the core will become idle if it is busy.

When a service function is called (whether by a client request or by another service function), the local scheduler is invoked. The scheduler tries to find an idle core in the scheduler table; if no idle core is found, then it finds the core with the earliest expected idle time and pushes the function into the task queue of that target core. The scheduler updates the status of the target core by querying a set of shared statistics to find the expected execution time of the function and updating the expected idle time of the target core accordingly. The task queues and core states are protected by a mutex for safe concurrent access. The scheduler on each core constantly pulls tasks from its own task queue and runs them.

Pyrosome's approach is simple and avoids the cost and complexities of preemption while working well for the workloads we measure (e.g. ZygOS must expose a virtualized APIC interface to userspace in order to efficiently trigger inter-processor interrupts [35] making it vulnerable to denial-of-service attacks).

With both network-level and scheduler-level load balancing, Pyrosome can balance load among all cores and avoid hot spots and accommodate bursty workloads well. In addition, the design exploits the internal parallelism of applications well. When a service function issues concurrent calls to multiple services, these functions are automatically spanned to different cores.

Pyrosome also implements a new approach to scheduling different service functions called *execution-time-aware sharding* (Figure 5); the idea is similar to size-aware sharding, which has been used to improve tail latency in key-value stores [14]. The key idea is that by isolating the short-running service functions (which are likely to be latency sensitive) from longer-running functions, tail latency is improved since it reduces head-of-line blocking. The scheduler collects function execution times at runtime and use them for future scheduling decisions, under the assumption that function execution times are mostly stable across invocations.

The scheduler uses a threshold to determine if a function is a short-running function or a long-running function. Long-running functions are scheduled on worker cores only, while short-running functions can be scheduled on any core. Limiting long-running functions to worker cores prevents them from blocking network packets processing on the dispatcher cores and improves latencies of the short-running functions by avoiding head-of-line blocking. The threshold can be adjusted

for different workloads and SLA requirements.

A problem is that concurrent requests can interfere with each other. When a function calls another function using `async_call()`, the callee function returns result to the caller function asynchronous through a callback function. The execution of the callback function can be delayed by another function from another concurrent request, resulting in higher latency. The impact of this problem depends on the CPU load and the structure of the workflow. Complex workflows consist of large number of inter-dependent functions are more impacted. We implemented an optimization called *fused-execution mode* if the scheduler detects high latency of requests. With *fused-execution mode* the workflow of a request will be executed in a run-to-completion mode to avoid such interferences.

4 Evaluation

We evaluate Pyrosome with a series of microbenchmarks and a social network application ported from DeathStarBench [19] seeking to answer five key questions, which we summarize results for here:

Does Pyrosome improve service throughput and efficiency over conventional microservice platforms? Pyrosome scales linearly to 16 cores, and it handles 10× the requests per second than the same service deployed as a containerized microservice.

What are the limits of service granularities that Pyrosome can support? On conventional containers decomposing a service that runs for 4 ms per invocation into 4 services will reduce the efficiency to about 50%. Our measurements suggest that Pyrosome can decompose a 4 ms computation into as many as 80 services before the efficiency drops below 50%, so Pyrosome can support services more than an order of magnitude more granular than today's services.

Does Pyrosome help on complex microservices? Our results show, Pyrosome reduces median latency of a social media microservices application by 4×.

Does Pyrosome handle load shifts well? Pyrosome can handle substantial and sudden load increases with no impact on client-perceived latency. Similar load increases will cause latency spikes that renders services unavailable on when using `Kubernetes`' autoscaling to handle shifts.

4.1 Hardware Setup

We run our experiments on the CloudLab testbed [37]. In all experiments each physical node is a Dell PowerEdge R430 server with two 2.4 GHz Intel Xeon E5-2630v3 8-core CPUs (16 hardware threads) and 64 GB RAM interconnected by 1 Gbps Ethernet.

4.2 Comparison to Kubernetes & Containers

In this section, we compare performance of microservices applications deployed on Pyrosome versus deployed in containers. To make a fair comparison, in this section we run Pyrosome without DPDK or its user-level network stack so that the Pyrosome deployment and container deployment are both running with the same default Linux kernel network stack.

4.2.1 Throughput and Scalability

First, we compare the throughput and scalability of a small service deployed on Pyrosome to one running as a conventional, containerized microservice to show the benefits of its reduced communication and isolation costs.

In this experiment we use 4 physical nodes. Each node runs Ubuntu 20.04 with Linux 5.4. We deploy a container orchestration platform using `Kubernetes` for microservice creation. One node runs the `Kubernetes` controller and another node is used as the server node either running Pyrosome or, for the baseline, the container cluster. Another node hosts the external (MongoDB) database that the microservices access. The last node runs a `wrk` client which generates load in a closed-loop (for 10 seconds per run with results averaged over 10 runs for each data point).

The user service has a `login()` function that validates the password of a user login request. Clients send requests via HTTP to call `login()`; the `login()` function then fetches the user's credentials before checking them against the function's arguments. On Pyrosome, the `login()` function is implemented in JavaScript; the container-based service is implemented in Go. By default in Pyrosome, after the HTTP request triggers `login()`, the request is handled entirely within Pyrosome. The user's credentials are cached in Pyrosome's local KVStore cache. When `login()` is deployed as a conventional microservice, the user's credentials must be accessed from the external MongoDB node. This simple function is fairly representative of many of functions in microservices, and it lets us compare Pyrosome against a baseline, conventional container-based approach to microservices.

Figure 7 shows the results as we run the service on an increasing number of CPU cores. If user credentials are cached in Pyrosome's local KVStore ("pyrosome w/o db") throughput is improved by 10× over a container-based deployment of the same service ("container w/ db"). When user credentials are not cached in Pyrosome ("pyrosome w/ db"), its throughput immediately collapses to match the performance of the container-based approach. So, eliminating costly, synchronous remote accesses for data is crucial to Pyrosome's performance. When running Pyrosome, these remote accesses to MongoDB cause CPU utilization to drop to 50% as threads block waiting on the database and experience costly context switches. Of course, a container-based solution can perform local caching as well, but even when we eliminated the remote

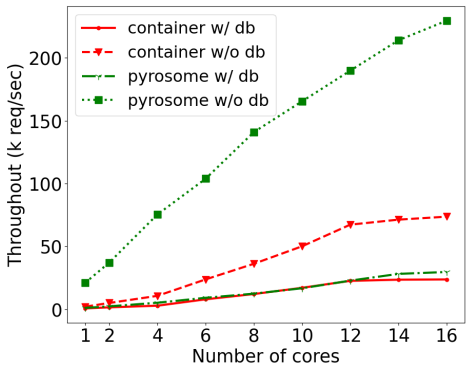


Figure 7: Throughput Under Increasing Load

database access from the container-based service ("container w/o db") its performance only improved by 2×.

Together these results show that improving runtime overheads only helps if other bottlenecks in remote communication are also eliminated, demonstrating the importance of Pyrosome's holistic approach. It not only eliminates costly inter-service communications, but it also eliminates data access costs via in-process caching. Containers could have a shared data cache running via another container on the local machine, but data accesses will still suffer costly cross-container boundary crossings.

Finally, Pyrosome also improves scalability. Container-based services can be scaled by adding additional cores to a container or by adding additional containers, each running on its own core. Here, `login()` scales better when adding a container per core, which is also common practice for most microservices. Even so, the container-based service scales less efficiently and flattens entirely after 12 hardware threads. Pyrosome scales nearly linearly to 16 hardware threads (2 hardware threads for each of the 8 physical cores).

4.2.2 Service Decomposition Costs

Here we microbenchmark performance as we progressively decompose a service function into finer and finer-grained services both with Pyrosome and using containers. The experiment runs on two physical nodes; one to run the service within Pyrosome or within Docker containers. We compare Pyrosome with two different implementations of the container version. One uses the conventional HTTP protocol for communications between containers, the other uses the Apache Thrift protocol [16] which is faster than HTTP. In both container implementations the service function is implemented in JavaScript running on Node.js. The other node is used as the client, and it runs `wrk2` in an open loop at a low request rate to measure the latency. Each run averages many samples, and each data point is the average over 10 runs.

In this experiment we emulate the decomposition of a service and measure the costs. Figure 8 shows an example of decomposition. First, we can split a 8 ms service it into two services, each of which runs for 4 ms and chained together to complete the functionality. Then we can split it further into 4 services each running for 2 ms.

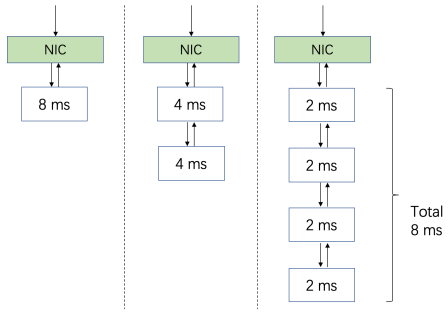


Figure 8: Decomposing a Service into Finer Services

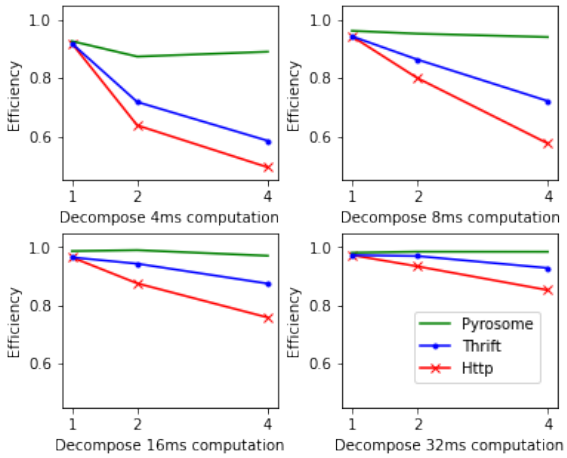


Figure 9: Efficiency of Decomposition

Figure 9 shows the results. We vary the time length to emulate the decomposition of short and long running services. The upper left is the decomposition of a 4 ms service, the upper right is a 8 ms service, the lower left is a 16 ms service, and the lower right is a 32 ms service. From the results, we can see that with containers decomposition lowers the efficiency significantly. For Pyrosome, decomposition cost is low, and the efficiency almost remains the same when a service is decomposed into a chain of smaller services. Also, the gap between containers and Pyrosome is much bigger when short running services are decomposed into even smaller services, demonstrating that the low decomposition cost on Pyrosome allows much finer grain microservices. From the measured numbers we can calculate the cost of a round-trip call between two services, which is about 44 µs on Pyrosome. So if we decompose a 4ms computation into 80 services on Pyrosome, which means each service runs for 50 µs, the efficiency will drop below 50%. This is an estimation of the

limit of decomposition on Pyrosome, which is more than an order of magnitude finer granularity compared to what can be achieved using containers.

4.2.3 Social Network Application

To evaluate how reduced decomposition costs can improve the performance of real world applications, we implemented the social network application from the DeathStarBench [19] on Pyrosome, and then we compare the latency of the `compose-post` request.

We use 2 physical nodes. One is used as the server to run Pyrosome or the DeathStarBench. Another node is used as the client, and it runs `wrk2` to average latency across many requests under a low request rate using an open loop; each data point is the average value of 10 runs.

`compose-post` is one of the client-facing APIs provided by the social network application. Its function is to upload a new post from a user. Similar to Twitter posts, a post can include text, media, user mentions and URLs. Figure 10a shows the graph of services invoked by a `compose-post` request. A `compose-post` HTTP request from a client first arrives at the Nginx server, which acts as the front-end of the application. The Nginx server then parses the HTTP request and invokes other services. In the case of `compose-post` request, the Nginx invokes 4 backend services to process the request. The `text` service is invoked to process and upload the text of the post, it then invokes the `user_mention` service to process user mentions and the `url_shorten` service to shorten URLs in the post. The `user` service processes the username and id of the author of the post. The `unique-id` service creates a unique post id for the post. The `media` service processes the media references of the post. The outputs of all the services mentioned above are sent to the `compose_post` service to be assembled into the final version of the post. `compose_post` then invokes the `post_storage` service to store the post into MongoDB. A memcached server is also used by the `post_storage` service to cache posts for faster access. `compose_post` also invokes the `user_timeline` service and `write_home_timeline` service to update timelines. `write_home_timeline` invokes the `social_graph` service to get followers of the user and update their timelines. Users' timelines are stored in MongoDB with Redis as cache. In DeathStarBench all these services are implemented in C++ and isolated in containers with the Apache Thrift communication protocol.

Figure 10b shows the structure of the social network application ported to Pyrosome, which is very similar to the DeathStarBench version, except that these services are implemented as JS functions in V8 sandboxes, and user posts and timelines are stored in the underlying datastore on Pyrosome.

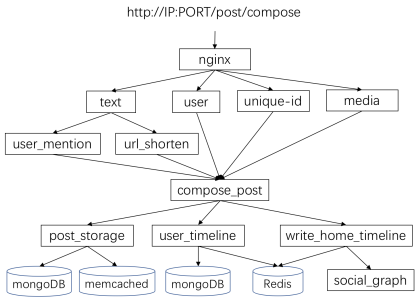
Figure 10c shows the median and 99 percentile latencies of the `compose-post` request on Pyrosome and DeathStarBench measured under low request rate. The median latency is 2.17 ms for Pyrosome versus 8.51 ms for DeathStarBench. The 99 percentile latency is 3.53 ms for Pyrosome versus 10.00 ms for DeathStarBench. The results show that with low decomposition costs, Pyrosome can reduce the median latency of a complicated microservices application by 3/4 and the 99 percentile latency by 2/3 compared to the containerized version of the application.

4.2.4 Resource Elasticity

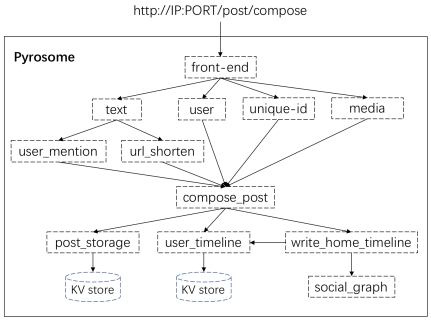
In this experiment we compare the scaling capabilities of `Kubernetes` [6] and `Pyrosome` in reaction to changing workloads. For `Kubernetes`, we use a cluster of 3 physical nodes, one runs the `Kubernetes` controller, another runs the `Kubernetes` cluster to host microservice containers, the remaining one is used to run the `wrk2` clients. `wrk2` can generate open loop load with specified request rate, we use it to control the offered load. For `Pyrosome` we use two physical nodes, one for `Pyrosome` server and the other for `wrk2` clients. We use the default settings for the `Kubernetes` autoscaler and set 90% CPU utilization as the trigger metric for scaling.

We use the same workload as in §4.2.1 that uses the `login` function in the `User` service, and we run the `User` service without an external database accesses to eliminate the its impact. We run two scripts on the client node at the same time, one runs `wrk2` to generate workload, the other runs `wrk2` under low load to measure latency. The workload generation script starts with very low load at 100 reqs/s, then later increases offered load to a much higher request rate. For `Kubernetes` the request rate increases to 16k reqs/s and for `Pyrosome` it increases to 150k reqs/s. From the throughput experiment of §4.2.1, that represents about 20% of the maximum throughput of `Kubernetes` and about 65% of the maximum throughput of `Pyrosome`.

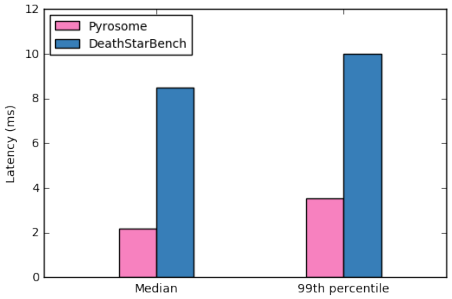
Figure 11 shows the measured median latencies of `Kubernetes` and `Pyrosome` during the workload. It shows that the median latency is greatly increased during the scale up period of `Kubernetes`. For `Pyrosome` is not impacted after the sudden large load increase. We observe that the `Kubernetes` autoscaler struggles to meet this load; after the load increase, the autoscaler is triggered 3 times, and each time it starts 3 or 4 more containers. This shows two problems with scaling via `Kubernetes`. First, the cost of starting new containers is high. Second, it doesn't know how many new containers need to be provisioned to meet the increased load. The autoscaler makes the speculation that starting 3 or 4 containers may be able to handle the load increase. However, when the load increase is too high, the autoscaler will be triggered multiple times to allocate enough resource, which slows the scaling up process further. From Figure 11, we can see that the scaling up phase of `Kubernetes` is more than 2 minutes, and during that time the `User` service is effectively unavailable because the service is saturated and all requests experience very high latency. On the contrary, `Pyrosome` is



(a) `compose-post` request in DeathStarBench.



(b) `compose-post` request on Pyrosome.



(c) Latency of `compose-post` request.

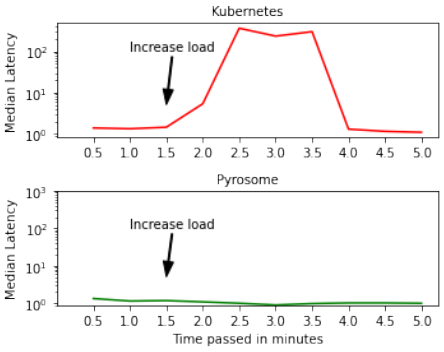


Figure 11: Kubernetes and Pyrosome react to sudden load increase.

able to immediate pivot resources to whichever service within its runtime needs them, even at fine-grained timescales. As a result, it handles bigger load increases with no impact on client-perceived latency.

4.3 Scheduler Evaluation

In this section, we evaluate the design of Pyrosome scheduler. In all of the experiments in this section we run Pyrosome on 16 cores, 8 of which are dispatcher cores. Each dispatcher core is allocated a hardware NIC queue and runs its own user level network stack and DPDK driver. In these experiments, Pyrosome's user-level networking stack adds increased pressure on scheduling; since the reduced overheads and response times mean scheduling and inter-service interference at the primary factors that determine client-observed response times, especially for short-running functions.

4.3.1 Execution-Time-Aware Scheduling

In this microbenchmark we demonstrate that Pyrosome's execution-time-aware scheduling allows better use of CPU resources than baseline approaches that have no visibility into service invocation runtimes. To show this, we construct a "fanout" application (Figure 12) that invokes 10 functions

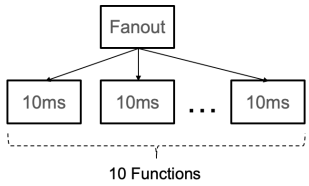


Figure 12: Fanout application.

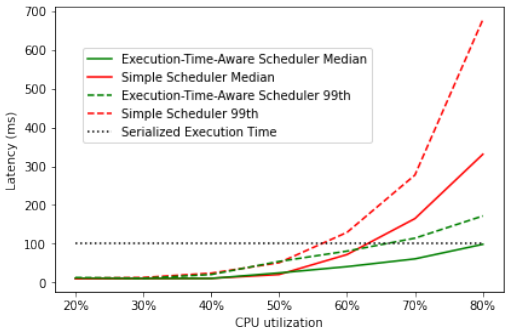


Figure 13: Latency of fanout application.

each runs for 10 ms in parallel. As a baseline, we implement a "simple scheduler" that doesn't leverage the knowledge of function execution times; instead, it simply tries to distribute work randomly to under-loaded cores when the core that receives a request is under high load.

We vary the load to test the schedulers' ability to optimize parallelizable functions under different CPU loads. From Figure 13 we can see that under low load, both schedulers can use available CPU resources to run parallelizable functions in parallel so that to reduce latency. When CPU load increases it becomes harder to find available CPU resources to run the functions in parallel, as a result more of the functions are run sequentially thus latencies increase. Under higher CPU load, our execution-time-aware scheduler is better than the simple scheduler at finding CPU resources to parallelize execution. The execution-time-aware scheduler efficiently parallelizes execution even under more than 70% CPU utilization while

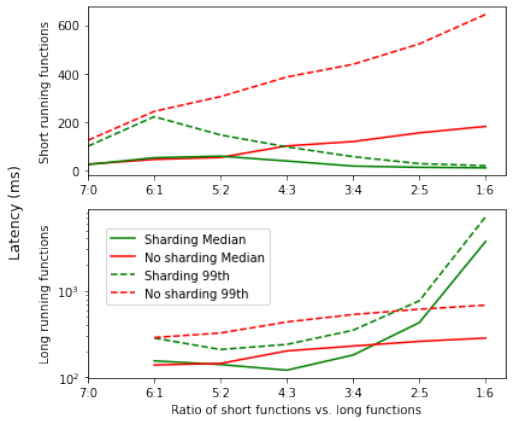


Figure 14: Mixed workload with/without sharding.

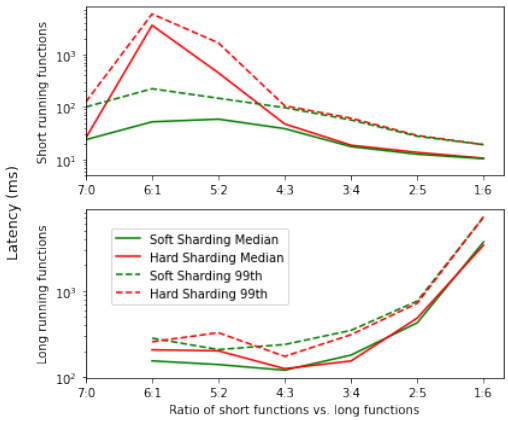


Figure 15: Soft sharding versus hard sharding.

response times under simple scheduler spike.

4.3.2 Execution-Time-Aware Sharding

In this microbenchmark, we evaluate Pyrosome's ability to handle a mixture of long-running and short-running functions with its execution-time-aware sharding. A short-running function runs for 10 ms and a long running function runs for 100 ms. The workload is held constant to use 70% of all CPU resources and the ratio of short-running functions and long-running functions is varied. The ratio is calculated by total CPU time occupied by short-running functions versus CPU time occupied by long-running functions. The upper graph of Figure 14 shows the latencies of short-running functions with and without sharding (the green line and the red line respectively), and the lower graph shows the latencies of long-running functions.

In this microbenchmark the scheduler uses execution-time-aware sharding to limit long-running functions on work cores while short-running functions can be scheduled on both the dispatcher cores and worker cores. This is called *soft sharding* and it is the default sharding policy of execution-time-aware sharding. From the graph we can see that with sharding, latencies of both the short-running functions and the long-running functions are much lower, especially for the short-running functions. When the ratio is 1:6 the latencies of long-running functions increase greatly, this is because long-running functions are limited to the 8 worker cores and the load of long-running functions at this ratio has exceeded the CPU capacity of the 8 worker cores.

We also compared soft sharding with *hard sharding* where the scheduler only schedules short running functions on dispatcher cores. Figure 15 shows that with hard sharding when the ratio of short-running functions is high the dispatcher cores will be overloaded. Overloading dispatcher cores not only results in much higher latencies for short-running functions, but also worsens the latencies of long-running functions

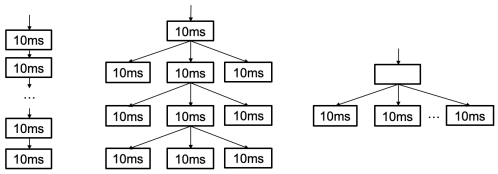


Figure 16: Different application structures.

as the dispatching of long-running functions is also impacted. In the middle, when the load is about evenly split between short-running and long-running functions and none of the cores is overloaded, the performance of soft sharding is similar to that of hard sharding. Overall, soft sharding is a better sharding policy that achieves similar performance at avoiding head-of-line blocking of long-running functions as hard sharding, while allowing more flexibility for scheduling short-running functions.

4.3.3 Scheduling Complex Applications

Microservices and serverless applications are naturally comprised of workflows of inter-dependent functions. User perceived end-to-end latency for these applications depends on the completion of the execution of the whole workflow. The structure of a workflow dictates how its execution can be optimized by the scheduler. Fanouts of functions in a workflow provide opportunities for parallel execution to optimize end-to-end latency of the workflow. Previous microbenchmarks showed that Pyrosome's execution-time-aware scheduler can leverage the parallelism within a workflow to optimize the latency. However, this leads to a challenge: it is also possible for concurrent workflows to block each other's execution resulting in worse latencies.

To evaluate the scheduler's performance with different application structures, in this microbenchmark we construct three example applications as shown in Figure 16. The left side shows an application of a sequential chain of functions,

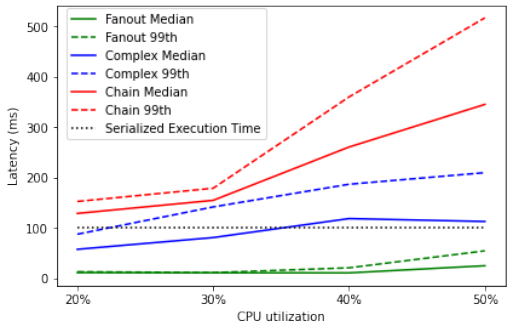


Figure 17: Latencies of applications with different structures.

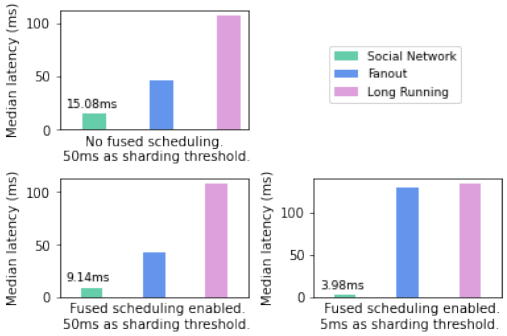


Figure 18: Latencies of mixed applications

the right side shows an application of fanout of functions, and the middle shows an application of mixed sequential and fanout stages. The three applications have the same total serialized execution time (100 ms).

Figure 17 shows the latencies of running the three applications on Pyrosome. From the result we can see that for high-fanout applications the scheduler can optimize latency even under relatively high CPU load. For complete sequential chain application the scheduler cannot optimize latency at all, and its latency worsens with increased CPU load because of more interferences between concurrent workflows under higher CPU load. For complex applications with both fanout and sequential stages, the scheduler can optimize its latency under low CPU load, but with increased CPU load its latency becomes worse than sequential execution due to interference.

This benchmark shows that application-level information, such as the structure of the application workflow, can be leveraged to optimize the end-to-end latency of complex microservices and serverless applications. Extracting this information automatically to optimize application execution is an interesting future direction; for now, we implement a *fused-execution mode* for applications. Requests from applications marked as fused are run using run-to-completion model with the entire workflow executed on a single core (as if it is a single function). One heuristic that may make sense for triggering fused-execution is to use a similar execution-time aware approach where if the latency of a workflow is greater its serialized execution time it is fused. With such a heuristic, the scheduler could leverage applications' internal parallelism to optimize latency under low CPU load, and when load increases, interference from concurrent workflows could be mitigated.

4.3.4 Mixed Workloads

In this experiment we evaluate the scheduler's performance with a mixture of heterogeneous applications. The mixture consists of the social network application, the fanout application and the long-running 100 ms function from previous experiments. We keep Pyrosome at about 60% CPU utilization with each application contributing to about 1/3 of the load. The results not only shows that Pyrosome's scheduler can accommodate a mixture of very different applications, but also it shows that different scheduler parameters can prioritize different applications. The upper-left graph shows the latencies with 50 ms as the scheduler's sharding threshold. The scheduler is able to optimize the latency of the fanout application, but the latency of social network application is high. The bottom-left graph turns on fused-execution mode for the social network application; this improves latency, but it still experiences some head-of-line blocking from the fanout application. In the bottom-right graph we set the threshold for sharding to 5 ms, which forces the fanout application to be scheduled on the worker cores with the long-running functions. This greatly reduces the latency for social network application because head-of-line blocking is avoided, but at the cost of increased latency for the fanout application because it is limited on worker cores with fewer CPU resources for parallel optimization. These parameters can be mechanisms for higher-level policies for mixed workloads.

5 Related Work

Lighter-weight Sandboxes. There are many efforts trying to address the performance and scaling challenges of serverless by reducing the overhead of containers or adopting lightweight sandboxes. SAND [3] addresses the issues by running functions of the same application as processes in the same container to reduce isolation costs, and providing fast local messaging bus for functions on the same host. Firecracker [2] is a new Virtual Machine Monitor (VMM) built by Amazon that runs serverless functions in lightweight MicroVMs with a minimized Linux kernel. Nightcore [22] is a serverless function runtime for latency-sensitive interactive microservices that implements fast internal function calls and other optimizations to achieve high performance with container-based isolation. All these solutions still rely on heavy weight sandboxes such as processes, containers and VMs, so their overheads and cold start latencies are still

high compared to Pyrosome. They also rely on OS-level scheduling which is costly and lacks request-level visibility. There are also solutions from the academia and the industry [18, 21, 24, 42, 45–47] that leverage lightweight language level isolation such as the V8 JavaScript or WebAssembly runtime to build fast serverless frameworks. But these frameworks don't consider the microservices scenario of complicated interconnections and communications between a large number of services.

Actor systems. Pyrosome's approach to containing several logical services within a single process runtime bears similarity to actors systems. Actors are small logical agents that communicate and trigger computation and concurrency via messages. Frequently many actors are multiplexed on a single machine or within a single runtime allowing similar optimizations to Pyrosome. For example, Scala's original actor system implements some inter-actor messaging as direct procedure call [20]. There are some popular actor systems used in production [8, 28, 44], and some recent efforts seek to improve inter-host messaging efficiency in actor systems [29]. Ray is a recent actor-based approach for executing distributed analytics tasks [31]. Pyrosome differs from these systems since it is focused on microservice-oriented architectures. Instances of the same service of a microservices application often need to share the same underlying database, whereas actors don't share state.

6 Conclusion

Today's cloud-native applications are naturally structured in a higher-level and more decomposed way than classic monolithic applications run on the abstraction of a full machine. However, today's inefficient legacy cloud software infrastructure and abstractions hinder the performance and scalability of these applications. Pyrosome shows that a clean-slate design of cloud services runtime targeted toward microservice- and serverless-era applications can greatly improve performance, enable more granular decomposition of services, and scale up/down better than today's container-based platforms. Additionally, Pyrosome shows that we can implement smart scheduling optimization that leverages information collected at runtime to utilize cores efficiently and minimize application tail latency.

References

[1] Azure functions. https://azure.microsoft.com/en-us/services/functions/, 2022.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *2018 Usenix Annual Technical Conference (USENIXATC 18)*, pages 923–935, 2018.

[4] Inc. or its affiliates. Amazon Web Services. Aws lambda. https://aws.amazon.com/lambda/, 2022.

[5] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[6] The Kubernetes Authors. Production-grade container orchestration, 2022. URL: https://kubernetes.io.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[8] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41*, 2014.

[9] Zack Bloom. Cloud computing without containers. https://blog.cloudflare.com/cloud-computing-without-containers/, 2018.

[10] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[11] Ryan Chard, Tyler J Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard. Serverless supercomputing: High performance function as a service for science. *arXiv preprint arXiv:1908.04907*, 2019.

[12] Matthew Clark. Powering bbc online with nanoservices. https://www.bbc.co.uk/blogs/internet/entries/5bdabd53-090e-4611-a5d5-4faea05aeb35, 2018.

[13] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[14] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, 2019.

[15] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 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)*, pages 363–376, 2017.

[16] Apache Software Foundation. Apache thrift. https://thrift.apache.org, 2022.

[17] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[18] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: a serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, pages 265–279, 2020.

[19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 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*, pages 3–18, 2019.

[20] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202 – 220, 2009. Distributed Computing Techniques.

[21] Pat Hickey. Lucet takes webassembly beyond the browser | fastly. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2022.

[22] Zhipeng Jia and Emmett Witchel. 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*, pages 152–166, 2021.

[23] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[24] MJ Jones. How compute@edge is tackling the most frustrating aspects of serverless. https://www.fastly.com/blog/how-compute-edge-is-tackling-the-most-frustrating-aspects-of-serverless, 2020.

[25] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for Microsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019.*, pages 345–360, 2019.

[26] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[27] Marcin Kolny. Scaling up the prime video audio/video monitoring service and reducing costs by 90%. https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90, 2023.

[28] Inc. Lightbend. Akka. http://akka.io/, 2022.

[29] Christopher S. Meiklejohn, Heather Miller, and Peter Alvaro. PARTISAN: Scaling the Distributed Actor Runtime. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 63–76, Renton, WA, July 2019. USENIX Association. URL: https://www.usenix.org/conference/atc19/presentation/meiklejohn.

[30] Microsoft. Cloud functions. https://cloud.google.com/functions, 2022.

[31] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association. URL: https://www.usenix.org/conference/osdi18/presentation/moritz.

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019.*, pages 361–378, 2019.

[33] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud function services. *arXiv preprint arXiv:1911.11727*, 2019.

[34] Danilo Poccia. New – provisioned concurrency for lambda functions, 2019. URL: https://aws.amazon.com/cn/blogs/aws/new-provisioned-concurrency-for-lambda-functions/.

[35] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3132747.3132780, https://doi.org/10.1145/3132747.3132780.

[36] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, 2019.

[37] Robert Ricci, Eric Eide, and the CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:*, 39(6):36–38, 2014.

[38] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. Infaas: A model-less inference serving system. *arXiv preprint arXiv:1905.13348*, 2019.

[39] Inc. Scylla DB. Seastar. http://www.seastar-project.org, 2019.

[40] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[41] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[42] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. *arXiv preprint arXiv:2002.09344*, 2020.

[43] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys european conference on computer systems 2007*, pages 275–287, 2007.

[44] The Erlang Team. Erlang programming language. https://www.erlang.org/, 2022.

[45] Kenton Varda. Introducing cloudflare workers: Run javascript service workers at the edge. https://blog.cloudflare.com/introducing-cloudflare-workers/, 2017.

[46] Inc. Wasmer. Wasmer. https://wasmer.io, 2022.

[47] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.