FuncMem : Reducing Cold Start Latency in Serverless Computing Through Memory Prediction and Adaptive Task Execution

Manish Pandey
Kyungpook National University
Daegu, South Korea
manishpandey@knu.ac.kr

Young-Woo Kwon
Kyungpook National University
Daegu, South Korea
ywkwon@knu.ac.kr

ABSTRACT

Because serverless computing can scale automatically and affordably, it has become a popular choice for cloud-based services. However, despite these advantages, a serverless architecture is not suitable for applications requiring instantaneous executions because of cold starts. Existing techniques primarily focus on extending keep-alive time or pre-warming containers, which alleviate performance issues for specific serverless functions but introduce additional overhead to the architecture. To address these issues, we introduce *FuncMem* , a methodology designed to manage memory resources by prioritizing non-blocking asynchronous requests in a serverless architecture. First, *FuncMem* predicts and reduces excessive memory requirements for serverless functions. Second, it dynamically reschedules functions within an invoker, creating an adaptive task queue at runtime to mitigate cold starts and reduce wait times. We implemented our approach in OpenWhisk, a popular open-source framework, and evaluated it with multiple FaaS applications. Through comprehensive evaluations, we show that *FuncMem* achieves significant performance improvements, including a 63.48% reduction in cold start latency, a 46.98% decrease in memory allocation, a 54.93% reduction in cumulative execution time, a decrease in average waiting time from 5.22 seconds to 2 seconds, an increase in average throughput from 0.76 to 1.63 functions per second, and a decrease in average initialization time from 0.16 seconds to 0.7 seconds. Our results show the effectiveness of *FuncMem* in terms of latency and resource usage.

KEYWORDS

Serverless Computing, Job Scheduling, Memory Estimation, Non-blocking Requests, and Cold Starts

ACM Reference Format:
Manish Pandey and Young-Woo Kwon. 2024. *FuncMem* : Reducing Cold Start Latency in Serverless Computing Through Memory Prediction and Adaptive Task Execution. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605098.3636033

1 INTRODUCTION

Serverless computing is rapidly gaining popularity in the cloud, with an estimated 50% of global enterprises expected to embrace serverless architecture by 2025 [3]. This cloud paradigm enables developers to focus on application logic rather than deployment, as a cloud provider manages the underlying infrastructure. The architecture eliminates operational overhead and offers automatic scalability in a pay-per-use billing model [7]. This serverless computing approach offers businesses an enticing option to enhance their operations and profitability by providing an agile and dynamically allocated resource environment.

In contrast to traditional microservice architecture, where services typically remain running, a serverless architecture needs to create the process, execute a task, and then remove the process after a specified keep-alive time[13]. The architecture creates a sandbox environment, such as a container or Virtual Machine, to run an individual function [10]. These functions are stateless, and the cloud provider treats each incoming request as a new black-box request. Such features allow the cloud provider to dynamically scale the applications across multiple servers or regions without affecting the function's behavior.

Despite the current serverless framework's advancements in easy development and deployment, the issue of cold starts still needs to be solved. Existing solutions primarily revolve around extending keep-alive duration [23] or pre-warming containers [16], which are reliable for a single function but introduce performance overheads. These approaches rely on historical request patterns; however, over half of serverless applications lack discernible patterns or have a limited invocation dataset [23]. Moreover, cloud tenants often over-allocate memory configurations, resulting in prolonged container lifetimes or frequent creations of new containers [4].

Background data processing and other asynchronous tasks are gaining popularity in serverless computing [9]. Serverless provides non-blocking request features that offer a quick acknowledgment response to reduce latency for these asynchronous tasks. Even though serverless architecture separates asynchronous and synchronous requests, existing techniques merge all requests in the same queue in an invoker. As platforms often use a first-in, first-out (FIFO) execution pattern to process an invoker queue, functions frequently suffer from longer waiting times and cold starts due to limited memory allocation to an invoker.

In this paper, we present *FuncMem* , that minimizes excessive memory allocation and reschedule function execution and manage warm function deletion to improve overall serverless performance. First, using a parallel running simulator, the tool calculates the essential memory usage of the deployed function. Then, dynamically pick and remove containers on the invoker node to decrease cold starts, prioritizing request methods, warm containers, function deadlines, and function availability. To evaluate the effectiveness of *FuncMem* , we implemented it with FaaS applications on the real system and conducted a comprehensive performance analysis, comparing it with a Vanilla OpenWhisk system. Our evaluation reveals that the *FuncMem* offers remarkable benefits, including reduced cold starts, reduced memory usage, improved execution time, shorter waiting and initialization durations, and improved throughput in serverless environments.

This paper's contributions include:

- A novel methodology for serverless function scheduling. It features a simulator to reduce over-memory allocation and a scheduler that reschedules function invocation in an invoker to reduce cold-start latency.
- *FuncMem* , a serverless infrastructure that implements the presented approach in the third-party framework.
- An empirical evaluation of *FuncMem* , incorporating its implementation within OpenWhisk alongside popular Function-as-a-Service (FaaS) applications.

The rest of this paper is structured as follows. In Section 2, we provide technical background and present our motivations for understanding our contributions. We discuss in detail our design assumptions and implementation of our methodology in Section 3. We then evaluate our methods in Section 4, discuss their applicability in Section 5, and compare them with related state-of-the-art approaches in Section 6. Finally, we outline future research directions and provide concluding remarks in Section 7.

2 BACKGROUND AND MOTIVATION

2.1 Background

Serverless architecture is a new approach in cloud computing that enables developers to perform event-driven operations without managing servers [12]. In traditional cloud architecture, once an application is deployed, it runs indefinitely. In contrast, a serverless architecture operates and removes applications after a specific keep-alive time. To deploy an application on this platform, cloud tenants should choose a supported programming language, write the code within a primary function, and upload it with a designated action name. Subsequently, they can trigger the application by invoking it using the platform's API or command-line tools like openwhisk-cli[1] or faas-cli[2]. In this paper, we build our approach on OpenWhisk[3], an open-source serverless platform. The platform consists of a controller which is responsible for managing requests and responses between users, as well as the internal architecture, which encompasses modules for authentication, authorization, load balancing, monitoring, logging, an API gateway, and the selection of worker nodes; and an invoker, which oversees the runtime management environment, including containers, queues, and other tasks.

The platform creates a sandbox environment for executing the function in a Docker container or a virtual machine (VM). This sandbox contains the application code, runtime libraries, and configuration to run the application. Assume the client invokes a function for the first time or after a period of inactivity (usually 600 seconds in Openwhisk). In such a scenario, the serverless platform builds a new sandbox, downloads and extracts the code, launches the container, copies the runtime based on the user-supplied metadata, schedules it to the appropriate worker node, and executes the code. This process is known as a cold start [5] [21] and has a longer running time. After execution, the architecture pauses the container for future reuse with an idle timeout. When a function is called again before idle timeout, a serverless platform reuses the existing sandbox, known as a warm start. As the function does not need to go through the initialization phase, it executes immediately, resulting in a shorter response time. A serverless platform does not charge for the cold start time; however, the cold start has a significant performance impact in serverless applications [4].

[1]Openwhisk-cli: https://github.com/apache/openwhisk-cli
[2]faas-cli: https://github.com/openfaas/faas-cli
[3]https://openwhisk.apache.org/

Serverless applications can invoke requests in two ways: blocking and non-blocking.

- **Blocking requests:** An application waits for a response to a request to be returned before continuing other tasks. For example, when a serverless application writes data to a database, it waits for the database to complete write operations.
- **Non-blocking requests:** An application immediately sends an acknowledgment message to a client indicating that the request has been received, but continues other tasks. The application can then periodically poll the system's status to obtain an invoked result.

The serverless platform allocates specific memory to the worker node (i.e., invoker nodes), determining the number of functions each node can run. Invoker maintains two pools: a *pre-warm pool* and a *busy pool*. The pre-warm pool consists of containers ready to accept function invocations, while the busy pool contains currently running containers. When containers in the busy pool finish their execution, the platform moves them to the pre-warm pool with a specified keep-alive time. If the platform receives a request and a container is available in the pre-warm pool, it moves the container to the busy pool.

2.2 Motivation

Serverless platforms allow cloud tenants to allocate memory and CPU to individual functions. The default memory allocation of the popular serverless platform is AWS Lambda 128 MB[5], Google Cloud Functions 256 MB [6], Microsoft Azure Functions 128 MB [7], IBM Cloud Functions 256 MB [8], and Alibaba Cloud Function Compute 256 MB. These platforms increase the price with an increase in memory allocation. For example, on AWS Lambda, the cost is 2.1 x 10^{-9} per millisecond for 128 MB of memory, while it rises to 1.667 x 10^{-7} per millisecond for 10 GB of memory[9].

In AWS Lambda, computing a prime number with 1,024 MB is 10-fold faster than computing with 128 MB over 1000 invocations [10]. Allocating higher memory is helpful for memory-intensive functions; however, it might not be useful for every task, as it is directly related to cost. The performance stats of AWS Lambda

[4]https://thenewstack.io/what-aws-lambdas-performance-stats-reveal
[5]https://aws.amazon.com/lambda/
[6]https://cloud.google.com/functions/
[7]https://azure.microsoft.com/en-gb/products/functions/
[8]https://cloud.ibm.com/functions
[9]https://aws.amazon.com/lambda/pricing/
[10]https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html

reveal that the average function uses around 65 MB of memory, and the median function uses only 29 MB [4]. To understand the allocation and utilization of serverless functions, we analyze the Azure Functions Trace 2019 datasets [23]. This dataset provides the workload production trace of Microsoft's Azure functions.

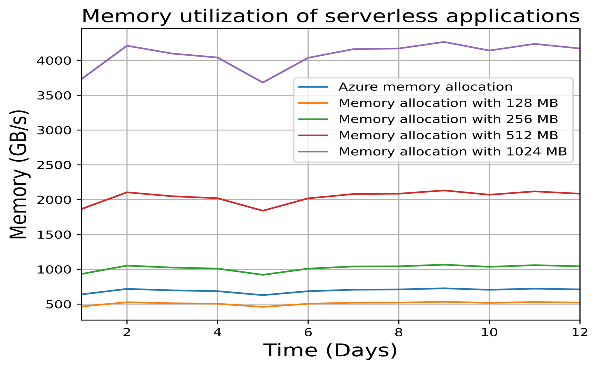


Figure 1: Memory utilization of serverless applications per seconds

Figure 1 shows the dynamic changes in memory utilization within the Azure dataset over 12 days about variations in memory allocation. Each color line represents a specific memory unit allocated to Azure functions, with green signifying an allocation of default memory (i.e., 256 MB) per function. If we assume that the user sticks to default memory allocation, the service provider should allocate an average of 1000 GB per second to execute these serverless functions. If the user increases their allocation to 512 MB (as indicated by the red line), the cloud provider should double their allocation, i.e., 2000 GB per second for execution. These excessive memory allocations drive up costs for the cloud provider and restrict the number of containers operating in prewarm and busy pools. Consequently, this leads to longer waiting times and a higher rate of warm container terminations in the busy pool, ultimately resulting in an increased number of cold starts.

The frequent occurrence of cold starts contributes to extended initialization times for container functions, resulting in resource contention, performance degradation, and system instability. The graph also indicates that the memory allocation of the Azure functions (i.e., blue line) is less than the default allocated memory. Reducing these over-allocation memories can improve the performance of the serverless platform [23]. This leads to our first research question, **R1: How can serverless platforms adopt automated techniques to optimize the memory usage of functions reliably and cost-effectively without compromising the execution time of the function?**

The proliferation of IoT devices has led to a significant increase in data generation, which presents challenges for processing on resource-constrained devices. Due to their inherent limitations, IoT devices often offload data to the cloud or edge server. However, the delay in receiving responses from the cloud can harm the timely delivery of services. Adopting serverless, non-blocking request executions can be a favorable choice as it can instantly receive the acknowledgment response. Later, these devices can inspect the results using the acknowledgment information.

To understand how the current serverless architecture handles invocation requests in high volumes, we conducted an experiment involving 10 functions in Vanilla OpenWhisk (i.e., an unaltered version of the Apache OpenWhisk serverless framework). Each function had an execution time ranging from approximately 0.9 seconds to 1.5 seconds. We invoked these 10 functions in parallel every 12 seconds. We execute functions on a single computer with 32 GB memory, an Ubuntu operating system with one invoker node. In Openwhisk, each invoker node operates in isolation and does not share resources with other nodes. In total, we performed 400 invocation requests with non-blocking requests. Surprisingly, only six functions out of 400 invocations experienced a warm start, even though we invoked functions every 12 seconds. We will delve into the reasons behind these cold starts in Section 3.1. We also found that waiting times tend to increase exponentially.

In other experiments, we tested invoking blocking requests when more than 20 functions were waiting in the queue. Until the timeout period, the framework treats functions as blocking categories and then moves them to non-blocking categories. On the contrary, the same blocking requests execute within a few seconds when the queue is empty. This observation demonstrates that while the serverless framework separates requests during invocation, the execution process does not prioritize functions based on their request type. As blocking requests are vital for latency-critical applications, providing them with higher priorities is essential to ensure timely execution. Minimizing wait time is crucial, as it directly impacts the performance of the serverless architecture. This raises our second research question:**R2: How can we minimize waiting time and prioritize requests to prevent cold start in serverless architecture?**

3 DESIGN ASSUMPTIONS

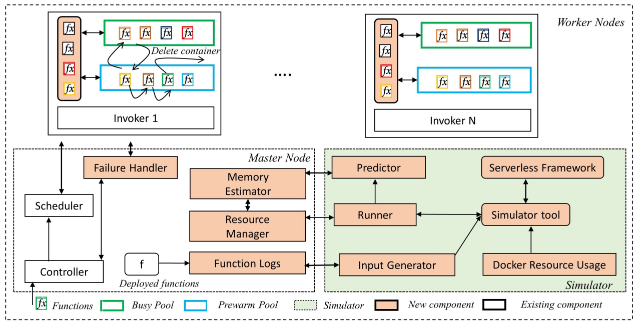


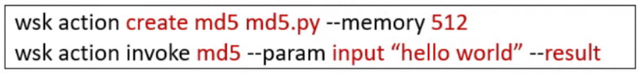
Figure 2: Serverless workflow

This section explains how the design of *FuncMem* can effectively address the two research questions (R1 and R2). Figure 2 shows the induced architecture, where the orange color indicates new components added to the current architecture. We classify our methodology *FuncMem* into two categories, as explained below.

3.1 Memory Estimator Module

The goal of *R1* is to minimize the excess memory allocated to the serverless function. We developed a serverless simulator that runs parallel to the existing serverless platform to achieve this objective. In Figure 2, the highlighted background green color indicates the components of the simulator. The simulator runs the existing deployed serverless function with different memory allocations and identifies the appropriate memory configuration for each function.

The following are the steps to run the simulator.



```
wsk action create md5 md5.py --memory 512
wsk action invoke md5 --param input “hello world” --result
```

Figure 3: Serverless command to create and invoke

Figure 3 shows commands to create and invoke function "*md5*" with 512 MB memory and the input parameter "hello world" as a blocking request. Our simulator focuses on extracting and modifying the function metadata, such as the red text command. We can extract this metadata using a platform Command-line Interface (CLI) or adding API to the action modules of the framework. Once we receive the metadata, we store it in *Function Logs*, where we identify its datatypes. Datatypes can be strings, integers, or floats, and based on the types, *Input Generator* employs techniques such as insertion or concatenation for augmentation. Additionally, the *Input Generator* conducts memory augmentation, creating datasets spanning from 96 MB to 4 GB, enhancing the scope of input variability for analysis.

Runner generates scripts to invoke functions, execute them on *simulator tool* and save the output results. It acquires input data generated by the *Input Generator* and transforms it into executable commands, executed as shell commands. Additionally, we use a shell script command that runs in parallel on the command line and is responsible for extracting memory usage information from Docker containers for each serverless function. A serverless Docker function starts with a framework keyword, such as '*wsk*', followed by a version name and an action name. The *Docker resources usage* component provides insight into the memory usage of each Docker function. By combining these components, we can extract memory usage data for each Docker image at intervals of 100 milliseconds. The *Simulator tool* is the running instance of the serverless framework that runs parallel to the existing framework.

The serverless platform provides CLI tools (like *wsk-cli*) that can extract the function execution time, invocation start and end time, and latency, along with other helpful information such as kind, timeout, limits, memory, logs, and initial time. We use a bash script to calculate, merge, extract the results, and save them in *Resource Manager* for prediction.

Prediction Problem and Approach: Serverless functions often integrate with external cloud services and databases [9]. Running these functions for analysis directly impacts the user's costs. To address this issue, we categorize the functions into two types: *dependent* functions and *independent* functions. To accomplish this, we scan the function files and search for remote plugins like "*HTTP*", "*resources*", and "*S3*" specifically for Python. We can apply similar methods to other programming languages. We perform pre- and post-memory identification for independent and dependent functions as part of this process.

Analysis and predictor: We analyze our techniques with a FaaS application [14] and algorithm functions [2]. We observed no significant differences in memory utilization between the cold start and the warm start for functions. However, it is important to note that cold start instances take longer to initialize the container and the runtime environment, resulting in higher memory consumption for an extended duration. Additionally, we encountered scenarios where serverless function containers required only a small amount of memory, but attempting to allocate that minimal memory proved insufficient for effective execution. For example, a web server function utilized approximately 30 MB of memory, but designating precisely 30 MB was inadequate. Therefore, we aimed to identify the minimum memory allocation to ensure smooth function execution without dropping the execution time. We set the minimum memory allocation to 96 MB for testing purposes. After the runner executes the function, we can calculate memory usage by examining the memory logs for different input scenarios and varying memory allocations.

$$M_f = \operatorname{argmin} M \forall M' \in [M_{min}, M_{max}] \mid E(M) = E(M') \quad (1)$$

where M_{min} and M_{max} represent the minimum and maximum memory limits, respectively. $E(M)$ is the function's execution time with memory M. The equation aims to find the minimum memory limit M_f such that the function's execution time remains constant for any memory limit between M_{min} and M_{max}.

For dependent functions, the minimum memory limit is

$$M_f = Q_3 + k \times (Q_3 - Q_1) \quad (2)$$

M_f is the minimum memory limit threshold for memory that is more than the average use, Q_3 and Q_1 are the third quartile and the first quartile, respectively, and k is a scaling factor set to 1.5 [11].

Evaluation: We evaluated the effectiveness of our approach with a machine learning function. We trained the ML model on the iris dataset[12] and uploaded it to an Amazon S3 bucket. We create a Docker image to support the function library and upload it to the Docker hub. We perform experiments on the Vanilla OpenWhisk system with default memory (256 MB) and our estimated threshold memory. Our method assessed a required memory of 101 MB, which is less than 50% of the allocated memory. In practice, the memory used during run-time was only around 53 MB, demonstrating the efficiency of our approach in optimizing memory usage.

Failure handling: Failure to predict the appropriate memory of a functioning container may result in the termination of the container. In the event of a failure, we promptly update the function with a default memory allocation and prioritize its execution in the scheduler. This proactive approach aims to mitigate potential performance issues and ensure the smooth functioning of the application. Our approach is more helpful to non-blocking requests to ensure minimal disruption to critical operations.

3.2 Invoker scheduler Module

The goal of **R2** is to minimize the wait time and reduce the cold start in the serverless architecture. To do that, we focus on function scheduling within an invoker node. Figure 4 illustrates the current approach and a proposed approach to running functions within an

[11]https://github.com/alexcasalboni/aws-lambda-power-tuning
[12]Iris dataset: https://gist.github.com/netj/883620

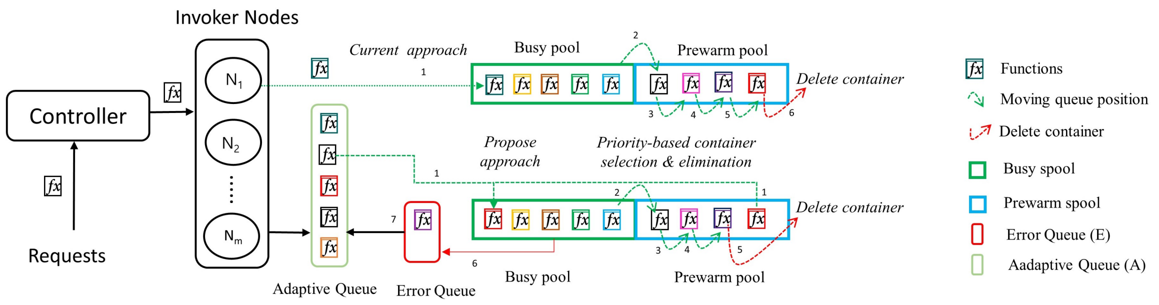


Figure 4: Proposed approach for invoker scheduler

invoker. When a user initiates a function, the controller directs it to an appropriate invoker node. Initially, an invoker assesses the availability of a container in the prewarm pool. If one is available, it executes the function when sufficient memory becomes accessible in the busy pool to support the execution. Following the completion of a function's execution, the invoker moves the container to the prewarm pool. However, if the prewarm pool's memory reaches maximum capacity, it removes the oldest container from the prewarm pool to create space. The current approach employs a FIFO (First-In-First-Out) execution method, which can result in significant wait times and cold starts when the invoker receives a high volume of function requests. Moreover, the invoker consistently deletes old containers that could potentially be reused. Also, the serverless framework treats blocking and non-blocking requests similarly, so we want to differentiate and modify their handling.

Algorithm 1: Function Selection and Deletion Algorithm

```
 1: Input: Incoming request($f$) = $f_1, f_2, ..., f_n$ and deadline ($D$) = $d_1, d_2, ..., d_n$
 2: Requests type($R_t$) = $Blocking(R_b), Nonblocking(R_n) | f_i \in R_t$
 3: $b_m, p_m$ are current memory usages of the busy pool and prewarm pool
 4: Busy pool functions ($B_b$): $f_i \in f, |, \sum_{i=1}^{k}$ memory$(f_i) < b_m$
 5: Prewarm pool functions ($B_p$): $f_i \in f, |, \sum_{i=1}^{k}$ memory$(f_i) < p_m$
 6: Ouput: $Next(f_i) | f_i \in f$
 7: Process:
 8: Adaptive queue: $Q = (f_1, f_2, ..., f_n), Q \leftarrow Q + f$
 9: $Error(E) \leftarrow f_i | f_i \in f$
10: while $f_i$ in $f$ do
        $Q \leftarrow Q + f_i$                    ▷ Insert each incoming $f_i$ request to Q
        while $b_m$ < default_memory do
        if $\exists f_i \in Q : f_i \in B_p$ then
            $F_W = \begin{cases} 1, & \text{if } f_i \in p_m \\ 0, & \text{else} \end{cases}$   $F_E = \begin{cases} 1, & \text{if } f_i \in E \\ 0, & \text{otherwise} \end{cases}$
            Time remaining($T_r$) =
            $((time\_in\_system)/timeout) * 100$
            $Priority(P(f_i))$ =
            $\begin{cases} 0.5 * F_W + 0.25 * T_r + 0.25 * F_E, & \text{if } = R_n \\ 0.5 * F_W + 0.5, & \text{if } f_i = R_b \end{cases}$
        $maxF = \max_{f \in Q} f.p$
        $B_b \leftarrow$ Execute:$(maxF), E \leftarrow f_i$
        $deque(Q(maxF))$
        $delete(B_p), f_i | f_i \leftarrow min\_deadline(B_p) | f_i \notin Q$
```

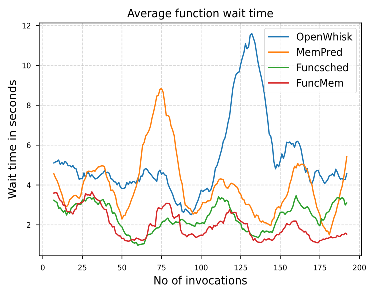
Algorithm 1 outlines the operational principles of the *FuncMem* scheduler. It begins by extracting essential properties such as deadlines and request types for the functions invoked within the platform, along with tracking the current memory usage for both the busy and prewarm pools. We introduce two novel queues in the invoker module: the Adaptive Queue (Q) and the Error Queue (E) to enhance the management of incoming requests. The Adaptive Queue (Q) operates on a First-In-First-Out (FIFO) basis and is a repository for incoming function requests. Each function in Q is assigned a deadline determined by the controller. Notably, if a function in Q already exists within the prewarm pool or the error queue, it is granted a priority weight of 1; otherwise, its weight is set to 0. The algorithm periodically recalculates priorities, estimating remaining execution times based on allocated deadlines. Priority assignment differs for blocking and non-blocking requests: blocking requests receive priority solely based on their weights, while non-blocking requests have their priorities distributed among weights, deadlines, and error status. Specifically, non-blocking requests are allocated 50% of their priority based on weight and 25% each for deadlines and error handling. Blocking tasks are latency-critical tasks and require higher priority than non-blocking tasks. The algorithm's core objective is to identify and execute the function with the highest priority within the warm pool while removing the function with the shortest deadline from the prewarm pool, optimizing the overall system performance.

4 EVALUATION

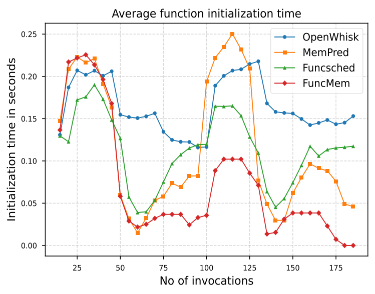
In this section, we provide insights into the implementation and evaluation of *FuncMem* , aligning with Research Questions *R1* and *R2*. We integrate *FuncMem* into the OpenWhisk open-source serverless framework where we modify controller and invoker modules. We experiment on an Ubuntu 20.04 environment with 32 GB of memory and 12 CPU cores.

4.1 Memory Prediction

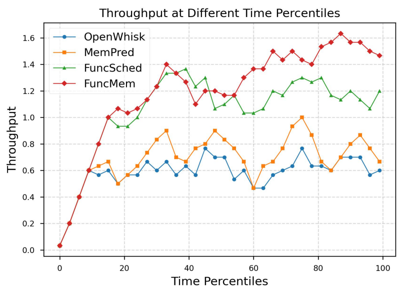
To evaluate *R1*, we select seven popular Function-as-a-Service (FaaS) applications [14]. To simplify the implementation, we coded the application in Python. Notably, six of these applications required various libraries for their execution. To accommodate this, we leveraged Docker images and stored them on DockerHub, initializing these images as needed during invocation. We maintained default



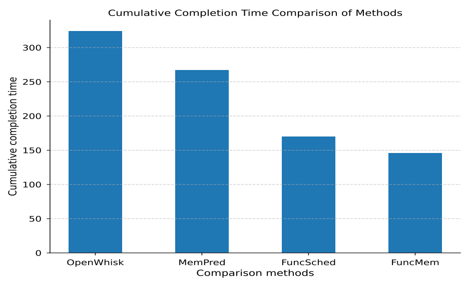
(a) Average function wait time



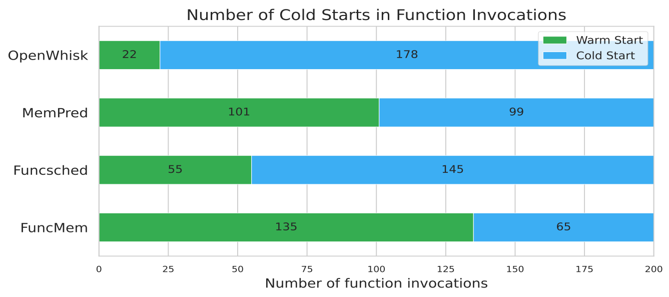
(b) Average function initialization time



(c) Average throughput



(d) Cumulative competition time



(e) Number of cold start

Figure 5: Performance Metrics

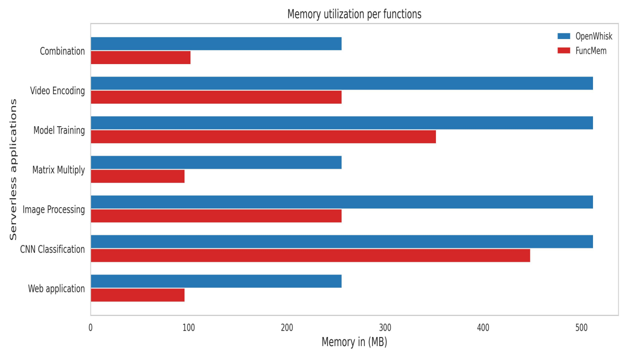


Figure 6: Memory utilization of FaaS applications

memory settings of 256 MB and 512 MB. These functions had average executions ranging from 300 ms to 30 seconds. We executed each application for 20 iterations, focusing exclusively on cold starts in both the baseline OpenWhisk environment and with the *FuncMem* implementation. Figure 6 displays a comparison of memory allocations per function between the baseline (OpenWhisk) and *FuncMem* , revealing that *FuncMem* achieves an impressive average memory reduction exceeding 46.98%, while consistently maintaining the execution times for individual functions.

4.2 Scheduler

To answer Research Question *R2*, we chose to use the FaaS application described in [14] along with 30 different functions from the algorithm repository [2]. Given the lack of a comprehensive dataset for large-scale serverless functions, these selections provided a suitable basis for our evaluation. In our comparative analysis, we pitted *FuncMem* against various other implementations, including vanilla OpenWhisk (*OpenWhisk*), vanilla OpenWhisk augmented with our memory prediction (*MemPred*), and vanilla OpenWhisk enhanced with our scheduler (*FuncSched*) alone. Since Serverless does not have a specific input pattern [23], we generated 200 random invocations for these functions as input while preserving a consistent invocation pattern across all scenarios. We executed these input invocations sequentially via a bash script, resulting in an average completion time of 2 minutes and 52 seconds. Additionally, we adjusted the default invoker memory support from 1024 MB to 3072 MB to facilitate the operation of multiple containers.

Figure 5(a) shows the average wait time for the function as the number of invocations increases. Wait times mean the time that the functions wait until a resource is available for invocation. In the current approach, waiting time peaks after a certain invocation due to limited memory allocation on an invoker node, adversely affecting serverless application performance. While applying the *MemPred* memory prediction method offers some relief, a more effective approach involves modifying the *FuncSched* scheduler. *FuncMem* significantly reduces the average wait time from 5.22 seconds to 2 seconds, greatly enhancing serverless application performance.

As seen in Figure 5(b), the average initialization times, known as cold start times, increase as the number of invocations increases. When we compare them to *OpenWhisk*, both *MemPred* and *FuncSched* exhibit lower initialization times. However, they reach a peak at certain points, while the *FuncMem* demonstrates consistently lower initialization times throughout, outperforming the existing method by reducing initialization time from 0.16 seconds to 0.7 seconds.

Figures 5(c) show the average throughput in different time percentiles. Notably, *OpenWhisk* and *MemPred* exhibit relatively similar throughput levels throughout the process. However, both *FuncMem* and *FuncSched* consistently demonstrate higher throughput rates. Our approach achieves higher throughput, with a maximum of 1.8 functions per second, while OpenWhisk reaches a maximum of 0.8 functions per second. Our scheduler also outperforms the current task execution.

Figure 5(d) shows the overall time in which the framework completes all tasks. It demonstrates that MemFunc, FuncSched, and *FuncMem* reduce the cumulative execution time by 17.59%, 47.53%, and 54.96% , respectively, compared to the baseline OpenWhisk. Furthermore, Figure 5(d) suggests that utilizing the scheduler is a more practical approach for increasing throughput and reducing the overall function completion time than solely focusing on reducing memory usage.

Figure 5(e) provides insights into the occurrence of cold starts as the number of invocations increases. Among the 200 invocations, *OpenWhisk* experiences 178 cold starts, while *FuncSched* reduces it to 145. Further improvements are observed with *MemPred* , resulting in only 101 cold starts. Finally, *FuncMem* shows the most significant reduction, achieving a mere 65 cold starts. These findings highlight the effectiveness of memory reduction in decreasing cold starts in the serverless architecture. Adopting our methodology minimizes the cold start, improving the overall serverless architecture's performance and latency.

5 DISCUSSION

One of the key reasons why cloud tenants are increasingly adopting serverless computing is the potential for cost reduction. Cloud providers face the challenge of optimizing their costs to accommodate growing demands. The proposed techniques can help reduce the memory usage of serverless functions, reducing the cost of executing the serverless function. The investigation of non-blocking requests has been relatively underexplored in existing literature and carries significant implications. Non-blocking requests differ from blocking tasks because they do not require immediate execution. Service providers have the flexibility to allocate resources gradually, reducing memory usage and scheduling to mitigate issues related to throughput and cold starts. To our knowledge, this is the first paper that explores the separation of blocking and non-blocking requests in serverless architecture.

This paper addresses the efficient minimization of waiting queues for serverless applications. Furthermore, our tool *FuncMem* offers valuable insights, including task queue sizes and task frequency. This information can be effectively combined with hybrid histogram techniques to estimate keep-alive and prewarm windows. Our energy-saving techniques from a previous paper on distribution systems [15] can be combined with current approaches to delineate task execution between edge and server in a serverless architecture. We can expand the proposed queue in scenarios involving multiple node clusters, leveraging the invoker's role as a replica within OpenWhisk.

6 RELATED WORK

Our work comprises memory estimation, function scheduling, cold start, profiling, and non-blocking requests in a serverless platform.

Cold start: Predicting pre-warmed containers and extending container lifespans are the two most common approaches to reducing cold starts. Researchers have adopted various strategies in this regard. For instance, one approach involves analyzing historical invocation patterns and implementing periodic [26] or seasonal invocations [26] to pre-warm containers. Shahrad [22] proposed the hybrid histogram policy, which utilizes invocation frequency to estimate pre-warm windows and keep-alive windows. Daw [8] addresses the cold start issue in function chains by predicting and executing subsequent functions in advance. Oakes [18] caches commonly used libraries and interpreters in containers. The practical implementation of OpenWhisk primarily influences the design assumptions in this paper. In *FuncMem* , we identify the minimum memory usage of the deployed function running through the proposed simulator. We leverage simulated information to effectively schedule functions within the invoker node to reduce the cold start.

Scheduler: Suresh [25] introduces ENSURE, a function-level scheduler, to prevent the cold start and minimize the service provider's costs by managing resources on the server node. Their scheduler dynamically scales the number of containers and hosts in response to an increasing workload without violating the service level agreement (SLA). Fifer [11] performs offline profiling to estimate the execution time of the function, and, based on the history of invocation, it spawns containers to reduce SLA violations. FnSched [24] minimizes CPU contention of serverless functions, considering resource consumption and invocation history patterns to dynamically regular CPU share. COSE [6] uses the Bayesian optimization method to estimate the configuration of serverless functions. Our recent study introduced a function distribution method across nodes, optimizing resource utilization and reducing waste by analyzing historical invocations, labeling, and memory usage patterns [20].

In this paper, we present a novel approach that involves the implementation of adaptive and error queues within invoker nodes to optimize the handling of incoming requests. These requests are periodically assigned priorities, taking into account their types, such as blocking and non-blocking requests, the presence of warm functions, function deadlines, and error execution requirements. Subsequently, our scheduler makes decisions regarding the execution and deletion of requests based on these established priorities.

Memory Estimation: OFC [17] utilizes machine learning to estimate memory allocation using the J48 decision tree algorithm. AWS Lambda Power Tuning is an open-source tool for optimizing Lambda functions' power and memory configuration [1]. Profiling services is the common approach to estimate estimation of services [19]. *FuncMem* provides detailed metrics for the actual memory usage and the essential minimum memory requirement, significantly enhancing the precision of memory estimations. Additionally, considering the frequent interactions of serverless functions with various external services, *FuncMem* excels at differentiating these functions based on their dependencies on these services. This capability enables *FuncMem* to profile and predict memory usage, ensuring that the performance of serverless functions remains optimal and uncompromised.

7 FUTURE WORK AND CONCLUSION

For future research, we will pursue the following research directions. First, we aim to pre-warm the containers to reduce the cold start issue. Because *FuncMem* only identifies the memory usage of a function, we want to utilize other metrics such as execution time and function category for scheduling. We will explore ways to reduce the memory usage of serverless functions and employ deep learning methods to predict memory usage. Another direction we focus on is expanding our scheduler to select appropriate nodes. Caching is another way to improve serverless architecture; we will explore run-time and node caching. Finally, we will investigate ways to enhance the security of our *FuncMem* .

In this paper, we presented *FuncMem* , a methodology that reduces the memory of the deployed function and prioritizes the execution and deletion of the function to reduce cold start latency in the serverless architecture. *FuncMem* profiles and estimates the memory usage of the deployed function and reduces the memory allocated without affecting function execution. At invocation, *FuncMem* provides an adaptive queue and executes and deletes a function prioritizing blocking requests, followed by warm containers, function deadlines, and the availability of warm functions. For the evaluation, we compared memory utilization, system throughput, average completion time, average function wait time, average function initialization time, and cold start with a baseline, *OpenWhisk* . We show that *FuncMem* significantly improves performance, reducing cold start latency by 63.48%, decreasing memory allocation by 46.98%, reducing cumulative execution time by 54.93%, and improving average waiting time in a serverless architecture.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Education (No. 2021R1I1A3043889) and the Ministry of Science and ICT (No.2021R1 A5A1021944) and also partially supported by the Digital Innovation Hub project supervised by the Daegu Digital Innovation Promotion Agency (DIP) grant funded by the Korea government (MSIT and Daegu Metropolitan City) in 2023 (No.DBSD1-03).

REFERENCES

[1] 2023. Aws Lambda Power Tuning. https://github.com/alexcasalboni/aws-lambda-power-tuning Last accessed 20 September 2023.
[2] 2023. The Algorithms. https://github.com/TheAlgorithms. https://github.com/TheAlgorithms Last accessed 1 Jan 2024.
[3] 2023. The CIO's Guide to Serverless Computing. https://www.gartner.com/smarterwithgartner/the-cios-guide-to-serverless-computing Last accessed 9 September 2023.
[4] 2023. What AWS Lambda's Performance Stats Reveal. https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/ Last accessed 20 September 2023.
[5] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.
[6] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. Cose: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 129–138.
[7] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. *Research advances in cloud computing* (2017), 1–20.
[8] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*. 356–370.
[9] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2020. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110* (2020).
[10] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. 2019. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 261–2615.
[11] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T Kandemir, and Chita R Das. 2020. Fifer: Tackling underutilization in the serverless era. *arXiv preprint arXiv:2008.12819* (2020).
[12] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–26.
[13] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).
[14] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 502–504.
[15] Young-Woo Kwon and Eli Tilevich. 2012. Energy-efficient and fault-tolerant distributed mobile execution. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*. IEEE, 586–595.
[16] Wei Ling, Lin Ma, Chen Tian, and Ziang Hu. 2019. Pigeon: A dynamic and efficient serverless and FaaS framework for private cloud. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 1416–1421.
[17] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. 2021. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 228–244.
[18] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 395–400.
[19] Manish Pandey, Breno Dantas Cruz, Minh Le, Young-Woo Kwon, and Eli Tilevich. 2021. Here, there, anywhere: Profiling-driven services to tame the heterogeneity of edge applications. In *2021 IEEE International Conference on Smart Data Services (SMDS)*. IEEE, 61–71.
[20] Manish Pandey and Young Woo Kwon. 2023. Optimizing Memory Allocation in a Serverless Architecture through Function Scheduling. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*. IEEE, 275–277.
[21] Bin Qian, Jie Su, Zhenyu Wen, Devki Nandan Jha, Yinhao Li, Yu Guan, Deepak Puthal, Philip James, Renyu Yang, Albert Y Zomaya, et al. 2020. Orchestrating the development lifecycle of machine learning-based IoT applications: A taxonomy and survey. *ACM Computing Surveys (CSUR)* 53, 4 (2020), 1–47.
[22] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 1063–1075.
[23] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *arXiv preprint arXiv:2003.03423* (2020).
[24] Amoghvarsha Suresh and Anshul Gandhi. 2019. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th international workshop on serverless computing*. 19–24.
[25] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. 2020. ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 1–10. https://doi.org/10.1109/ACSOS49614.2020.00020
[26] Vladislav Tankov, Yaroslav Golubev, and Timofey Bryksin. 2019. Kotless: A serverless framework for kotlin. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1110–1113.