Lukewarm Serverless Functions: Characterization and Optimization

CS422 Project Final Report

Group 1 Members:
Sarthak(190772), Manish(190477), Nibir(190545)

November 14, 2022

Contents

1 Introduction **3**

2 Preliminary Observations **3**

3 Solution Proposal **3**

4 System Details **4**

4.1 Essential Components 4

4.2 Workflow 5

5 Implementation Details **6**

6 Results **6**

6.1 MPKI Graph: 7

6.2 IPC Graph: 8

7 References **9**

1 Introduction

- Serverless computing has emerged as a widely-used paradigm for running services in the cloud. In serverless, developers organize their applications as a set of functions, which are invoked on-demand in response to events, such as an HTTP request. To avoid long start-up delays in launching a new function instance, cloud providers tend to keep recently-triggered instances idle (or warm) for some time after the most recent invocation in anticipation of future invocations.

- To avoid the long delays of booting a new function instance, cloud providers tend to keep recently-invoked instances alive (or warm, in serverless parlance) instead of shutting them down, in anticipation of additional invocations to that instance.

2 Preliminary Observations

- High degree of interleaving in the execution of warm serverless functions with short running times leads to obliteration of their on-chip microarchitectural working sets between invocations, resulting in 31-114% performance degradation relative to execution with a warm micro-architectural state.

- A detailed Top-Down analysis of the causes of the performance loss in the interleaved setup showed that the largest fraction (56%) of extra execution cycles is attributed to fetch latency indicating a bottleneck in instruction delivery.

3 Solution Proposal

- Jukebox, a record and replay instruction prefetcher specifically designed to accelerate lukewarm executions of serverless functions. Jukebox exploits instruction commonality(the high commonality of instruction blocks across invocation) by recording the working set of one invocation and replaying it whenever a new invocation to the same instance arrives.

- Compared to state-of-the-art instruction prefetchers which target the L1-I, a unique feature of Jukebox is that it prefetches into the L2. This choice is motivated by two observations. First, the instruction footprints of individual invocations of the containerized functions generally stay within 800KB, a value much higher than a typical L1-I capacity of 32–64KB. However, such instruction footprints fit comfortably within the L2 capacities of today's server processors.

- Secondly, prefetching into the large L2 significantly simplifies the prefetcher's design, since it avoids the need to place prefetches into the small L1-I. With a small cache as a prefetch target, it is essential that prefetches arrive just in time to avoid being evicted (if they arrive too early) or not being useful (if late). With L2 as the prefetch target, an aggressive prefetcher can simply fill it at the start of execution and expect instructions to not be evicted in the duration of a short-running function.

4 System Details

4.1 Essential Components

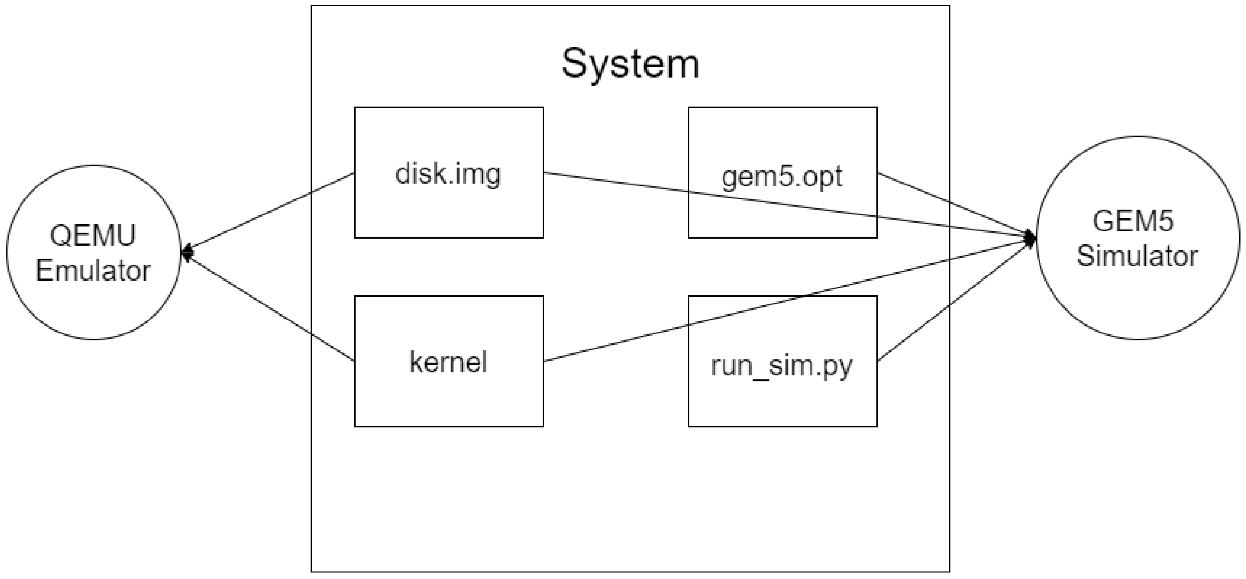


Figure 1: Essential Components

disk.img

The virtual disk image contains all files for the guest OS on which the experiments are run.

kernel

The operating system on which the experiments are performed (Linux kernel v5.4.84).

gem5.opt

Gem5 binary simulates the hardware and launches the system using the disk.img, kernel and run_sim.py.

run_sim.py

Python script file used by gem5.opt as a config that describes the initial script to be run and allows us to configure our experimental workflow.
The QEMU simulator uses the kernel and the disk.img to emulate the system and also provides shell access to use a system as is.

4.2 Workflow

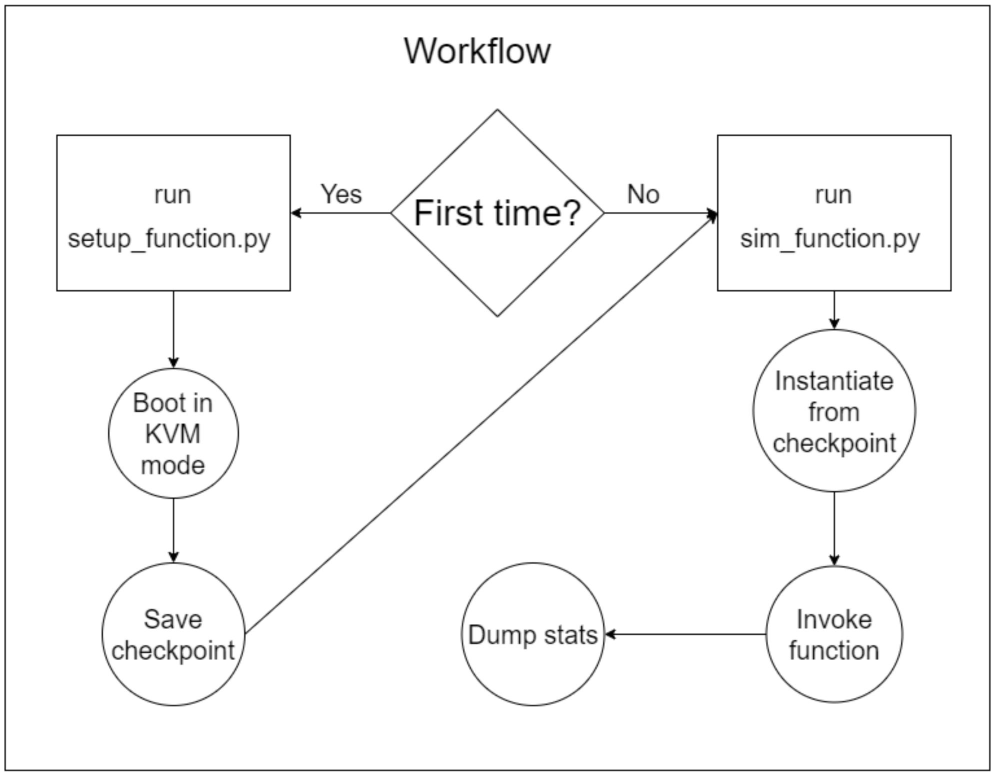


Figure 2: System Workflow

To reduce the duration of the experiment which increases unnecessarily due to the long booting process, the KVM core is used to perform a quick boot and save a checkpoint of the system. For the subsequent experiments where we need to measure important performance statistics, we instantiate from that checkpoint and continue our experiments.

5 Implementation Details

- Added two hardware MSR(Model Specific Register) registers. One register is like a flag register which is set when we want the recording phase of Jukebox to start. The other register is like an identifier register to identify the process for which the record and replay logic has to be done. This register is only set for the process running the function under evaluation.

- The record logic sits at the **cache.cc** file in the **src/mem/cache** folder in the function **handleTimingReqMiss()**. At first, we extract the thread context of the currently running thread. Then we check the **misc_reg::ISR** register if it is time to record. On approval, we push the miss addresses to the Jukebox buffer. We have multiple condition checks as we only want to record instructions misses in the L2 level cache. The **ctr** variable is the current pointer up to which addresses have been pushed/popped. It also acts as an identifier for the function instance process for which evaluation is being run.

- The replay logic sits at the **tagged.cc** file in the **src/mem/cache/prefetch** folder in the function **calculatePrefetch()**. It gives a considerable part of its *degree* to the Jukebox Prefetcher and only a small part to the default NextLine Prefetcher.

Please check out the code here.

6 Results

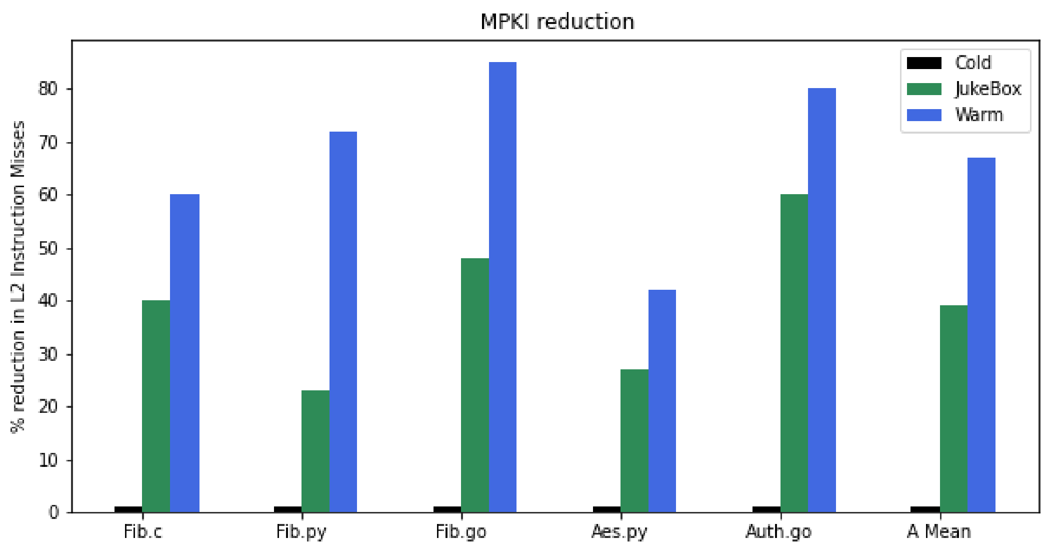
- **Warm Execution**: This is the repeated execution of the same function which will result in cache hits. To simulate warm execution we execute a function 10 times and then record the stats in the eleventh execution which we say is the warm execution with respect to the micro-architectural state of the CPU.

- **Cold Execution**: This is the first time execution of the function where there are no cache hits. To simulate this scenario we record the stats of the first execution of the function.
- **JukeBox Execution**: This refers to the execution of the function in the replay phase of the jukebox prefetcher. To simulate this execution, we first record the misses in the first execution of the function then we run another function to thrash the cache then we again call the function in the replay phase in which all the recorded state is prefetched.

These are the function we run to test our implementation of the Jukebox:
1. Fib.py
2. Fib.c
3. Fib.go
4. Aes.py
5. Auth.go

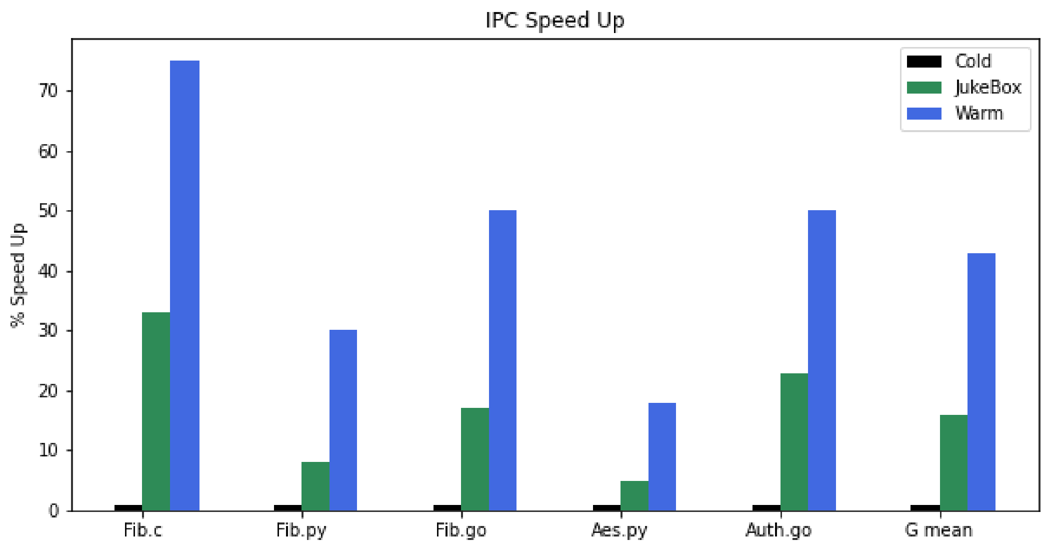
These functions are implemented and compiled by us. These are not exactly the same as the paper. We implemented these functions in a way that our system supports carrying out the experiments. We chooses 3 different implementation of fibonacci function to show that speed up is language independent.

6.1 MPKI Graph:



Cold execution is taken as a baseline. We achieved a 39% reduction in L2 instruction misses in JukeBox execution.

6.2 IPC Graph:



Cold execution is taken as a baseline. We are able to achieve a 16% speed-up as compared to the baseline.

7 References

- Lukewarm serverless functions: characterization and optimization
  Schall et al - ISCA '22: Proceedings of the 49th Annual International Symposium on Computer Architecture June 2022 Pages 757–770
  https://doi.org/10.1145/3470496.3527390

- Gem5 users mailing list https://www.mail-archive.com/gem5-users@gem5.org

- https://cirosantilli.com/linux-kernel-module-cheat-split

- MSR library for go: https://github.com/fearful-symmetry/gomsr