Master's Capstone Project

Exploring the Unexplored in Death Star Bench

Charitha Bandi

cbandi@ucsc.edu

Winter 2023

Abstract

In recent years, there has been a notable shift in software architecture towards microservices, which are smaller and more loosely connected than the monolithic services. However, this shift has led to an increased need for research aimed at understanding the characteristics, requirements, and performance effects associated with microservice-based architectures. Performance issues, such as tail latencies, are often not evident until systems are deployed at large scales due to complex dependencies between services. To study these characteristics of microservices and their performance, many researchers use the Death Star Bench [3], a benchmark suite that can simulate the real-world large-scale systems, such as social networking apps, media apps, and e-commerce websites.

This project provides a system's perspective on analyzing large-scale microservices and, in particular, addressing the tail latencies [2] by taking two extremely different approaches such as critical path and asynchronous model analysis. Inspired from the Mystery machine's model [1], we have developed a model to identify the per request level critical path and therefore identify the bottleneck services. Additionally, the efficiency of microservice designs is dependent upon the efficiency of the asynchronous programming models upon which microservices are traditionally built. So, we conducted a series of micro-benchmarks to examine the impact of different asynchronous programming models on overall system performance and quantify these effects. Our findings show that the std::thread outperforms std::async by a significant amount, and also the promise type can effect the overall latencies observed. To minimize thread creation overheads and resource exhaustion risks, we recommend implementing std::async model with the thread pool. In addition, a considerable amount of research has been devoted in the recent years in the development of workload aware schedulers [4] to drastically improve the throughput, tail latency. Therefore, we evaluated the scheduling delays associated with asynchronous operations to gain further insight into potential performance improvements that could be achieved through the use of workload aware schedulers.

1 Introduction

Microservices are gaining popularity among enterprise and many cloud companies are adopting microservice-based architectures. A microservice is a small, independently deployable and scalable software service that encapsulates a specific function in a larger application. This architecture offers various benefits such as scalability, resilience, fault tolerance and isolation. However the life of a request in these systems would result in hundreds of complex microservice interactions. These interactions are deeply nested, asynchronous, and invoke numerous other downstream operations. As a result of this complexity, it is very hard to identify which underlying service(s) contribute to the overall end-to-end latency experienced by a top-level request. Answering this question is critical in many situations, including identifying bottleneck services and optimization opportunities etc.

In this project, we focus on the DeathstarBench which is a benchmark suite that focuses on large-scale real world applications with tens and thousands of unique microservices allowing us to study effects that only emerge at large scale. We picked social network application in the deathstarbench suite for our analysis because it is complex enough with tens of microservices and tends to display the issue of high tail latency when operated at heavy load. We employed two different approaches to figure out the causes of tail latency.

First approach is to do Critical path analysis on the system. Critical path is a list of services that directly contribute to the slowest path of the request processing in a distributed system. Therefore, performing the critical path analysis helps in identifying the critical path, through which we can pinpoint bottleneck services such that, optimizing these services can reduce the overall latency. Our critical path analysis model is based on the Mystery machine's model [1]. The major ways in which our model differs from that of the Mystery machines is that it uses a causal model by comparing each request trace with all other requests to find non contradicting relationships among the services and uses only the converged result. In contrast, our project performs the critical path analysis per request in isolation and aggregated stats of the critical paths generated are used to reason about the bottleneck services. Through the critical path analysis we are able to identify the bottleneck services and further investigating of these services provided new insights into how scheduling decisions can impact the overall performance of the system.

Microservices are designed to be lightweight, scalable, and loosely coupled, often leveraging asynchronous models to run multiple tasks concurrently. As such, the efficiency of microservices is dependent on the efficiency of the underlying implementations of these asynchronous models. Our second approach is to investigate the implementations of GNU C++ std::async and compare it with that of the performance of std::thread. By conducting a series of micro-benchmarks, we were able to identify the costs involved in launching asynchronous tasks using the GNU C++ std::async in async mode and understand the effect different promise types can have on the performance. Through our experiments we could show that the std::thread outperforms std::async by a significant margin, with lower costs associated with its usage.

The remaining sections of the paper provide a detailed account of both approaches. Section 2 introduces the deathstarbench, while Section 3 examines the tail latency observed in the social network workload of deathstarbench. This section further provides a more in-depth description of the critical path analysis and the asynchronous programming model analysis approaches we used to identify the bottleneck services and causes of the tail latencies.

2 DeathStarBench

Deathstarbench [3] is a benchmark suite for cloud microservices, designed to evaluate the performance and scalability of distributed systems and to facilitate the development of new techniques for improving the performance of these systems. The benchmark suite consists of six different real-world workloads that are representative of different type of distributed applications, including social networking, media services, hotel reservation, Banking and E-commerce websites. Each workload is designed to simulate a realistic set of user interactions with the application, and includes a mix of read and write operations. It is also designed to be highly configurable and extensible allowing researchers and developers to customize the benchmark to suit their specific needs.

In this project, we predominately focused on social networking application that allows users to perform actions such as following other users, creating posts with various types of content like text, media, links, and tags, which are then broadcasted to their followers. Users can also interact with posts by re-posting, replying publicly or sending direct messages to other users. It is a complex application designed using various microservices to handle these multitude of tasks as shown in the Figure 1.

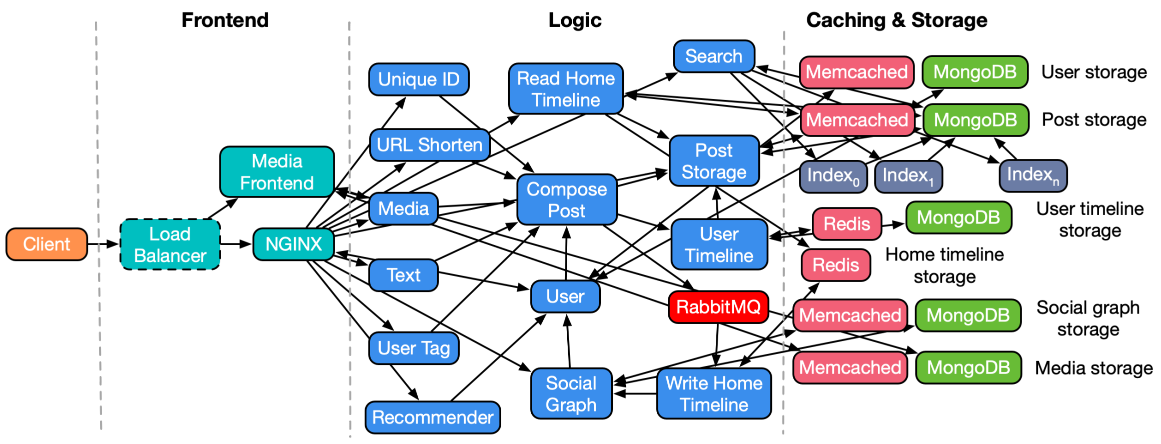
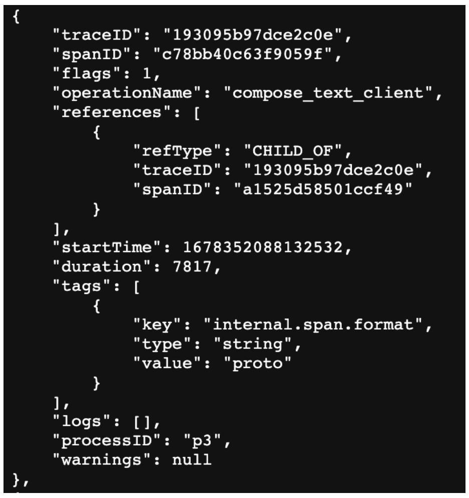


Figure 1: DeathstarBench's Social Networking application

To simulate the request load, Deathstarbench uses wrk2 benchmarking tool which is a closed



```
{
    "traceID": "193095b97dce2c0e",
    "spanID": "c78bb40c63f9059f",
    "flags": 1,
    "operationName": "compose_text_client",
    "references": [
        {
            "refType": "CHILD_OF",
            "traceID": "193095b97dce2c0e",
            "spanID": "a1525d58501ccf49"
        }
    ],
    "startTime": 1678352088132532,
    "duration": 7817,
    "tags": [
        {
            "key": "internal.span.format",
            "type": "string",
            "value": "proto"
        }
    ],
    "logs": [],
    "processID": "p3",
    "warnings": null
},
```

Figure 2: Span object in Jaeger traces

loop load generator, designed for testing the performance and scalability of HTTP services with the ability to customize request patterns and concurrency levels.

Death Star Bench is instrumented with Jaeger [5], which is an open-source distributed tracing system, to trace requests as they flow through the complex microservices architecture and stores them in a centralized database. Jaeger works with spans and traces as defined in Open-Tracing specification. A Span represents an unit-of-action in the microservice. It includes operation name, span duration, start-time along with the other information as shown in Figure 2. Traces are a collection of spans that are connected in a child-parent relationship, which specifies how requests are propagated through the microservices and other components in an application.

3 Investigating Tail Latency in DeathstarBench

In this project, we focused mainly on the compose post service, which is responsible for creating and broadcasting posts, as well as updating the user timeline. We chose the compose post workload because of its high level of complexity and concurrency. To analyze the performance of compose post service, we used the wrk2 tool to generate traffic and collected Jaeger traces. Figure 3 shows the cumulative distribution function (CDF) plot of request latencies from the compose post traffic. The median latency for the compose post workloads on the social network application is approximately 10ms and the latencies for the workload ranging from 50th to 90th percentile increases by a factor of 5. However, the system experiences a very high tail latency ranging from 50ms to 200ms.

Most of the cloud outages are due to the very few requests with the high tail latencies [2]. So, it is important to address the tail latencies in large-scale distributed systems. The rest of the paper discusses the approaches taken to analyze the microservices performance and identify the problem areas within the system.

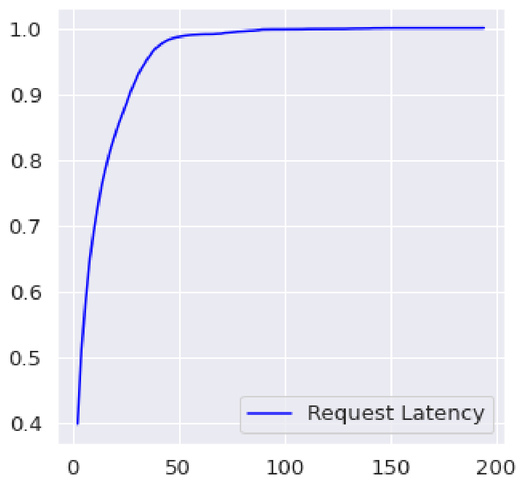


Figure 3: CDF plot of request latencies

3.1 Critical Path Analysis

Critical path analysis is a classic technique in understanding how individual components of a parallel execution impact the end-to-end latency of a system. It involves identifying the critical path, which is defined as, the chain of components that contribute to the overall latency of the system. This is important because any improvement in the latency of a service that is not on the critical path will not have a significant impact on the overall system performance. Therefore, by identifying the services on the critical path, we can determine which services are most critical to the overall performance of the system.

Inspired from the Mystery machine [1], we created a critical path analysis model to detect critical paths per request in the socialnetwork application and determine the percentage of duration each service spent in these critical paths and identify the bottleneck services that affect the system's performance. We used Jaeger traces from Death Star Bench to implement this model, which operates based on the following principles.

1. We established an acyclic dependency graph by exploiting the parent-child relationship between the spans.

2. Start from the root node and get all the children of the node in the order of their start-time of their spans. Based on which of the below relationships (Figure 4) the spans exhibit, construct the critical path respectively. Consider 2 spans s1 and s2 with start-time as t1 and t3, end-time as t2 and t4 respectively. These spans exhibit:

   - **Mutually-Exclusive** relationship when the spans don't overlap i.e if $t1 < t3 \; and \; t2 < t3$. Critical path includes both s1 and s2 and the critical path duration is $(t2 - t1) + (t4 - t3)$
   - **Complete Overlap** relationship if $t1 < t3 \, and \; t2 > t4$. Critical path includes just s1

and the critical path duration is $(t2 - t1)$

- **Partial Overlap** relationship if $t1 < t3$ *and* $t2 > t3$ *and* $t2 < t4$. Critical path includes both s1 and s2 and the critical path duration is $(t4 - t1)$

3. Repeat the same for the child nodes.

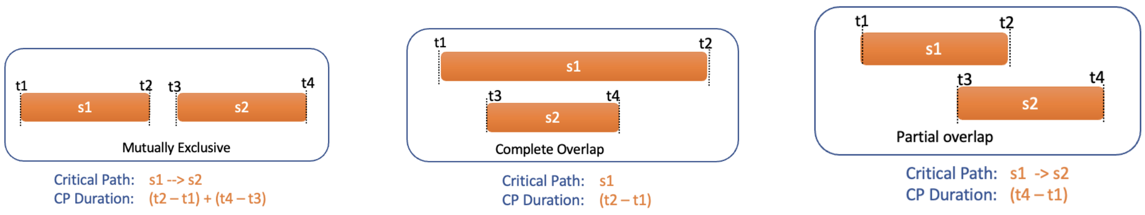


Figure 4: Span relationships

Doing the above critical path analysis for all the requests will provide the following insights:

1. Possible critical paths in this system and the number of requests that took each critical path.

2. List of services in the critical path per request and the amount of duration each service spent on the critical path.

3.1.1 Identifying the bottleneck services

In the case of the compose post load, performing the critical path analysis gives us the services contributing to the critical path and the duration each service spends on the critical path as shown in the Figure 5.

A service with a longer latency could be a sign of a bottleneck or it could indicate that the service is performing more complex operations. In either case, doing a performance analysis on that particular service could provide opportunities for accelerating the service to improve the overall performance of the system.

Another approach is to focus on microservices with a high variation in latency. High variability in latency can lead to unpredictable and inconsistent system performance and so it can negatively impact the performance of downstream services that depends on the inconsistent services. By identifying the microservices with high latency variation, we can investigate the root causes of the variability and take steps to reduce it.

Looking at the Figure 5, compose_post_server, compose_text_client seems to have both high latencies and high variability. So the rest of the paper focuses on analyzing these services to find the areas for improvement.

3.1.2 Analyzing the bottleneck services

In the previous section, we identified the bottleneck service to be compose_post_server. As shown in the code below, compose_post_server creates four async calls using the std::async with launch::async

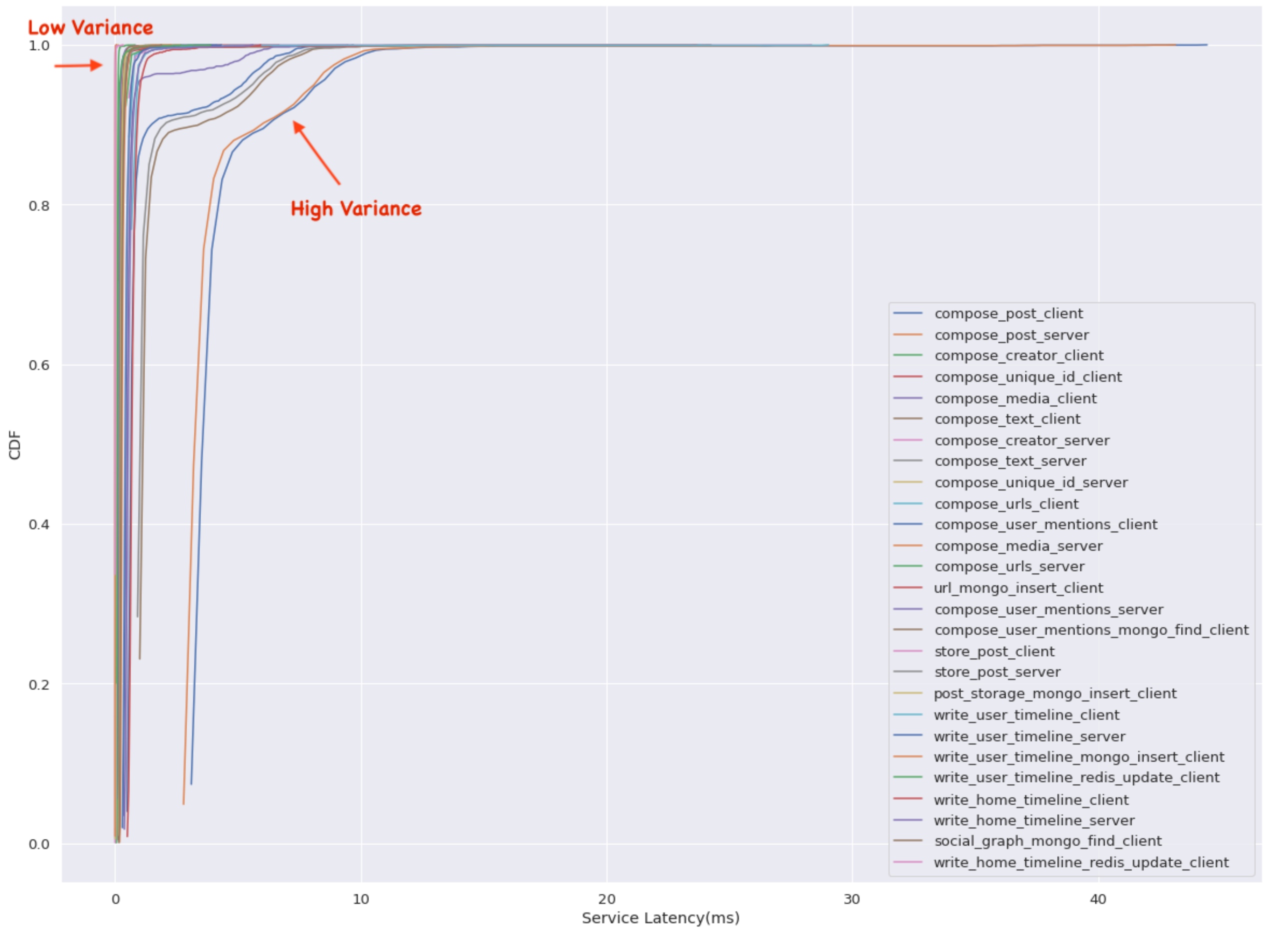


Figure 5: "CDF plot of critical path latency of a service"

mode to *compose text client, compose creator client, compose media client and compose unique-id client*. Running them in the launch::async mode will run these functions asynchronously in a new thread. This function then wait until all the four async calls are complete and gets the return value of the asynchronous functions using get(). The function then creates another async call to the *store post client*, as well as two deferred calls to *write user timeline client and write home timeline client*, using the std::launch::async and std::launch::deferred launch policies, respectively. std::deferred would actually defer the execution of the asynchronous function to when the values are requested. The function then waits until all three async calls are complete and receives the return values.

Looking at the trace timeline of the compose_post_server as shown in Figure 6, there are 2 things that are really striking that has huge impact on the overall latency of this request.

- compose_post_server triggers the first 4 async calls back-to-back. However, the Jaeger trace timeline shows that there is a delay of around $8 - 23$ ms before the asynchronous operations are scheduled, which is a significant delay on the critical path
- The first 4 asynchronous operations in compose_post_server have finished executing and returns

by 29ms. So calling get() on these future objects in the original thread should return the value right away and then the thread should go on to launch 1 async and 2 deferred calls. However, it takes another 37ms for these async and deferred operations to be scheduled, which is a significant delay on the critical path. This delay may be due to factors such as the Scheduling delays from the system scheduler, delays incurred by the asynchronous threads to rejoin the original thread and return the values or contention.

```
void compose_post_server() {
    // Starts the span
    start_span(compose_post_server)

    // Creates 4 async calls
    text_future = std::async(std::launch::async, &compose_text_client);
    creator_future = std::async(std::launch::async, &compose_creator_client);
    media_future = std::async(std::launch::async, &compose_media_client);
    unique_id_future = std::async(std::launch::async, &compose_unique_id_client);

    // Wait until the 4 async calls are done and fetch the values using get()
    unique_id_future.get();
    creator_future.get();
    media_future.get();
    text_future.get();

    // Creates 1 async call and 2 deferred calls
    post_future = std::async(std::launch::async, &store_post_client);
    user_timeline_future = std::async(std::launch::deferred, &write_user_timeline_client);
    home_timeline_future = std::async(std::launch::deferred, &write_home_timeline_client);

    post_future.get();
    user_timeline_future.get();
    home_timeline_future.get();

    // Ends the span
    end_span(compose_post_server)
}
```

Listing 1: compose_post_server

The trace timeline analysis revealed that a considerable proportion of the overall latency (at least 45ms) is caused by scheduling, join and queuing delays. As a result, we were highly motivated to access and quantify the scheduling and join delays and examine their impact on the overall latency. As the majority of tail latency problems arise only under heavy loads, we aimed to investigate how contention affects these delays and in-turn the request latencies.

Furthermore, In the recent years, a significant amount of research has been focused on the creation

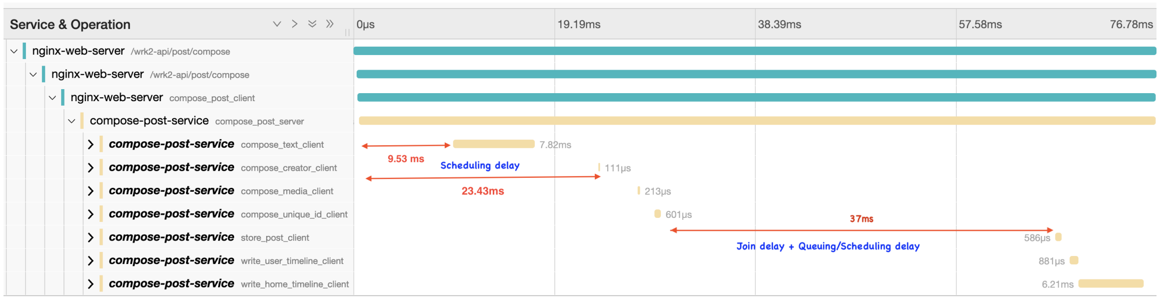

Service & Operation
0µs
19.19ms
38.39ms
57.58ms
76.78ms
nginx-web-server /wrk2-api/post/compose
nginx-web-server /wrk2-api/post/compose
nginx-web-server compose_post_client
compose-post-service compose_post_server
compose-post-service compose_text_client
compose-post-service compose_creator_client
compose-post-service compose_media_client
compose-post-service compose_unique_id_client
compose-post-service store_post_client
compose-post-service write_user_timeline_client
compose-post-service write_home_timeline_client
7.82ms
9.53 ms
Scheduling delay
111µs
23.43ms
213µs
601µs
37ms
586µs
Join delay + Queuing/Scheduling delay
881µs
6.21ms

Figure 6: "Trace timeline"

and enhancement of workload aware schedulers [4] with the objective of considerably improving the throughput and latencies of microservices. The interest in the scheduling delays by these studies strongly motivated us to further examine and comprehend these delays and their impact on the overall performance of the system in detail.

3.1.3 Quantifying scheduling & Join delays

We defined Scheduling delay as the time between when an asynchronous operation is launched and when it actually starts executing. Whereas, a join delay is the time between when an asynchronous operation finishes executing and when the original thread receives the result of that operation. These delays could occur due to multiple reasons such as contention, overhead costs of the async functions, queuing effects etc.

To measure the scheduling and join delays of an asynchronous function, we identified specific points in its execution using the five different timestamps shown in Figure 7.

1. trigger_async_fn: Time when the asynchronous operation is launched

2. fn_start: Time when the async operation begins its execution

3. fn_end: Time when the async operation finishes its execution

4. request_val: Time when the result of the async operation is requested

5. val_received: Time when the result is received by the original thread

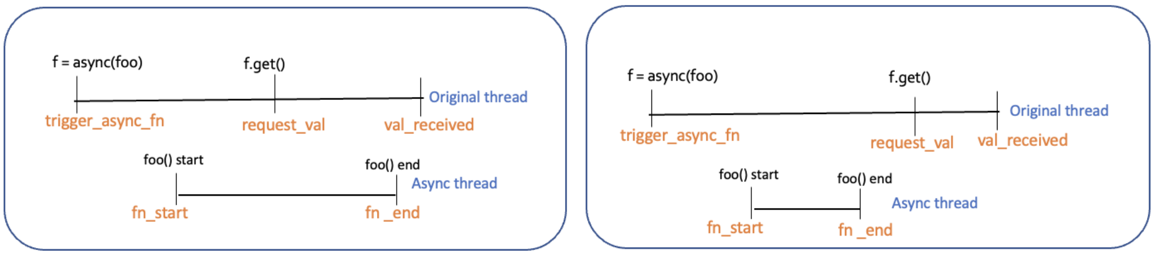


Figure 7: "Quantifying delays"

By marking these boundaries, we were able to quantify the scheduling and join delays associated with the asynchronous function as follows:

- $Scheduling\ delay(launch :: async) = fn\_start - trigger\_async\_fn$
- $Scheduling\ delay(launch :: deferred) = request\_val - trigger\_async\_fn$
- $Join\ delay = val\_received - max(\ fn\_end,\ request\_val\ )$

As async functions launched with launch::deferred mode are not actually until the result of the async operation is requested, therefore the scheduling delay in the deferred async calls is defined as the duration between the time the value is requested and the time the async operation starts executing.

3.1.4 Scheduling and Join delays in the DeathstarBench

We instrumented the socialnetwork application in the deathstarbench by adding spans to mark the boundaries defined in the section 3.2.1 for all the asynchronous functions in this application. We reran the benchmarking tests using the wrk tool on the compose post service, extracted the traces and computed the scheduling and join delays from the newly added spans.

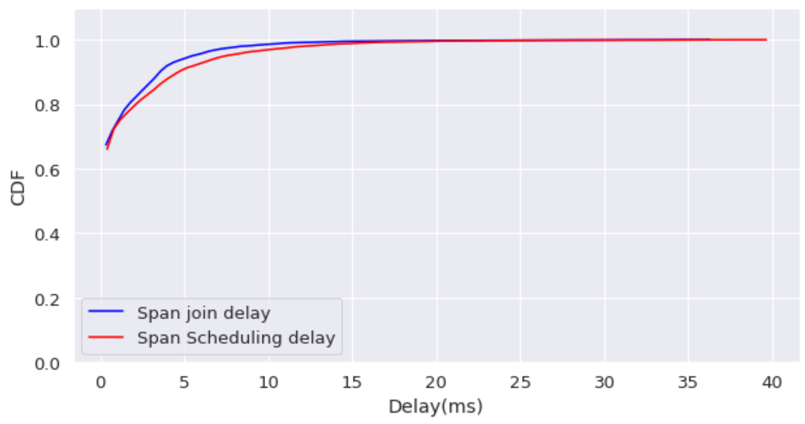


Figure 8: "Quantifying delays"

The CDF plot in Figure 8 depicts the join and scheduling delays of every asynchronous call made in the social network application. Surprisingly, these delays also experience the problem of tail latency. The plot illustrates that 70% of requests have very low delays close to 0ms, but this figure changes rapidly for the 90th percentile of requests, where the delay is amplified by 5 times. Furthermore, the tail latency shows a significant variation ranging from 10-40ms, representing at least 2-8 times increment over the 90th percentile of requests.

The results indicate that there are substantial tail latencies, which signify possible issues with the way scheduling is performed. These findings further emphasize the need to focus more on application-aware schedulers to improve the performance of microservices by improving these delays.

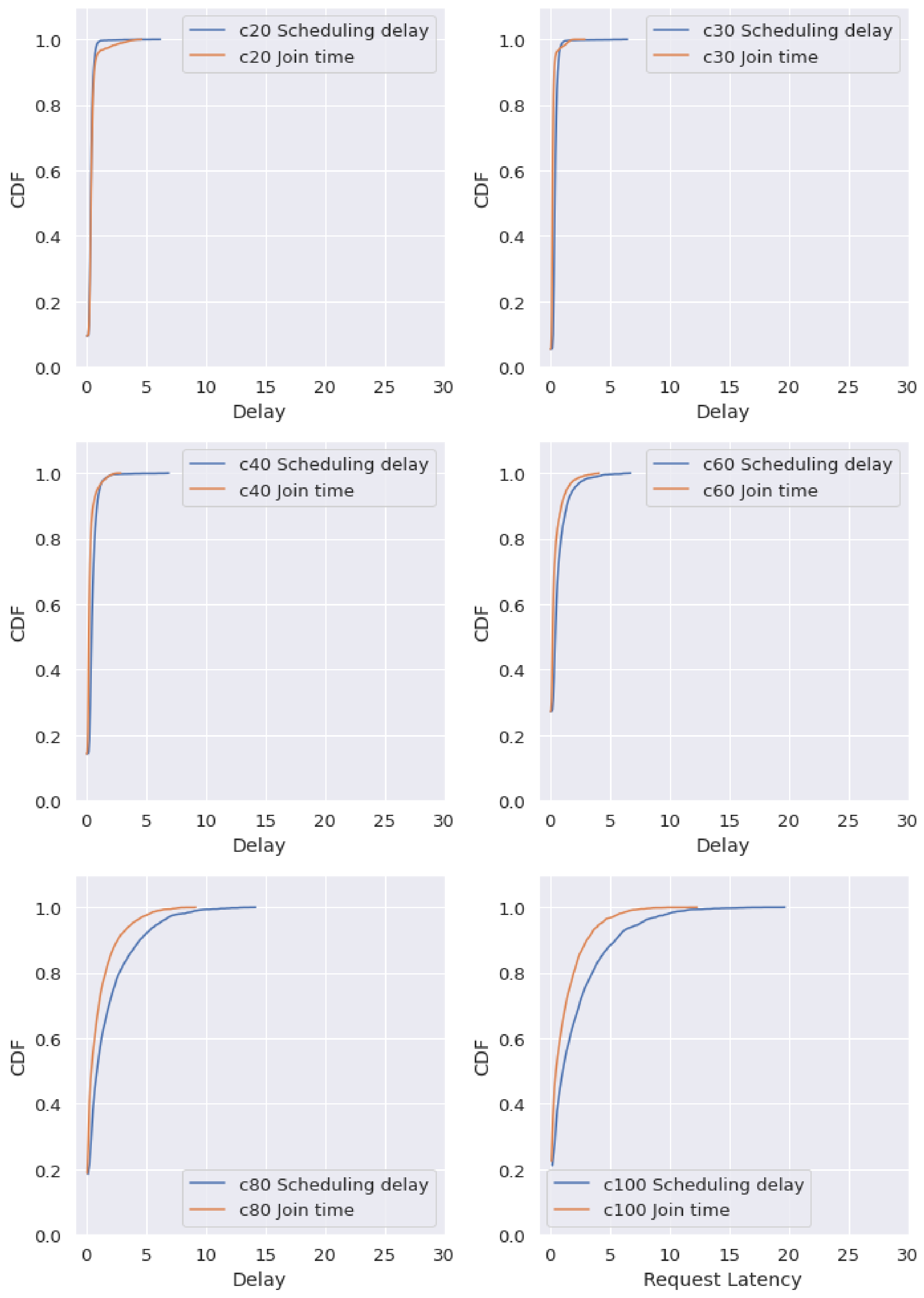


Figure 9: Effects of Contention on the Scheduling & Join delay

3.1.5 Effects of contention on Scheduling & Join delays

One of the main possible factor that contributes to high tail latency is Contention. All the tests we ran till now are around 70-80% of the system capacity. To further isolate the system from contention, we ran deathstarbench in a swarm mode where we split the services onto 3 different nodes where one node handles the front-end nginx service, whereas the other deals with compose_post_service and its child services and storage services on the last node

With this deployment, we did observe a huge drop in the overall latencies. To further understand the effect of contention on the tail latencies, we ran multiple tests at different loads by varying the degree of concurrency of the request load in the wrk2 benchmark suite by tuning the connections parameter. The CDF plots of the join and scheduling delays are as shown in the plots in figure 9.

The tail latencies increases almost linearly with the increase in the load. As we increased the number of open connections from 20 to 80, the tail latencies increased three folds and quadrupled when increased to 100. Even though the service isolation contributed to bringing down the amount of contention within a system, by increasing the load, the contention is unavoidable. Scheduling delays seems to be more impacted than the join delays, which is understandable because of thread management overheads, resource exhaustion, trashing etc.

3.2 Impact of Asynchronous programming models on the performance of Microservices

Microservices are designed to be lightweight, loosely coupled, and independently deployable. They often communicate with each other through APIs and rely on asynchronous programming to handle incoming requests at scale, process them, and send a response back. Microservices in general, are designed to handle high volumes of requests simultaneously without any delays and they use Asynchronous programming models such as promises, futures, and async operations to handle multiple requests simultaneously and efficiently. Therefore, the performance of microservices is highly dependent on how well the underlying asynchronous models are implemented.

Since the Deathstarbench is utilizing the std::async model, we will conduct a detailed analysis of the std::async implementation. In addition, we will perform performance tests to compare it with that of other existing models.

3.2.1 GNU C++ implementation of the std::async model

In C++, Promises and futures are used in conjunction with std::async to communicate between threads as shown in the figure 10. A std::promise object is used to store a value that is produced by the asynchronous task. Whereas, std::future object is a special type of object that represents a value that may not yet be available. It can be used to retrieve the value when it is available, either by polling the future or by waiting for it to become available. is used to retrieve the value from the promise.

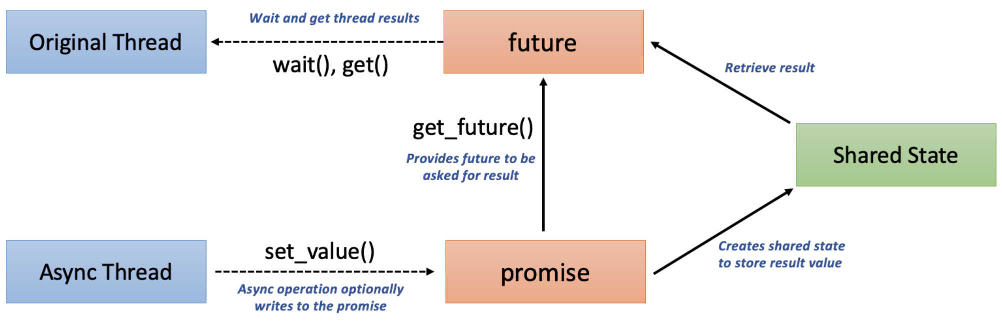

Original Thread
Wait and get thread results
wait(), get()
future
Retrieve result
get_future()
Provides future to be asked for result
Shared State
Async Thread
set_value()
Async operation optionally writes to the promise
promise
Creates shared state to store result value

Figure 10: std::async model

std::async is a function used to create an asynchronous task and it can be called with two launch policies: launch::async and launch::deferred. When called with launch::async, the asynchronous task is launched in a separate thread and returns a std::future object that is associated with this new thread of execution. Whereas, the main thread can continue its execution without waiting for the result. The std::future object can be used to retrieve the result of the asynchronous task.

When the asynchronous task is launched with launch::deferred, it returns a std::future object that is associated with the calling thread of execution. The function is not executed immediately, but only when its result is requested by calling the get() method of the std::future object. This means that the function is executed synchronously with the calling thread of execution.

Under the hood, GNU implements the std::async library using std::thread. std::thread is a lower-level construct that creates a new thread and returns a std::thread object representing that thread. With std::thread, you have complete control over the thread's lifetime and execution, but you are responsible for managing the thread's resources, including joining or detaching it when it's done executing. std::thread implementation is platform dependent and different platforms may use different threading models. For example, On Linux systems, std::thread is typically implemented using POSIX threads(pthreads library), whereas, on Windows systems, it's done using the Windows threading APIs.

On the other hand, std::async is a higher-level construct that returns a std::future object representing the result of an asynchronous computation. With std::async, you don't have direct control over the thread's lifetime and execution, but the implementation handles resource management for you, automatically detaching the thread when the computation is complete. Additionally, std::async provides an easy way to retrieve the result of the computation using the std::future object.

There are 2 limitations that are pretty apparent with the usage of std::async over std::thread. Launching asynchronous task using launch::async mode can be relatively expensive due to the overheads added by the operations such as: thread creation, allocation of resources such as stack space, shared state management overhead which includes creation of std::promise, std::future objects and std::mutexes to synchronize access to the shared state. Also, The overhead may vary depending on the size of the promise object. Additionally when the system is operating at load, launching multiple asynchronous tasks in parallel with launch::async can lead to resource exhaustion. Whereas, when launched in deferred mode, there is no cost associated with the creation of new thread as it doesn't create a new thread, but instead defers the execution of the task to a later time on the same thread. But it still creates a shared state which is local to the thread.

Although std::async with promises, futures and multiple launch modes offers a more straightforward approach to build expandable applications that can manage numerous concurrent tasks, it has substantial overhead. To gain a better understanding of this overhead, we conducted several micro-benchmark tests to evaluate the performance of both std::thread and std::async, and to examine how the promise type can influence the costs incurred by std::async.

3.3 Micro-benchmark tests

We created Micro-benchmark tests to understand the effects of various delays incurred by the async implementation and how it compares to the performance of std::thread. Furthermore, we aimed to replicate the tail latency issues locally outside of the Deathstarbench environment. This section focuses on analyzing the results of the experiments listed below and proposes few suggestions on improving the performance of Async functions:

1. Recreate Deathstarbench tail latencies
2. Performance comparisons of std::thread with that of std::async launched with modes async and deferred
3. Effects of promise sizes on the join delay

Our micro-benchmark tests are designed to measure the join, scheduling delays, and total latency of async calls. To accurately measure these delays, we needed to eliminate any effects that contention could have on the delays. So, we devised a closed-loop experiment where one thread created multiple async calls and another thread consumed the effects of these async operations. Our program starts with a closed loop that increments the incoming rate of the async calls per second and measures the achieved rate and the average latency of the calls and it does that until the maximum rate is achieved and after which the effects of the contention starts to affect the performance. All the further experiments in this section are done at 75% of this load except the load test where we experiment with various percentages of this threshold.

3.3.1 Recreating Deathstarbench tail latencies

Running the micro benchmark tests at less than the threshold capacity, we could see significant tail latency in scheduling and join delays as well as total latency. As shown in Figure 11, the median

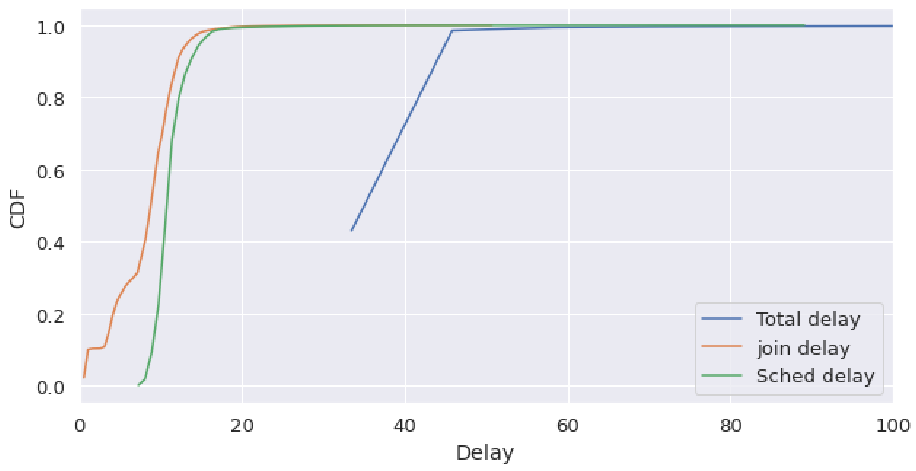


Figure 11: CDF plot of delays in the micro benchmark tests

latency of scheduling and join delay is around 8-10ms, whereas it doubles with the 99th percentile, Added to that, there is a significant variation in both the Join and Scheduling delay among the 99th percentile of the requests ranging from 20ms to 80ms. We can see the resulting effect of both of these delays on the total latency.

Thus, we are able to recreate the problem of tail latencies outside of the deathstarbench test environment using as simple service as a described above. This is interesting because, it highlights the problems and limitations with the current asynchronous approaches in C++ for building microservices dealing with heavy load environments. The further micro benchmarks done in this section analyses the performance of various asynchronous methods available in C++.

3.3.2 std::async vs std::thread

This experiment is focused on the understanding the performance of various thread creation mechanisms available on C++. We ran three different experiments using different async modes described below. All these experiments are run at 75% of the threshold to avoid potential performance degradation due to contention.The results of this experiments are shown in the Fig 8.

- std::async with launch::async mode
- std::async with launch::deferred mode
- std::thread

Looking at these latency plot at request level in Figure 12, it is clear that std::thread has better performance over std::async using launch::async mode. It is expected because std::async in async launch mode involves creating a new thread, shared state and mutexes to manage shared state. All of these can incur significant overhead especially when operating under load. std::async in launch::deferred mode performs better than async mode or std::thread because it doesn't involve creation of the new thread and all the execution occurs in the context of the original thread making

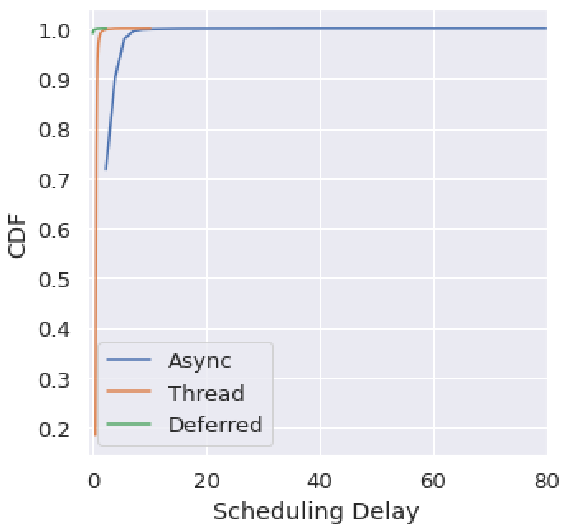


Figure 12: Performance analysis between async methods in C++

it much faster but at the limitation of lack of concurrency. Thus the scheduling delay of deferred mode is very small followed by std::thread and std::async. std::async has higher scheduling and join delays because of the shared state management of the promises and future objects.

3.3.3 std::async with different promise sizes

This experiment is focused on understanding the correlation between the promise types and the join & scheduling delays. We rerun the same experiment but change the promise type being returned by the asynchronous operation. We ran three different experiments with Asynchronous function returning promise type of long long, vector of size 10k and None.

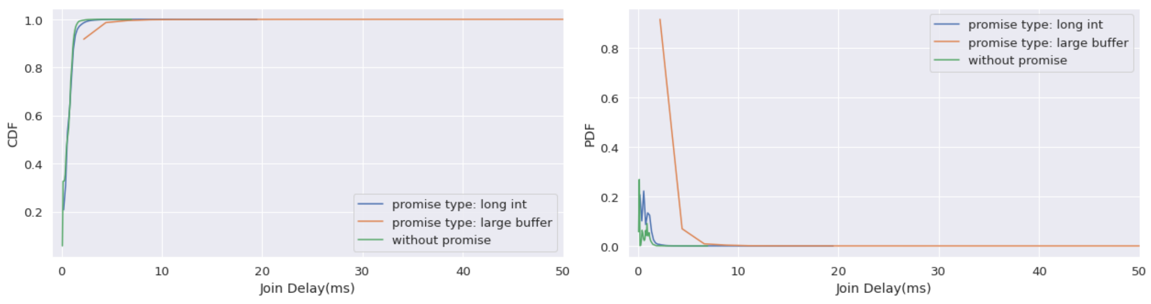


Figure 13: Effects of Promise type on the Join delay

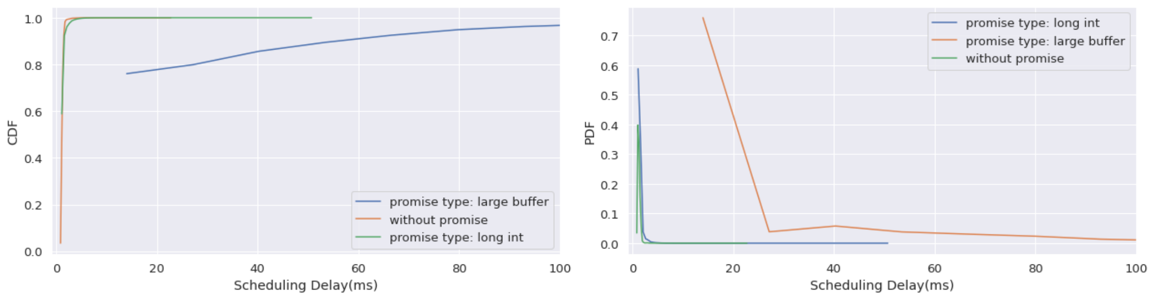


Figure 14: Effects of Promise type on the Scheduling delay

It is very clear from the Figure 13 and Figure 14, that there is a strict correlation between the promise size and the Scheduling and Join delays. Looking back on the scheduling delay, it is the duration between the launch of std::async function and the start of the execution of the asynchronous operation. This duration includes the thread creation, allocation of the required resources like memory, stack etc, initializing the shared state promise object, initializing mutex to synchronize the shared state access and the queuing delay at the scheduler. Whereas, the Join delay is defined as the duration between when the timestamp when the value is set to the promise object and when it's received by a thread through get() operation. This duration includes taking a lock on the shared state, setting the value, releasing the lock, receiver thread to be scheduled on the CPU and fetch a read lock and receive the value of the promise. This experiment shows the effects of initializing, locking, writing to the shared state object on the Scheduling delay and how increasing the size of promise object increases the overall delay as expected. It's much more strikingly clear between the scenario where the asynchronous function doesn't return any value compared to when it returns a 10k buffer.

3.4 Suggestions for improving the performance of Async calls

Thus from these micro benchmark tests, we can say that the std::async in launch::async mode has significant overheads associated with the management of shared state and can lead to resource exhaustion and trashing when run on load in GCC and LLVM environments. One of the simpler ways of reducing the overheads of async(launch::async) is to use them with the thread pool. In this way, not only we can have control over the number of threads running in the system, we can also avoid the thread creation and deletion costs. Although there will still be some overhead in maintaining the thread pool, but it's still minimal compared to the thread creation and maintenance costs. Microsoft MSDN does exactly the same, it uses the task class from the Windows thread pool as its scheduler instead of the concurrency runtime to avoid these overhead costs and resource exhaustion issues.

One other way to fix this is to use much thinner threading models like C++ co-routines(similar to golang's co-routines). Co-routines are much lighter versions of threads, they don't reserve entire memory stack for its operation and can have thousands of co-routines triggered in the system without exhausting memory.

4 Conclusion

The objective of this project is to identify the root cause of high tail latencies in the social network application within the DeathstarBench suite. To accomplish this, a critical path analysis model was utilized to detect bottleneck services. Additionally, micro-benchmark tests were developed to analyze the performance effects of different asynchronous implementations. It was determined that the use of std::async with launch::async mode is one of the factors contributing to the tail latency. To address this issue, it is recommended that the std::async model be used in conjunction with a thread pool model to minimize the overhead caused by the creation of new threads and mitigate resource saturation. By implementing these measures, it is possible to improve the overall performance of the application.

References

[1] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. *The mystery machine: End-to-end performance analysis of large-scale internet services.* 2014.

[2] Jeffrey Dean and Luiz André Barroso. *The tail at scale*, volume 56. ACM New York, NY, USA, 2013.

[3] 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.* 2019.

[4] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. *ghost: Fast & flexible user-space delegation of linux scheduling.* 2021.

[5] Andrea Janes, Xiaozhou Li, and Valentina Lenarduzzi. Open tracing tools: Overview and critical comparison. *arXiv preprint arXiv:2207.06875*, 2022.