Pyrosome: A Runtime for Efficient and Fine-grained Microservices

Anonymous authors
Paper under double-blind review

Abstract

Developers have shifted from deploying applications on physical machines, to virtual machines and to containers. Such a shift of abstractions has 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 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-era JavaScript applications we call Pyrosome.

Pyrosome provides simple programming interfaces for executing JavaScript code, storing and sharing data, and low-overhead communication without worrying about resource allocation and scheduling. It leverages the V8 JavaScript runtime's low-overhead language-based sandboxing that avoids the full state and scheduling costs of operating system processes or containers. This enables a holistic scheduler that quickly redistributes load among cores and exploits parallelism in applications. The scheduler also avoids tail latency with execution-time-aware partitioning. Pyrosome speeds up microservice applications by as much as $4\times$ and improves throughput by $10\times$, as demonstrated by the DeathStarBench microservices benchmarks. Additionally, we show Pyrosome balances load nearly instantly compared to standard microservices 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) [14, 23]. The evolution of cloud computing technologies, coupled with the need for efficient software development and maintenance, has led to fundamental changes in how cloud applications are designed and deployed. This has resulted in new cloud application paradigms. 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 microservices architecture, allows each service to be developed, maintained and scaled independently while matching organizational structures, which improves developer productivity especially for large-scale applications [10]. Some organizations have further reduced service size (e.g. the BBC's nanoservices platform [9]) to limit the impact of failures, enable more rapid iteration, and support flexible resource sharing.

However, today's cloud platforms, as the legacy from a decade of evolution of cloud technologies, are comprised of layers of software infrastructure and abstractions that are redundant and inefficient for fine-grained microservices. For example, the containerization layer, on top of which most of today's microservices 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 [16]. 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 applications during scaling up.

A key mismatch stems from the strong isolation inherent in virtual machines and containers. This level of isolation, designed to mitigate risks between untrusted users, is unnecessary for services within a single application where only basic inter-service fault-isolation is needed. Furthermore, many microservice applications are adopting a cloud-native approach, relying on cloud platforms to provide high-level abstractions and hide the complexities of underlying infrastructure. Thus the virtualized operating system abstraction provided by containerization is no longer needed.

Therefore, we propose Pyrosome, a clean-slate design for a microservices deployment system. Recognizing that strong security isolation is often unnecessary, we designed Pyrosome as a proof-of-concept runtime with minimal isolation costs between mutually trusted microservices and applications:

- Microservices of the same application should be mutually trusted, so there's no need for strong security isolation between them.

- Many microservice applications are deployed in trusted environments, such as private infrastructure or private clouds, where security is less of a concern.

- On public cloud, mutually trusted applications (e.g. those from the same client) can share the same sandbox (e.g. a VM), eliminating unnecessary strong security isolation between them. Pyrosome runtime can be deployed within such a sandbox to support these applications.

Two key aspects of Pyrosome's design lead to these benefits. First, 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. Furthermore, the low overhead of isolation and communication enables Pyrosome to support much finer-grained microservices than today's container-based approach. This also allows Pyrosome to maintain a pool of inexpensive sandboxes, eliminating costly *cold starts* and enabling instant resource reprovisioning for handling sudden load changes. Second, Pyrosome's design demonstrates that radically redesigning cloud software infrastructure can unlock opportunities for advanced optimizations in cloud-native applications. With low-overhead isolation, each service operation on Pyrosome completes in microseconds or less. This allows its scheduler to operate on a much finer timescale than conventional microservice schedulers (e.g. Kubernetes). Pyrosome's scheduler supports several new optimizations inspired by recently fine-grained dispatch policies that minimize tail latency in services like key-value stores [5, 11, 25, 38, 40]. Specifically, we propose: 1) an execution-time-aware scheduling policy that exploits parallelism, avoids inter-service interference, and maintains high, balanced CPU utilization; and 2) *execution-time-aware partitioning* that isolates latency-sensitive services from longer-running ones to prevent head-of-line blocking.

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 microservice 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. 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. Because of the many benefits of microservices architecture, nowadays, many large web applications are implemented as microservices. For example, the popular Netflix video streaming platform is implemented as a cluster of more than 700 microservices [37], Uber's backend has 4,000 microservices [55], Meta's customer applications such as Instagram or Facebook mobile, also depend on its large-scale microservice architecture [21]. Microservices applications are usually deployed as a cluster of OS-level containers, and services are packaged using easily deployable container images [46], which can contain a service and its software dependencies.

VM and Container-based Isolation Virtual machines rely on hypervisor-based [4, 27] isolation. With much smaller attack surface than the OS kernel and the strong isolation provided by the hardware virtualization features, VM-based isolation is preferable in scenarios where strong security isolation is needed. But in VM-based isolation, each sandbox must have its own OS kernel and libraries, resulting in heavy performance costs. Container-based virtualization emerged to address this problem. In container-based virtualization sandboxes share the same OS kernel, and rely on OS-level virtualization mechanisms, such as namespaces and cgroups in the Linux kernel, for security and resource isolation. Docker [35] is a popular container platform used by many of today's applications. Kubernetes [3] is a popular container-orchestration system used to deploy, maintain, and scale container clusters. The security isolation of containers is weaker than VMs, as the attack surface is much larger with the shared OS kernel. As a result, containers are also often deployed on top of virtual machines for stronger security isolation between groups of containers.

Lightweight Language-based Isolation Safe high-level programming languange and runtime mechanisms can be leveraged as a lightweight alternative to heavyweight VM-based and OS-level isolation. The V8 JavaScript runtime [49] is widely used to run JavaScript code in isolation within the same browser or server process. Cloudflare [50] has built a serverless edge compute platform with the V8 JavaScript runtime that is able to support massive multi-tenancy with low overhead. Rust [48] is a safe systems programming language that allows lightweight isolation, and is used to build extensible multi-tenant, low latency storage systems [7, 29, 53]. A disadvantage of language-based isolation is that usually it only supports a single language. A solution to this problem is offered by WebAssembly (Wasm) [17, 18], which is a compilation target for programming languages. Untrusted code of various languages can be compiled to Wasm and run in Wasm's default light-weight sandboxed environment. Wasm is not only used in the web environments, but also used to build cloud systems [12, 45, 51] and edge computing systems [15, 24] that achieve significant performance improvement over container-based systems.

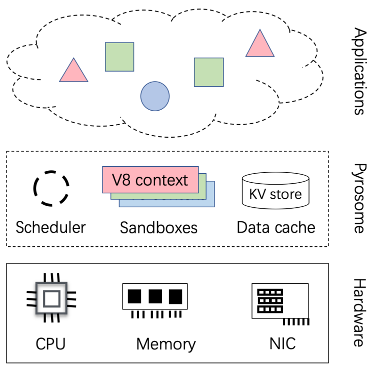


Figure 1: Pyrosome overall design

3 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 hardware resources. It can fluidly reprovision hardware resources on-demand, providing an automated resource pool for applications. It also provides shared access to a 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 lightweight 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 [20, 52].

3.1 Design Goals

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

Simple high-level abstractions. Pyrosome should provide high-level programming abstractions that simplify cloud application development. By hiding the complexities of the underlying infrastructure, developers can focus solely on application logic.

Low-overhead Deployment and Flexible Modularization. While logically isolated, services in Pyrosome are not strictly physically isolated, reducing communication costs and promoting efficient resource utilization. This allows for finer-grained services, which limit the impact of failures, enable rapid iteration, and support flexible resource sharing.

Instant and Transparent Resource Reprovisioning. Unlike Containers with slow cold start times, Pyrosome's lightweight sandboxes V8 Isolates are created almost instantly, minimizing over-provisioning [39] and resource fragmentation. Each core requires only one V8 Isolate, with individual services on that core utilizing even lighter-weight V8 Contexts. Idle sandboxes (V8 Isolates) only occupy 3 MB of memory (compared to 35 MB for a container [8]). Hence, Pyrosome can keep a V8 Context for each service on each core, which avoids sandbox creation overheads altogether and enables Pyrosome to dynamically shift service loads 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's runtime-level scheduler operates within the same process as the services, granting it direct visibility into request execution and enabling highly efficient scheduling of thousands of functions with minimal overhead. This allows services to utilize CPU cores effectively at much finer timescales.

Moreover, Pyrosome's runtime-level scheduler unlocks advanced optimizations exploiting information collected at runtime for microservices workloads, which is beyond the capabilities of OS-level scheduling. We demonstrate this with a scheduling policy called *execution-time-aware partitioning* that optimizes application tail latencies. Microservices applications vary in structure and per-invocation execution times. Some applications are composed mostly of short-running functions [16, 22], but some services (e.g. machine learning) rely on long-running, compute-intensive services [26]. By observing these differences, Pyrosome can mitigate issues like head-of-line blocking, which can significantly impact performance. This approach, inspired by the Minos key-value store's size-aware sharding [11], extends the concept to generalized, opaque functions with variable runtimes.

Low-overhead Access To State Microservices are typically stateful, requiring efficient access to data for optimal performance. Traditional approaches often store service state in external databases or separate database services, incurring significant overhead due to cross-boundary data access. Pyrosome addresses this by providing in-process data stores. Each service has a dedicated data store, shared by all its instances, residing within the same process. This minimizes latency and maximizes efficiency. These data stores can hold ephemeral data or act as a cache for persistent data stored in external databases.

3.2 Threat Model

While language-based isolation offers low-overhead, it supports weaker security isolation compared to virtualization and process-based isolation, especially after the discovery of side-channel attacks such as Spectre [28] that exploits the CPU's speculative execution. Malicious code can leverage such attacks to bypass language-based isolation and access data outside its sandbox if that data resides within the same address space. Spectre attack has already been proven effective against JavaScript and WebAssembly [33, 43].

Consequently, Pyrosome is designed as a runtime for running mutually trusted services and is not suitable for running untrusted code. All services within the same Pyrosome runtime instance effectively share the same process and address space, necessitating mutual trust. Pyrosome's light-weight isolation primarily focuses on fault isolation, preventing software bugs in one service from impacting others.

It is reasonable to assume that services within the same microservice application are typically mutually trusted. For instance, services within the Netflix video streaming platform [37] should be mutually trusted, as all service code is developed and maintained by the same company. Multiple mutually trusted microservice applications, such as those from the same organization, can also share a Pyrosome runtime instance. Meta presents a real world example of deploying multiple mutually trusted microservices applications within a shared infrastructure [21]. At Meta, multiple customer applications, such as Instagram or Facebook mobile, and internal applications, such as dashboard or internal tools, share the same infrastructure running across dozens of geographically-distributed datacenters. In fact, the notion of an application is ill-defined within Meta's microservices topology, as individual service instances may process work on behalf of multiple applications.

If running untrusted code within a service is necessary, an approach similar to Chrome's Site Isolation [41] can be adopted. isolating the service in its own dedicated process.

4 Implementation

Pyrosome is built on the Seastar framework [44]. Seastar's shared-nothing execution model enables Pyrosome to scale almost linearly across cores. Pyrosome leverages the V8 JavaScript runtime, which is directly embedded into the Seastar framework, lightweight language level isolation for services. Other runtimes such as Lucet and Wasmer, which support WebAssembly, could be used instead to give developers more flexibility [20, 52].

In Pyrosome, each core processes incoming requests in a single-threaded event loop. Each core also hosts a V8 isolate, an instance of the V8 JavaScript runtime (Figure 2). Services residing on the same core are isolated within separate V8 Contexts, with fault isolation so errors in one service do not cause other services to malfunction. A V8 Context also provides an isolated execution environment with its own set of global variables, built-in objects and functions. This promotes independent development and maintenance of services.

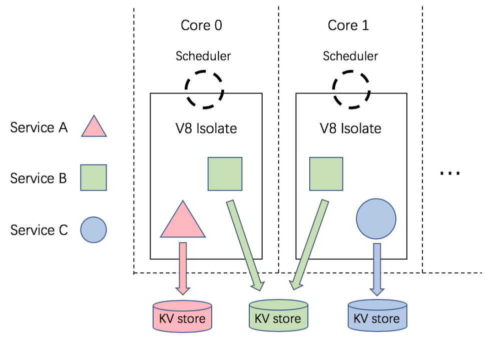


Figure 2: Pyrosome basic architecture

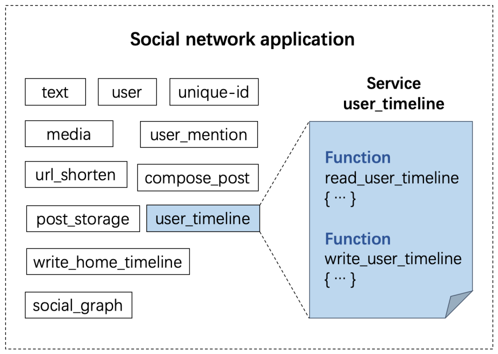
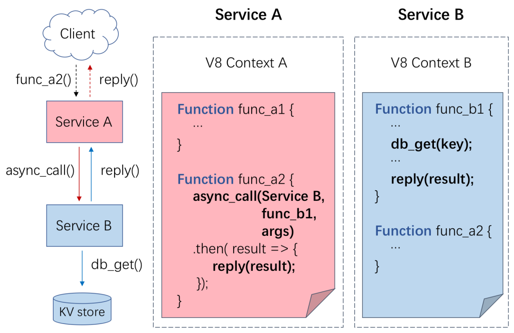


Figure 3: Pyrosome application structure

Furthermore, each core features a scheduler to manager the execution of services and a dedicated key-value store for each service. This store holds states that must persist between invocations, and also acts as a cache for external database accesses. Instances of the same service running on different cores share the same key-value store, ensuring data consistency.

4.1 Application Structure.

Figure 3 illustrates the structure of an example microservice application in Pyrosome. This application comprises a collection of services, each implements a logical component. A service may encompass several related functionalities, implemented as callable functions that can be invoked by other services. For instance, the `user_timeline` service manages user timelines and includes two functions: `read_user_timeline()` and `write_user_timeline()` , handling read and write operations respectively. Functions within the same service on the same core share a V8 Context. Note that the term "function" in subsequent chapters refers to these callable functions.



```
Function func_a1 {
    ...
}

Function func_a2 {
    async_call(Service B,
               func_b1,
               args)
    .then( result => {
        reply(result);
    });
}
```

```
Function func_b1 {
    ...
    db_get(key);
    ...
    reply(result);
}

Function func_a2 {
    ...
}
```

Figure 4: Programming interfaces.

4.2 Programming Interfaces.

Pyrosome provides a list of very simple programming interfaces. `async_call()` and `reply()` allow 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 from and write to their key value stores.

Figure 4 shows the code structure of services on Pyrosome. 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". 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. 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 reply is received. A client-facing service also uses `reply()` to return result to the requesting client via an HTTP reply.

Stateful services can use `db_get()` and `db_set()` to read from and write to 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 from and then writes to its key value store, `db_set()` checks if the given version number matches the record's current version number to ensure no other instance on some other core has updated the record between the read and write. If the version numbers do not 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, redo the operation on the record and then retry `db_set()`.

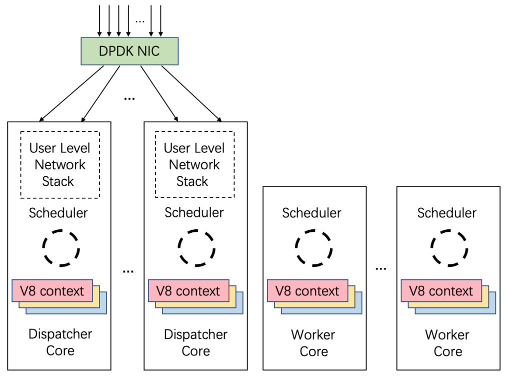


Figure 5: Pyrosome networking architecture

4.3 Networking

In Seastar, all incoming network connections are randomly distributed among all cores by the network card. Pyrosome modifies Seastar so that cores are divided into *dispatcher cores* and *worker cores* (Figure 5); incoming network connections are 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 partitioning* which we will detail later.

There are two levels of load balancing in Pyrosome. First, incoming HTTP requests are randomly distributed to the dispatcher cores (Figure 5). The second level of load balancing is done by the scheduler on each core, which we describe next.

4.4 Execution-Time-Aware Scheduling

As shown in Figure 6, in Pyrosome each core has its own scheduler, and all local schedulers share one scheduler table. The scheduler table tracks all cores' task queues and states, including whether the core is busy or idle and the expected time that the core will become idle if it is busy.

Functions are the basic scheduling unit of the Pyrosome scheduler. When a function is called (whether by a client request or by another service), the local scheduler is invoked. The scheduler first attempts to locate an idle core within the scheduler table. If no idle core is available, it selects the core with the earliest expected idle time. The function is then added to the task queue of that target core. To maintain accurate scheduling information, the scheduler updates the

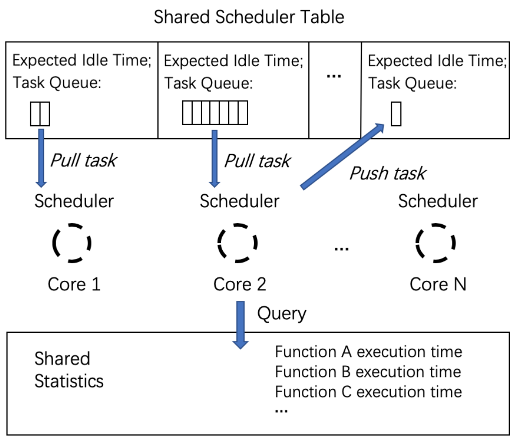

Shared Scheduler Table
Expected Idle Time; Task Queue:
Expected Idle Time; Task Queue:
...
Expected Idle Time; Task Queue:
Pull task
Pull task
Push task
Scheduler
Scheduler
Scheduler
Core 1
Core 2
...
Core N
Query
Shared Statistics
Function A execution time
Function B execution time
Function C execution time
...

Figure 6: Scheduler workflow

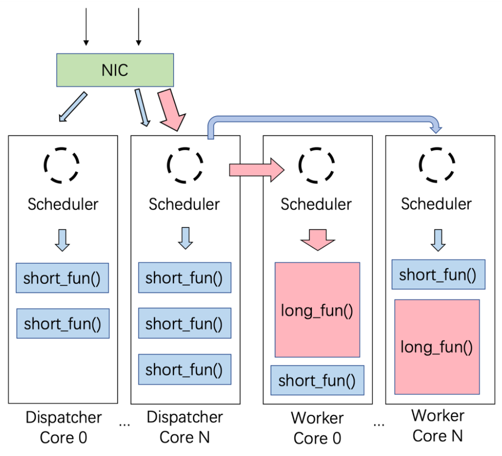


Figure 7: Execution-time-aware partitioning

target core's status. This involves querying shared statistics to estimate the function's expected execution time and adjusting 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 [40], making it vulnerable to denial-of-service attacks).

With both network-level and scheduler-level load balancing, Pyrosome can balance load among all cores, avoid hot spots and accommodate bursty workloads well. In addition, the design exploits the internal parallelism of applications well. When a 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 partitioning* (Figure 7); the idea is similar to size-aware sharding, which has been used to improve tail latency in key-value stores [11]. The key idea is that by isolating the short-running 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 and use the average execution time 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 ones 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.

One issue is the potential for inteference between concurrent requests. When a function utilizes `async_call()` to invoke another function, the called function asynchronously returns its result via a callback. However, the execution of this callback can be delayed by functions from another concurrent requests, leading to increased latency. The severity of this issue is influenced by both CPU load and workflow structure, with complex workflows comprising numerous interdependent functions being more susceptible. To mitigate this, we've introduced an optimization called *fused-execution mode*. Activated by the scheduler upon detecting high request latency, this mode executes the entire workflow of a request in a run-to-completion sequence, minimizing interference and ensuring smoother processing.

Our scheduler serves as a proof-of-concept, showcasing the potential of low-overhead runtime level scheduling. It is not intended as a full-fledged implementation, and we make simplifying assumptions, such as stable and predictable function execution times. In reality, execution times can vary significantly. While this work does not address all such real-world challenges, we believe they can be overcome with further engineering effort. For instance, more sophisticated algorithms could predict execution time, potentially by learning from function inputs. Analyzing production microservice traces [21, 30, 32, 55] could also yield valuable insights for improving the scheduler's design and implementation, offering a promising direction for future work.

5 Evaluation

We evaluate Pyrosome with a series of microbenchmarks and a Social Network application ported from DeathStarBench [16], seeking to answer five key questions:

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 a containerized microservice deployment.

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 more than an order of magnitude of granularity.

Does Pyrosome help on complex microservices? Our results show, Pyrosome reduces median latency of a social media microservice 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 when using `Kubernetes`' autoscaling.

5.1 Hardware Setup

We run our experiments on the CloudLab testbed [42]. 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.

5.2 Comparison to Kubernetes & Containers

In this section, we compare performance of microservice 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.

5.2.1 Throughput and Scalability

To demonstrate the advantages of Pyrosome's reduced communication and isolation overhead, we begin by comparing the throughput and scalability of a small service deployed on Pyrosome against its conventional, containerized counterpart.

In this experiment, we use 4 physical nodes with each node running Ubuntu 20.04 with Linux 5.4 kernel. We deploy a container orchestration platform using `Kubernetes`. 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 service 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).

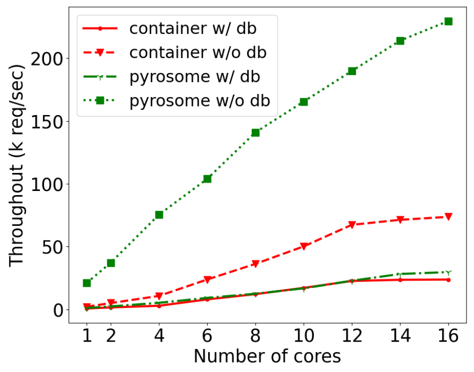


Figure 8: Throughput with increasing number of cores. Number of containers is equal to number of cores, with each container pinned to it's own core.

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.

Figure 8 shows the results of running 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 ("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 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, which not only eliminates costly inter-service communications but also eliminates data access

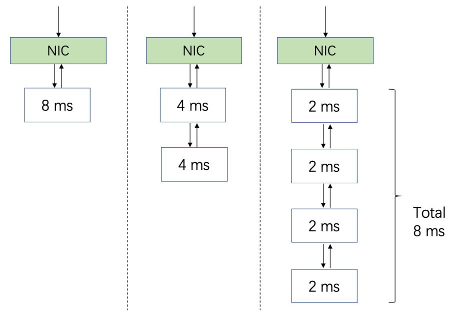


Figure 9: Decomposing a service into finer services

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 with each running on its own core. Here, `login()` scales better when adding a container per core, which is also common practice for most microservice deployment. 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).

5.2.2 Service Decomposition Costs

Here we microbenchmark performance as we progressively decompose a service 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 [13] which is faster than HTTP. In both container implementations the service 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 9 shows an example of decomposition. First, we split a service that implements 8ms of computation into two services, each of which runs for 4 ms and chained together to complete the total 8ms computation. Then we split it further into 4 services each running for 2 ms. In all these cases, the total computation time is the same (8ms), but as the computation is decomposed into more services, more context switching and communication costs are added in between the services. We measure the total time to

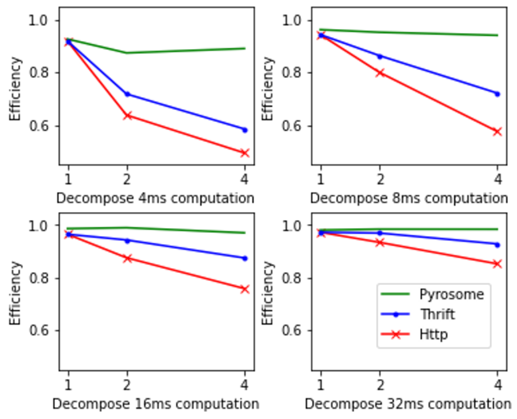


Figure 10: Efficiency of decomposition

complete the service chain, and then calculate the efficiency as:

$$\text{Efficiency} = \frac{\text{Total Service Computation Time}}{\text{Service Chain End-to-End Completion Time}}$$

Figure 10 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-grained 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.

5.2.3 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 [16] on Pyrosome, and then we compare the latency of the `compose-post` request. We chose the Social Network application for our evaluation as it exemplifies typical microservice applications, featuring a cluster of interacting services and utilizing databases for caching and storage. While other microservice benchmark applications exist, such as the Hotel

Reservation and Media Microservices applications from the Deathstarbench and the TrainTicket application [56], they share similar structures and service counts with the Social Network application. Although some minor distinctions exist, like TrainTicket's use of synchronous and asynchronous invocations alongside message queues, these are not our primary focus. The Social Network application should suffice to evaluate the performance impact of reduced decomposition costs on typical microservice applications.

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 11a 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 11b 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, with user posts and timelines stored in the underlying data store.

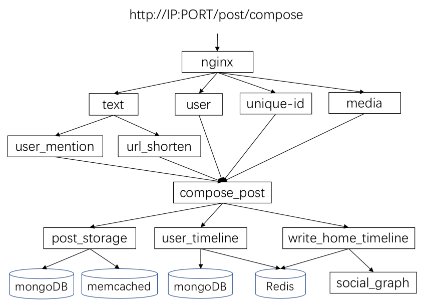
Figure 11c shows the median and 99 percentile latencies of the `compose-post` request measured under low request rate. The median latency is 2.17 ms for Pyrosome versus 8.51 ms for DeathStarBench. The 99 percentile latency is 3.53 ms for Pyrosome versus 10.00 ms for DeathStarBench. The results show that with low decomposition costs, Pyrosome can reduce the median latency of a complicated microservice application by 3/4 and the 99 percentile latency by 2/3 compared to the containerized version of the application.

5.2.4 Resource Elasticity

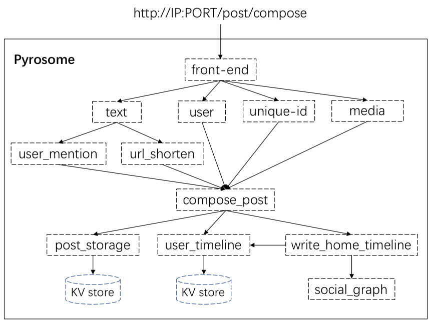
In this experiment we compare the scaling capabilities of `Kubernetes` [3] and `Pyrosome` in reaction to changing workloads. For `Kubernetes`, we use a cluster of 3 physical nodes, one runs the `Kubernetes` controller, another runs the `Kubernetes` cluster to host microservice containers, the remaining 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 §5.2.1 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 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 §5.2.1, that represents about 20% of the maximum throughput of `Kubernetes` and about 65% of the maximum throughput of `Pyrosome`.

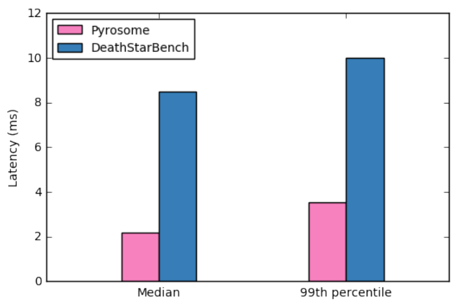
Figure 12 shows the measured median latencies of `Kubernetes` and `Pyrosome` during the workload. It shows that the median latency is greatly increased during the scale up period of `Kubernetes`. For `Pyrosome` is not impacted after the sudden large load increase. We observe that the `Kubernetes` autoscaler 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 does not 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 12, 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 able to immediate pivot resources to whichever service within its runtime, even at fine-grained timescales. As a result, it handles bigger load increases with no impact on client-perceived



(a) `compose-post` request in DeathStar-Bench.



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



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

Figure 11: Compare `compose-post` request in DeathStarBench and Pyrosome

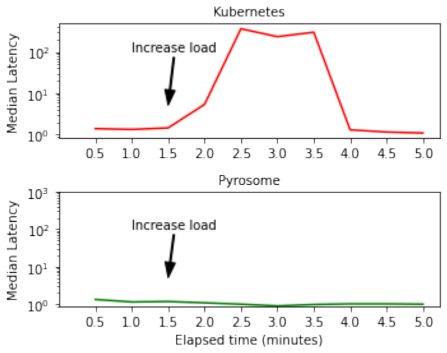


Figure 12: Kubernetes and Pyrosome react to sudden load increase.

latency.

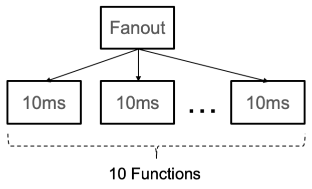


Figure 13: Fanout application.

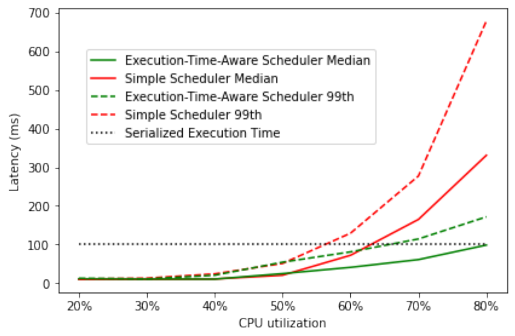


Figure 14: Latency of fanout application.

5.3 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 that scheduling and inter-service interference are the primary factors that determine client-observed response times, especially for short-running functions.

5.3.1 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 runtimes. To show this, we construct a "fanout" application (Figure 13) that invokes 10 functions each runs for 10 ms in parallel. As a baseline, we implement a "simple scheduler" that does not leverage the knowledge of function execution times; instead, it simply tries to distribute work randomly to under-loaded cores.

We vary the load to test the schedulers' ability to optimize parallelizable functions under different CPU loads. From Figure 14 we can see that under low load, both schedulers can use available CPU resources to run functions in parallel 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 causing latencies to 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

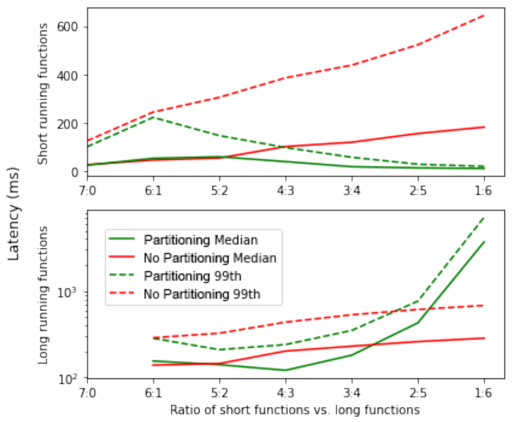


Figure 15: Mixed workload with/without partitioning.

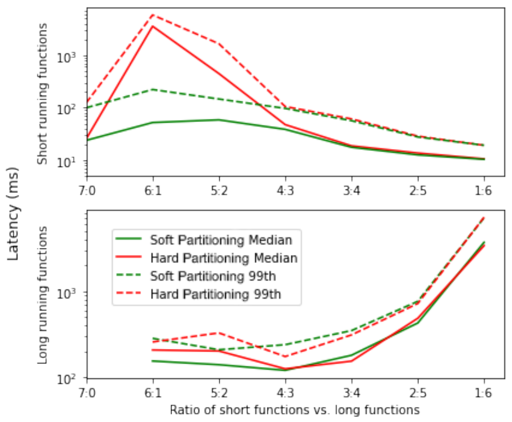


Figure 16: Soft partitioning versus hard partitioning.

response times with simple scheduler spike.

5.3.2 Execution-Time-Aware Partitioning

In this microbenchmark, we evaluate Pyrosome's ability to handle a mixture of long-running and short-running functions with its execution-time-aware partitioning. 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 15 shows the latencies of short-running functions with and without partitioning (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 partitioning 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 partitioning* and it is the default partitioning policy of execution-time-aware partitioning. From the graph we can see that with partitioning, latencies of both the short-running and 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 partitioning with *hard partitioning* where the scheduler only schedules short running functions on dispatcher cores. Figure 16 shows that with hard partitioning 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 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 partitioning is similar to that of hard partitioning. Overall, soft partitioning is a better partitioning policy that achieves similar performance of avoiding head-of-line blocking as hard partitioning, while allowing more flexibility for scheduling short-running functions.

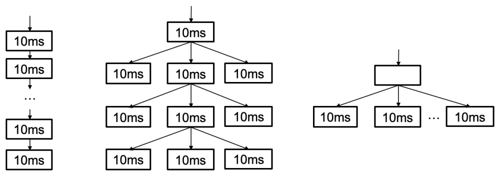


Figure 17: Different application structures.

5.3.3 Scheduling Complex Applications

Microservices 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 17. The left side shows an application of a sequential chain of functions,

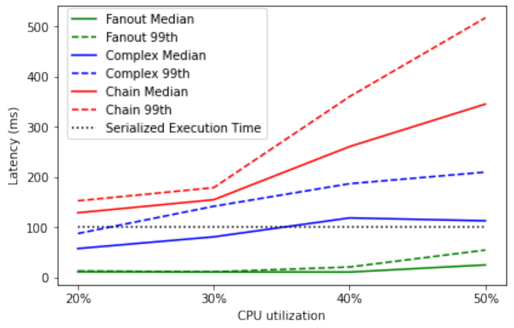


Figure 18: Latencies of applications with different structures.

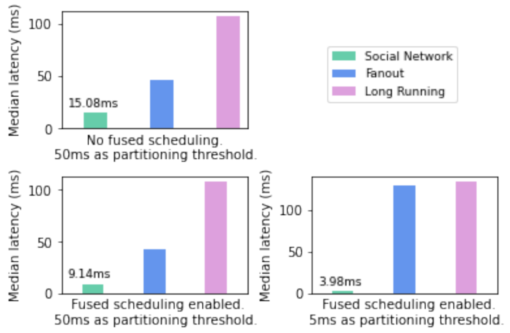


Figure 19: Latencies of mixed applications

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 18 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 microservice 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 executed in run-to-completion model with the entire workflow running on the same 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 than 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.

5.3.4 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 (Figure 19) not only show that Pyrosome's scheduler can accommodate a mixture of very different applications, but also show that different scheduler parameters can prioritize different applications. The upper-left graph shows the latencies with 50 ms as the scheduler's partitioning 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 partitioning 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.

6 Related Work

Lighter-weight Sandboxes. There are many efforts trying to address the performance and scaling challenges by reducing the overhead of containers or adopting lightweight sandboxes. SAND [2] 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 [1] 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 [15, 20, 24, 45, 50, 52, 54] that leverage lightweight language level isolation such as the V8 JavaScript or WebAssembly runtime to build fast serverless frameworks. But these frameworks do not consider the microservices scenario of complicated inter-dependencies 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 [19]. There are some popular actor systems used in production [6, 31, 47], and some recent efforts seek to improve inter-host messaging efficiency in actor systems [34]. Ray is a recent actor-based approach for executing distributed analytics tasks [36]. Pyrosome differs from these systems since it is focused on microservice-oriented architectures and leverages low-overhead language based isolation. Instances of the same service of a microservice application often need to share the same underlying database, whereas actors do not share state. In Ray, workers and actors on the same node are running in separate processes, whereas in Pyrosome all services are running within the same process to aggressively reduce context swithing and communication costs. Compared to Ray, Pyrosome is pushing the limit of performance optimization even further with language-based isolation.

7 Discussion

In this section we discuss the contributions and limitations of the paper, and possible future work directions.

7.1 Contributions

This paper makes a key contribution by evaluating the potential benefits microservices can reap from a clean-slate redesign of the cloud stack with light-weight, language-based isolation. Given the prevalence of container-based deployments, we choose container and kubernetes as our baseline for comparison. While both academia and industry have explored language-based isolation for cloud systems, these efforts primarily focus on serverless applications with limited number of functions, overlooking the unique challenges of microservices, such as their numerous services and intricate inter-service communication patterns. Our work addresses this gap by implementing a complex application from DeathStarBench and designing a scheduler specifically tailored to the demands of complex microservice deployments.

7.2 Limitations

The paper aims to evaluate the potential benefits of lightweight language-based isolation for microservice applications through a minimal proof-of-concept implementation. Consequently, it lacks many features required for production readiness, such as cluster management, orchestration, autoscaling, monitoring and package management etc. Furthermore, production-critical aspects like fault tolerance, reliability and consistency guarantees fall outside the scope of this work.

As previously discussed, Pyrosome's security isolation is weaker, requiring mutual trust between services within the same instance. This makes Pyrosome better suited for trusted internal infrastucture or private cloud platforms. However, many large-scale microservice applications, such as those at ~~Netflix and Uber,~~ Meta [21], operate within such trusted environments, suggesting significant potential value for Pyrosome in these contexts. Stronger security isolation can be achieved by isolating some services in their own Pyrosome instances, akin to Chrome's Site Isolation [41], but with the trade-off of increased performance overhead.

Furthermore, this work is limited to a single node. We focus on this scenario because the performance differences between heavyweight container-based isolation and lightweight language-based isolation are most pronounced within a single node. While Pyrosome can be scaled to multiple nodes using techniques like a cluster-wide scheduler, exploring this is beyond the scope of this paper.

7.3 Future Works

A natural extension of this work is to scale Pyrosome across a cluster of nodes. A cluster-wide scheduler must account for the significantly higher communication cost between nodes compared to within a single Pyrosome runtime. A design similar to the bottom-up two-level hierarchical scheduler in Ray [36], could be utilized to prioritize co-locating interdependent services on the same node. Furthurmore, the global scheduler could leverage application-level information, such as service dependencies, call graphs, data access patterns and runtime statistics, to optimize service placement across the cluster.

While the Social Network application used in our evaluation comprises roughly two dozens of services, many popular real-world microservice applications operate at a far larger scale, encompassing hundreds or even thousands of services [37,55]. Evaluating the performance gains achievable by deploying such large-scale applications on Pyrosome presents a compelling avenue for future investigation. As shown in our evaluation, the structure of the application workflow is found to be valuable information for optimizing the end-to-end latency of complex microservice applications. How to analyze and leverage such information to optimize microservice applications at large scale is also an interesting question.

While our current Pyrosome implementation utilizes the V8 JavaScript runtime, the underlying design is not language-specific. Exploring Pyrosome's implementation on WASM [17] runtimes [20, 52] presents an intriguing direction for future work, potentially expanding support to a wider range of programming languages.

8 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-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] 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.

[2] 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.

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

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[5] 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.

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

[7] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. Adaptive placement for in-memory storage functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 127–141, 2020.

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

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

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

[11] 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.

[12] Inc. Fermyon Technologies. Fermyon. http://www.fermyon.com/, 2024.

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

[14] 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.

[15] 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.

[16] 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.

[17] WebAssembly Community Group. WebAssembly. https://webassembly.org, 2024.

[18] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[19] 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.

[20] Pat Hickey. Lucet takes webassembly beyond the browser | fastly. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2024.

[21] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.

[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] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[28] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[29] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter:{bare-metal} extensions for {multi-tenant}{low-latency} storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, 2018.

[30] I-Ting Angelina Lee, Zhizhou Zhang, Abhishek Parwal, and Milind Chabbi. The tale of errors in microservices. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(3):1–36, 2024.

[31] Inc. Lightbend. Akka. http://akka.io/, 2024.

[32] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[33] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[34] 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.

[35] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[36] 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.

[37] Mayukh Nair. How netflix works: the (hugely simplified) complex stuff that happens every time you hit play, 2017. URL: https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b.

[38] 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.

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

[40] 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.

[41] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1661–1678, 2019.

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

[43] Stephen Röttger and Artur Janc. A spectre proof-of-concept for a spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html, 2021.

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

[45] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[46] 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.

[47] The Erlang Team. Erlang programming language. https://www.erlang.org/, 2024.

[48] The Rust Team. Rust. https://www.rust-lang.org, 2024.

[49] The V8 Team. V8 javascript engine. https://v8.dev, 2024.

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

[51] wasmCloud LLC. Wasm-native orchestration. http://wasmcloud.com/, 2024.

[52] Inc. Wasmer. Wasmer. https://wasmer.io, 2024.

[53] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship compute or ship data? why not both? In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 633–651, 2021.

[54] 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.

[55] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

[56] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 323–324, 2018.