Serverless Cold Start Solution with Swap

Yih CHENG – Independent Research

Supervised under Professor Wei WANG, HKUST

Abstract

Serverless computing, or Function-as-a-Service (FAAS), provides users the ability to deploy applications and execution functions without paying additional effort in resource allocation and management. Cold start occurs when no warm instances are available when a function request arrives, thus a new execution instance must be spawned. Solutions to pre-warm instances are limited by memory capacity and therefore cannot scale up well. Swapping is a technique utilized in operating systems to free up additional physical memory by moving rarely used data to external storages. In this project, we apply swapping of pre-warm instances between memory and solid state drives to scale up the storing capacity and avoid cold start. Knix, a serverless platform designed by Bell Lab, is used as the framework for our project, as we revise and test the idea of swapping on it [1].

Keywords: Serverless, Cold start, Swap

1. Introduction

Serverless services such as AWS Lambda and Microsoft Azure Functions free cloud users from managing and dealing with resource clusters. A piece of code can be easily deployed as a stateless, containerized function. In addition, serverless also allows users to pay for only the execution time instead of paying a price for resource reservation. This is due to its ability to scale down to zero while scaling up to meet rapid requests.

Downsides of serverless computing include its inability to communicate between functions due to its stateless characteristics, and the slow cold start that penalizes performance when lacking warm execution instances.

1.1 Dissecting Cold Start

In serverless, cold start occurs when a function request arrives but no ready instance is available to serve it, resulting in the need of creating a new execution instance. Creating a new instance involves two major initializations, being platform initialization and function initialization. Platform initialization allocates required resources to create the environment for the instance. The environment may be a virtual machine, a container, or just a process. Function initialization depends on the programming language and the user code.

1.2 Existing Solutions

Most approaches taken to mitigate cold start issue uses pre-warm techniques, where periodic dummy requests are sent to the platform to keep a function instance alive [2]. This method works effectively, as it eliminates cold start after the first instance initialization. However, it is not ideal when workloads are unpredictable, as no guarantee can be made that simultaneous requests will not exceed the provisioned warm instances.

Another proposed solution is to keep snapshots of common environment partitions to reduce the workload of platform initialization [3]. This method can minimize repeating procedures during instance initialization, thus reducing cold start effect. However, it also leads to poor resource utilization, as a large pool of unused skeleton containers awaits in memory.

2. The “Swap” Approach

As identified in the previous sections, the trade off of keeping warm instances alive to avoid cold start is memory utilization. Swap, a virtual memory management method, allows us to avoid this problem by swapping unused memory content from physical memory space to secondary storage. Therefore, the main idea of this project is to implement swapping in [4], [5] so that keeping large pools of active instances is possible. The idea is to pre-warm a bunch of instances in memory, swap them out to second storage, and keep them alive there. In this case, memory is freed up as only a small number of instances live here while the majority awaits in the second storage. When a request arrives and no available instances in memory exist, we then swap in a living instance from the second storage to serve it. The external storage can be fast solid-state drives or other NVME storages, as they provide state-of-the-art data transmission speed, which can reduce swap-in latency to the greatest extent. As these external storages are much cheaper and cost efficient compared to memory, scaling up the amount of available instances will no longer be impossible.

3. Implementation

We implement this project using Knix [1] as the backbone. Knix is a serverless platform originally designed by Nokia Bell Lab. It is also the base framework of [4].

3.1 Knix Original Design

Here, we briefly describe and analyze the original design of Knix to compare with our original design.

1) **Component:**
    a) *FunctionWorker:* In Knix, each FunctionWorker object runs as an individual process to execute a function request. Each state in the deployed workflow calls a corresponding FunctionWorker. Once created, it continuously listens to *LocalQueue* for topics designated for it. If a specific topic is captured, i.e. this FunctionWorker receives a request to run a specific function task, it forks a child process to handle it while the parent process continues to listen for new requests. The result of the executed task along with the next topic will be placed in the *LocalQueue* by the child process for the next state’s FunctionWorker to capture. If it is the last state, the result of the whole workflow will be

put in the *DataLayer*, which is just a simple shared key value structured storage. After execution, the child process is terminated. As a result, this chained execution of FuntionWorkers allow a series of workflow to be executed correctly.

b) *LocalQueue:* LocalQueue is a unified platform for processes to communicate in Knix, as shown in *FunctionWorker*. A process (which is in fact a FunctionWorker instance) first creates a message containing a specific topic and intended content (can be in key-value pair structure), and places it in the *LocalQueue*. It expects another process (which is also another FunctionWorker instance) to capture this message on *LocalQueue* and decode the content correctly.

c) *Frontend:* This is the entry and exit point of Knix. A workflow execution request arrives at *Frontend* first, and *Frontend* places the first topic in the workflow on *LocalQueue* to trigger the start of the workflow execution. It then waits for the final result to be placed on the *DataLayer*.

2) Request flow:

When a workflow execution is requested, the request arrives at *Frontend* of Knix. *Frontend* places the first state's topic along with input parameters on *LocalQueue* to kickstart the whole execution. The respective *FunctionWorker* listens and captures the topic, and it forks a child process to execute the state's request if there is any. After the child process finishes execution, it places the result labeled with the next state's topic back to *LocalQueue.* The child process is then terminated, while the next state's *FunctionWorker* captures the result on *LocalQueue* to continue the execution. After the last state is executed, the final result is placed on *DataLayer*, where *Frontend* is waiting to pull the result. A diagram of this workflow is shown in Fig. 1 for better illustration.

3) Analysis:

In Knix, a container is created for each workflow, and each function is executed in a *FunctionWorker* process. The original design utilizes Linux fork to create new child instances whenever a *FunctionWorker* receives a new request. Linux fork system call lazy copies the parent instance, and therefore skips both environment and function initialization while creating the new function instance. It also takes advantage of the copy-on-write characteristic of Linux fork, where read-only data is not copied but shared between forked processes. In addition, different workflows are isolated by containers, while function instances are isolated by different processes.

3.2 Revised Design

This revised design is based on [4]'s idea with some revisions. The main goal is to implement "swapping" of pre-created *FunctionWorkers* instead of creating a new child process whenever a request is received by a parent.

1) Component (only those that differ from the original design):

a) *FunctionWorker:* In the revised design, a fixed number of *FunctionWorkers* are pre-allocated during the deployment process of the workflow. They are created utilizing Linux fork, and each process reports itself's `pid` to the *ExecutionManager* through *LocalQueue* topic `{new_fw_reporting}`.

These instances will then be swapped out and stay in external storage. Each *FunctionWorker* pulls on the topic of `{function_topic}_{pid}` where `function_topic` is the function state itself and `pid` is its own process id. After an instance executes a topic, it will request for `pid` of a function instance corresponding to the next function topic by consulting the *ExecutionManager* through topic `{request_next_fw}`. The instance will then wait for topic `{function_topic}_{pid}`, which is pushed to *LocalQueue* by *ExecutionManager,* that contains the `pid` of the next *FunctionWorker*. After receiving the `pid`, this instance will push the result to *LocalQueue* using topic `{function_topic}_{pid}` (here the `function_topic` is the next state's function topic) so that the next function instance can pull down the result and continue executing the workflow. Unlike the original design, all *FunctionWorker* instances preserve even after executing the state task. They are swapped out if idle by *ExecutionManager*.

b) *ExecutionManager: ExecutionManager* acts as the manager of *FunctionWorker* pool. It stores a dictionary of each *FunctionWorker* instance and their corresponding children's pid. *ExecutionManager* pulls on topic `{new_fw_reporting}` to record newly forked instances and swap them out. It also pulls on topic `{request_next_fw}`, implying that a *FunctionWorker* has finished execution and requests an instance for the next state in the workflow. *ExecutionManager* will assign a corresponding *FuncionWorker* process, swap it in to memory, and push its pid to *LocalQueue* with topic `{function_topic}_{pid}`. After execution of the workflow, *ExecutionManager* is responsible for terminating all *FunctionWorkers*.

2) Request flow:

During the deployment stage of the workflow, every function instance that appears in the workflow will be forked and swapped out. When *Frontend* places the first function topic and input parameters on *LocalQueue*, a *FunctionWorker* of the first state will be swapped in. It will capture the entry topic on *LocalQueue* and execute the task. After execution, it consults *ExecutionManager* for a next instance. *ExecutionManager* assigns a corresponding process for the instance that finished execution, and swaps in that process. After receiving the `pid` of the next *FunctionWorker* instance, the current instance directly sends the result to that instance through *LocalQueue* to continue the workflow execution. After the last state is executed, the final result is placed on *DataLayer*, where *Frontend* awaits. A diagram of this workflow is shown in Fig. 2 for better illustration.

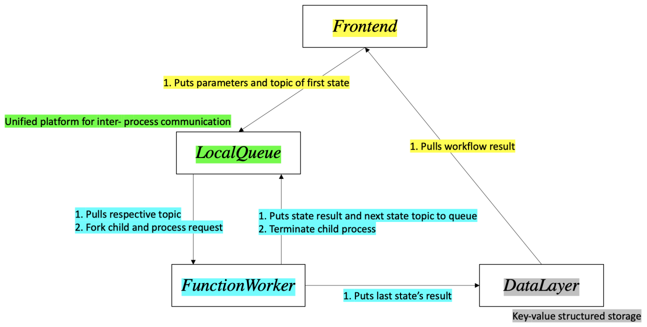
Frontend
1. Puts parameters and topic of first state
Unified platform for inter- process communication
LocalQueue
1. Pulls workflow result
1. Pulls respective topic
2. Fork child and process request
1. Puts state result and next state topic to queue
2. Terminate child process
FunctionWorker
1. Puts last state's result
DataLayer
Key-value structured storage

Fig. 1 Original request flow of Knix

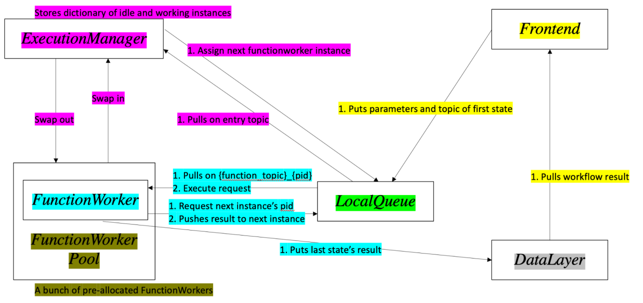
Stores dictionary of idle and working instances
ExecutionManager
1. Assign next functionworker instance
Frontend
Swap in
Swap out
1. Pulls on entry topic
1. Puts parameters and topic of first state
1. Pulls on {function_topic}_{pid}
2. Execute request
FunctionWorker
LocalQueue
1. Pulls workflow result
1. Request next instance's pid
2. Pushes result to next instance
FunctionWorker Pool
1. Puts last state's result
DataLayer
A bunch of pre-allocated FunctionWorkers

Fig. 2 Revised request flow of Knix

3) **Analysis:**

The revised version introduces the idea of swapping. The main argument is that the time needed to swap in instances is significantly lower than forking, which is in fact coldstarting a function instance whenever needed.

4. Result and Future Improvement

So far no confident results can be made through our testing. We have successfully tested simple sequential functions where each function worker only requires little workload, and the results did not differ much from the original Knix implementation. For example, a simple workflow that randomly creates 1000 float numbers then averages them takes 2ms to execute first state and 1ms to execute second state. In between the states, the original design takes 10ms which is equal to the coldstart time of the second state's function instance, while it takes roughly 9ms for the revised design, which is the latency of swapping the second function instance in to memory. While a small improvement is shown, not a lot of difference can be noticed compared to the huge differences in the preliminary results of [4].

We believe the main reason for the similar results is because of the longer request flow shown in Fig. 2 compared to the original design of Knix as shown in Fig. 1. Also, the overly simple workflow does not showcase the power of swapping. However, during our testing of more complex workflows such as convolution operations makes the current revised Knix platform unstable. We believe it is because of the pre-allocation of numerous function instances that are actually used for only a few times during the workflow execution, and some even stay idle for the whole execution. For future improvements, the above problem should be dealt with, and different instance scheduling methods along with resource allocation algorithms can be implemented.

5. Reference:

[1] Knix Github: https://github.com/knix-microfunctions/knix
[2] "Keeping Functions Warm - How to Fix AWS Lambda Cold Start Issues,": https://www.serverless.com/blog/keep-your-lambdas-warm/
[3] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile Cold Starts for Scalable Serverless," *USENIX, 2019*
[4] Hok Chun NG, "Report: Boosting Serverless Infrastructure with Swap and Fast Storage": SheepiesLab/knix_doc (github.com)
[5] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, B. Grot "Benchmarking, analysis, and optimization of serverless function snapshots," *ASPLOS, 2021*