CodeCrunch: Improving Serverless Performance via Function Compression and Cost-Aware Warmup Location Optimization

Rohan Basu Roy
Northeastern University

Tirthak Patel
Rice University

Rohan Garg
Nutanix Inc.

Devesh Tiwari
Northeastern University

Abstract

Serverless computing has a critical problem of function cold starts. To minimize cold starts, state-of-the-art techniques predict function invocation times to warm them up. Warmed-up functions occupy space in memory and incur a keep-alive cost, which can become exceedingly prohibitive under bursty load. To address this issue, we design CodeCrunch, which introduces the concept of serverless function compression and exploits server heterogeneity to make serverless computing more efficient, especially under high memory pressure.

CCS Concepts: • **Computer systems organization** → Cloud computing.

Keywords: Serverless Computing, Function Compression

ACM Reference Format:
Rohan Basu Roy, Tirthak Patel, Rohan Garg, and Devesh Tiwari. 2024. CodeCrunch: Improving Serverless Performance via Function Compression and Cost-Aware Warmup Location Optimization. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3617232.3624866

1 Introduction

The serverless computing model improves programmer productivity by enabling them to leverage cloud computing resources without explicit management and provisioning of hardware resources. This has accelerated the adoption of the serverless model in a wide variety of areas. Serverless is specially designed for short-running functions, with the execution time in the order of seconds to minutes. Each time a function is invoked, a serverless instance (container or microVM) may be created on a server. The function can only be executed once the instance downloads and installs the function source code, runtime, and all the necessary dependencies. This initialization time or the *cold start time* can constitute 40%-75% of the execution time of the function, thus, increasing its overall latency [61, 67, 79]. To avoid cold starts, service providers keep functions alive in the memory for some time after their execution, in anticipation of a subsequent invocation [24, 61, 67]. This enables a function to undergo a *warm start*, avoiding the added overhead of a cold start. However, the provider incurs a *keep-alive cost* for keeping the function warm.

Current Challenges and Gaps in Existing Solutions. The best solution to avoid cold starts is to keep all functions alive in memory at all times. However, this is prohibitively expensive in terms of the keep-alive cost. Existing solutions [24, 61, 67, 79] predict function invocations and accordingly keep functions alive to minimize cold starts and keep-alive costs. However, there are several challenges.

Previous research on function keep-alive strategies are shown to be effective at achieving warm starts on average [24, 61, 67]. However, our experimental characterization (Sec. 2) reveals two main observations. First, during periods of high function invocation load, the fraction of warm starts can drop significantly due to high memory pressure – leading to a lower return on investment (i.e., a low fraction of warm starts of functions during peak load). Second, there are interesting performance and keep-alive cost trade-offs for serverless functions on recent commercial offerings of ARM and x86 platforms — existing strategies do not exploit these trade-offs for improving the performance of serverless functions.

To reduce memory pressure and increase the likelihood of warm starts, we demonstrate that in-memory compression of a serverless function's filesystem can be promising, and it can achieve more warm starts, especially under high a load. However, compression is a double-edged sword due to its potential impact on execution time as decompression latency falls on the critical path of function execution, even though the compressed function receives a warm start. Second, we

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0372-0/24/04…$15.00
https://doi.org/10.1145/3617232.3624866

leverage the recent heterogeneous serverless offerings (e.g., x86 vs. ARM processors on Amazon Lambda platform) [82]. We found that some serverless functions achieve lower execution times on ARM processors, while others on x86 processors, but the keep-alive cost is lower on ARM processors – presenting a performance and keep-alive cost trade-off.

CodeCrunch is the first work to demonstrate that serverless service time can be improved by (a) optimally deciding how long to keep functions alive after their invocation, (b) deciding when and which functions to compress during the keep-alive period, and (c) exploiting x86 and ARM processors for performance and keep-alive cost optimization. Unfortunately, this leads to a complex optimization problem. As we show in Sec. 2, traditional techniques are often sub-optimal at reducing service time due to the large and complex search space of optimization. CodeCrunch proposes an effective and online strategy to solve this problem by leveraging the sequential random embedding (SRE) optimization [57].

Contributions of CodeCrunch

CodeCrunch Key Concepts. To the best of our knowledge, this is the first work to identify the opportunity scope of the following two ideas: *(a) function compression in serverless platforms, and (b) processor heterogeneity (x86 vs ARM) for warming up serverless functions.* CodeCrunch discusses the challenges in exploiting these ideas effectively for improving the performance of serverless workloads. CodeCrunch designs and implements a novel framework that uses *sequential random embedding (SRE) optimization* [57] to determine the near-optimal keep-alive time and processor type for a large number of functions on the fly. CodeCrunch demonstrates that *intelligent function compression (compression of the filesystem) increases the number of warm starts for serverless functions*, especially under peak load – where due to high memory pressure, the current schemes return low reward (fewer warm starts) on investment (keep-alive budget).

CodeCrunch Experimental Results. *CodeCrunch improves the average service time by 32% compared to the state-of-the-art baseline solution (SitW [67]), using the same keep-alive budget as SitW.* CodeCrunch is within 6% of the Oracle solution that uses Oracle knowledge to warmup functions right before being invoked. *CodeCrunch also enhances other recent techniques (IceBreaker [61] and FaasCache [24])* that primarily focus on predicting function invocation time and pre-warming based on their historic invocation patterns, instead of the keep-alive approach. CodeCrunch leverages compression, processor heterogeneity, and sequential random embedding to obtain over 15% and 25% improvement in the total service time over IceBreaker [61] and FaasCache [24], respectively. In fact, this work shows that CodeCrunch's key ideas can enhance the state-of-the-art baseline solution (SitW [67]) to the extent that enhanced SitW performs similarly or slightly better than recent techniques (IceBreaker and FaasCache) – this is particularly significant because it allows us to enjoy the better performance while maintaining the simplicity, production-level practicality, and effectiveness of SitW, without the complex overhead of more recent techniques such as IceBreaker that rely on FFT-based prediction algorithms. CodeCrunch's implementation is open-sourced at: https://zenodo.org/doi/10.5281/zenodo.8347244.

2 CodeCrunch: Background and Motivation

In this section, we first define the basic concepts of serverless computing. Then, we motivate the design of CodeCrunch.

Basic Definitions. Serverless computing is attractive for users as it offers a *pay-as-you-go* billing model. Unlike VMs, users do not have to estimate resource requirements. Cloud providers scale computing resources depending on an application's demand, and the user is charged for only the resources used. When a serverless function is invoked, it incurs an initialization overhead, called the *cold-start* time. To avoid this, cloud providers keep functions warm in memory for some time after their last execution; if the functions are re-invoked, they undergo a *warm start*. The end-to-end time between the invocation of a function and the completion of its execution is called the *service time*, which includes the *execution time* and the cold-start time (zero for warm starts).

Service providers may not directly charge users for warm starts, however, they incur a keep-alive cost to avoid cold start overhead. This provides lower service times to users, opening the potential for higher revenue due to smaller QoS latency targets. A warmed-up function consumes memory, which could have been allocated by the cloud provider for some other purpose. The *keep-alive cost* is the product of the cost per unit time per unit memory for reserving the server, the memory required by the function, and the time for which the function is kept alive. For the operation of a serverless system, the provider cannot let the keep-alive cost increase without bounds as that will consume a lot of resources just for warming up functions. Minimizing service time while operating with a keep-alive budget constraint is a major challenge.

Motivation. CodeCrunch is a solution to optimize the service time for serverless functions. The design of CodeCrunch is motivated by the three observations discussed below.

One of the ways of reducing service time is by reducing cold starts. Reducing the cold-start overhead implies improving the fraction of functions receiving warm starts. Unfortunately, achieving a high warm start fraction requires devoting a large amount of memory to keep functions alive. This can incur high memory pressure and result in high keep-alive memory and keep-alive cost, as the warm-up candidate functions change over time in response to load.

We conducted an experiment with an open-source production grade serverless trace from Microsoft Azure [67] (more experimental methodological details are discussed in Sec. 4).

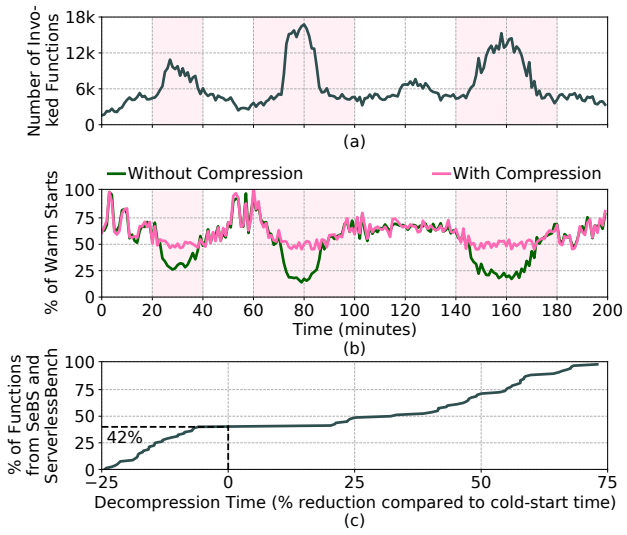

Figure 1. (a-b) In-memory compression of functions can yield more warm starts, especially during periods of high invocation load (shown by shaded regions). (c) The decompression time is lesser than the cold-start time for 42% of evaluated functions (*favorable case*), but some functions can have up to 75% higher decompression time compared to their cold-start time (*unfavorable case*).

In this experiment, we use *lz4* compression for all functions, set 10% system memory reserved for warming up functions, and use a static 10-minute keep-alive strategy used in production by Amazon Lambda and Microsoft Azure [24, 67], although the observed trends are valid for other settings. We chose these settings for interpretability and simplicity.

Through compression, each warmed-up function instance (container) consumes less memory than the original function, thus, allowing more functions to be warmed up. Fig. 1(a-b) reveals that in-memory compression of warmed-up functions is effective at reducing memory pressure, especially during high invocation load (highlighted). This, in turn, increases the fraction of warm starts during the periods when the invocation load is high. Overall, the mean percentage of warm starts increases from 51% (without compression) to 61% (with compression), just by compressing more functions during periods of lower warm-start rate. This is because in-memory compression allows more functions to be kept alive, even when it is challenging to predict the function re-invocation. This increases the chances of warm starts.

Unfortunately, compression is a double-edged sword due to its potential impact on execution time. A compressed function needs to be decompressed before execution. Keeping more functions alive via function compression only provides benefits in terms of function service time if the decompression time is smaller than the cold-start time without compression. Fig. 1(c) reveals that different functions show different characteristics – compressing is better than cold-start for 58% of all SeBS [16] and ServerlessBench [84] functions, but not

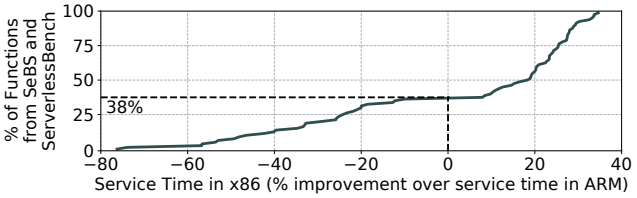

Figure 2. Performance of serverless functions varies with the processor hardware architecture (x86 or ARM).

for all. For functions where compression is effective (*i.e.,* decompression time is smaller than cold start time), we achieve 48% less cold starts on average. This can greatly improve the function service time. However, the challenge here lies in determining the functions for which compression is beneficial and when — it depends on the function and other functions that are invoked within the same time period (a complex optimization problem). CodeCrunch solves this challenge.

Finding I. *A large fraction of serverless functions benefit from being kept alive through in-memory compression as it enables more warm starts. However, compression is a double-edged sword due to its potential impact on service time. Determining which functions to compress and when to compress is challenging.*

Another promising strategy for reducing service time is by intelligent scheduling of functions in heterogeneous serverless function instances. Commercial serverless providers (e.g., AWS Lambda) have recently started providing options for executing serverless functions on different processor types (e.g., ARM and x86 [2]). Our experiments reveal that a single processor type is not more performant for all types of serverless functions. This is because the source code of serverless functions may have a natural performance affinity to either ARM or x86 processors [78]. For example, as Fig. 2 shows, nearly 38% of enterprise functions experience higher performance on ARM processors than on x86 processors.

Interestingly, the keep-alive cost on ARM processors is significantly lower than on x86 processors [1, 41]. Therefore, keeping alive functions in ARM nodes reduces the keep-alive cost compared to the corresponding x86 nodes. This allows more functions to be warmed up within a keep-alive budget. While operating within a given keep-alive budget, ARM execution can improve the service time by improving the percentage of warm starts due to lower keep-alive costs and better performance for some functions. However, determining which functions to warm up on which processor and for how long is a complex optimization problem, which CodeCrunch solves.

All the aforementioned results correspond to all the serverless function benchmarks from ServerlessBench [84] and SeBS [16] benchmark suites. On the x86 platform, compression is favorable (decompression time < cold start time) for

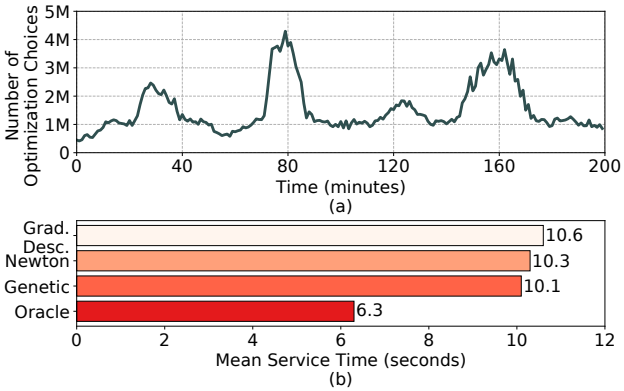

Figure 3. The optimization space is very large and traditional optimization schemes have sub-optimal performance.

42% of all evaluated functions. On ARM, this percentage is 46%, where almost all the compression-favorable functions on x86 are a part of compression-favorable functions on ARM. For 38% of all evaluated functions, ARM is a more performant processor choice, and 60% of these 38% functions are compressional-favorable functions on ARM.

Finding II. *Latest serverless offerings (e.g., Amazon Lambda) demonstrate the potential of using different types of architectures (ARM vs. x86 processors) for function scheduling. By design, the keep-alive cost is lower on a particular node type (ARM) for all functions. However, in terms of execution time, not all functions achieve better performance on x86 nodes. The combined trade-off of performance and keep-alive cost opens up the opportunity to optimize for service time and keep-alive cost — which has not been exploited before.*

We specifically note the distinction from a recent technique (IceBreaker [61]). IceBreaker [61] proposes executing serverless functions on heterogeneous high-end and low-end servers. In their setting, high-end servers are *always faster for all functions, by design.* Therefore, the optimization opportunity is limited, and the challenges are simpler. IceBreaker simply places a function on the "faster" or "slower" node depending upon the re-invocation probability. Unfortunately, the lack of function-performance-specific selection between x86 and ARM is the root cause of the limited effectiveness of IceBreaker compared to CodeCrunch (Sec. 5).

As established above, serverless service time can be optimized by (1) optimally deciding how long to keep functions alive after their invocation, (2) deciding when and which functions to compress in memory during the keep-alive period, and (3) intelligently exploiting servers with x86 and ARM processors for performance and keep-alive cost. Thus, for each function, all three tuning knobs need to be optimized. Again, since the optimization choice of one function affects the performance of other invoked functions (since the total compute and keep-alive resources are fixed), all invoked functions should be optimized together. This increases the size of the optimization space, which can be in the range of millions (Fig. 3(a)). The size changes with time based on the number of invoked functions per unit of time. This large size, along with the fact that the optimization space is discrete, is the reason why traditional optimization algorithms like gradient descent and Newton's methods perform sub-optimal in terms of reducing service time (Fig. 3(b)). Further, the genetic algorithm also performs poorly due to the large size of the optimization space. Here, Oracle is the most optimal, theoretically best solution that is practically infeasible as it is obtained via a brute-force search in the optimization space, constrained by a keep-alive cost budget. Thus, there is a need to develop techniques to reduce optimization space size to effectively reduce the service time of serverless functions.

Finding III. *Traditional multi-dimensional optimization techniques like gradient descent, Newton's method, and genetic algorithm are sub-optimal in reducing service time due to the large and complex search space of optimization. An effective solution to this problem requires an online strategy of optimization space reduction.*

3 CodeCrunch: Design and Implementation

CodeCrunch's goal is to optimize the service time of serverless functions with a keep-alive budget constraint. It does so by (1) intelligently selecting functions for compression to leverage the trade-off between decompression and warm starts, (2) scheduling functions on ARM or x86 processors based on the performance and keep-alive cost trade-offs, and (3) accurately predicting the keep-alive time of each function to improve the chances of warm starts. CodeCrunch uses Sequential Random Embedding (SRE) [62] to perform its optimization. CodeCrunch's optimization involves operation in a high dimensional space, due to a large number of invoked functions in a production serverless system. SRE reduces the complexity of the optimization problem by breaking down the entire problem into smaller, lower-dimensional sub-problems, making the optimization more feasible. It operates by randomly selecting functions for optimization in each round, ensuring a fair and dynamic approach. By employing SRE, CodeCrunch can effectively adapt to the changing characteristics of serverless functions and perform efficient function warm starts. Next, we describe the details of CodeCrunch's SRE-based operation to determine the compression choice, processor (ARM vs. x86), and keep-alive time.

3.1 CodeCrunch's Sequential Random Embedding

The first step to perform this optimization is to generate the search space of optimization, *i.e.,* all possible choices of compression, keep-alive time, and processor type for the invoked functions. CodeCrunch uses a choice space generator to perform this step, which we discuss next. Then, we discuss

that optimizing over such a large space is not possible and hence, a Sequential Random Embedding (SRE) optimization technique is used to mitigate this [57].

Optimization Space Generator. After every optimization interval, CodeCrunch generates all possible combinations of compression choice, keep-alive time, and processor type for all functions that were invoked within the interval. We choose an interval of one minute as it is a popular choice for commercial serverless services like Amazon Lambda and Google Cloud Functions [67, 79]. The choices are generated such that all of them result in a keep-alive cost lesser than or equal to the keep-alive budget for a given optimization interval, which is the sum of the average keep-alive budget for a given duration and the keep-alive cost saved up from the previous rounds of optimization (if the total keep-alive budget is not spent to warm up functions). CodeCrunch's design is practical as it lets the service provider determine and manage the average keep-alive budget, unlike previous state-of-the-art solutions. CodeCrunch's evaluation (Sec. 5) demonstrates the effectiveness of CodeCrunch for different keep-alive budgets.

Let the set S_t contain all available choices at time t. Each element of S_t is a tuple (C, T, K_t) corresponding to each function i, where C is the compression choice, T is the set of processor types, and K_t is the set of keep-alive times, invoked in the optimization interval t. Each (C, T, K_t) is generated such that all of them satisfy the following inequality to maintain the keep-alive cost while operating under the available keep-alive budget (K_t).

$$\sum_{i=1}^{N}(1-T_i)[C_i(M_iK_{t_i}X_{x86}) + (1-C_i)(M_{C_i}K_{t_i}X_{x86})]+$$
$$T_i[C_i(M_iK_{t_i}X_{ARM}) + (1-C_i)(M_{C_i}K_{t_i}X_{ARM})] \leq K_t$$

Here, N is the total number of functions invoked in the optimization interval at instance t, M_i is the uncompressed memory requirement of the i^{th} function, M_{C_i} is the compressed memory requirement of the i^{th} function, K_{t_i} is the keep-alive time for the i^{th} function, X_{ARM} is the keep-alive cost per unit time per unit memory to keep a function alive in a server with an ARM-based processor, and X_{x86} is the corresponding keep-alive cost of a server with an x86 processor. T_i denotes the choice of processor type – 0 for x86 and 1 for ARM, to keep the i^{th} function alive. C_i denotes the compression choice – 0 when it is compressed and 1 when it is not. The keep-alive time is chosen between 0 and 60 minutes, which is usually the range used in commercial serverless platforms [79]. In this expression, the top line corresponds to generating choices in S_t with x86 processor types, and the bottom line corresponds to generating choices with ARM-based processors.

Next, we discuss how CodeCrunch selects compression, processor type, and keep-alive time for all functions invoked within an interval out of all possible choices in the set S_t.

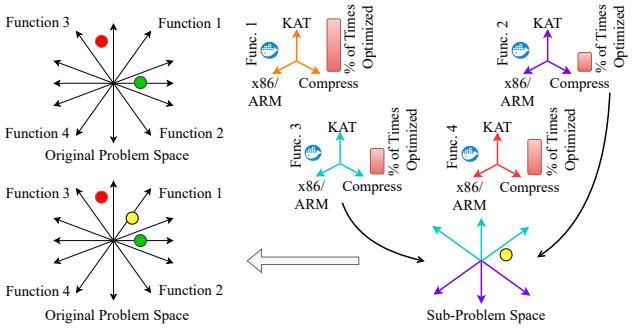

Figure 4. SRE creates lower dimensional sub-problems to find the optimal solution (the optimization progresses from the red to the yellow point, where the green point is the optimal solution).

Sequential Random Embedding for Optimizing Over the Search Space. If there are N functions invoked during interval t, each entry in S_t has $3N$ dimensions (compression choice, processor type choice, and keep-alive time for each of the N functions). Among all entries of S_t, CodeCrunch tries to determine the optimal entry, thus optimizing over $3N$ dimensions. For production serverless platforms, N can be in the range of thousands. Thus, the optimization dimensions can be excessively large. The keep-alive decision of functions needs to be done quickly, to properly warm up functions before their re-invocation. Hence, for production systems, optimization over $3N$ dimensions is impractical. To solve this problem, CodeCrunch uses sequential random embedding (SRE) [57, 62].

Why is SRE useful in CodeCrunch's context? Serverless functions do not always follow a predictable invocation pattern [61, 67]. Some of the functions can have invocation patterns that are to some extent periodic, but the presence of multiple periodic frequencies and frequently changing invocation patterns (including changing periods) can make their invocation period hard to predict. Frequently their inputs change, which results in changes in their execution time and invocation frequency. This is why a randomized algorithm like SRE is beneficial in a serverless setting as the generated solution does not depend too much on the individual characteristics of the functions and can capture the changes in execution time and invocation patterns by randomly re-selecting functions to create sub-problems at different intervals.

Integrating SRE into CodeCrunch's framework. SRE attempts to find a near-optimal solution by decomposing the original problem in multiple sub-problems – this decomposition reduces the high dimensionality of the original problem. The basic working principles of SRE are as follows.

If the original optimization problem has $3N$ dimensions, SRE reduces the problem into N_{SRE} sub-problems of D_{SRE} dimensions each. D_{SRE} includes the search space of $\frac{D_{SRE}}{3}$

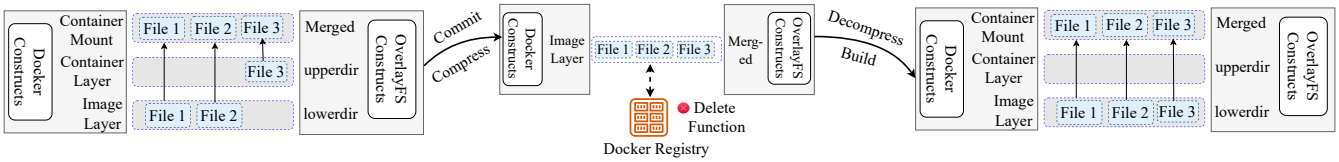

Figure 5. Compression of functions helps CodeCrunch to warm up more functions within a fixed keep-alive budget.

functions, with compression choice, processor type, and keep-alive time as three dimensions for each of the functions. Here, $D_{SRE} \times N_{SRE} < 3N$. D_{SRE} is the dimension of each of SRE's subproblems, and since each function has 3 dimensions (keep-alive time, compression choice, and processor choice), the space D_{SRE} is made up of $\frac{D_{SRE}}{3}$ functions. Each of these sub-problems is optimized in parallel, and the resulting solution is combined and mapped across all the dimensions of the original problem. This process is done for P_{num} rounds and the final result is taken as the mean of all the P_{num} optimization solutions. SRE has been shown to efficiently solve various optimization problems in large dimensions [17, 34, 57]. The values of D_{SRE}, N_{SRE}, and P_{num} scale in proportion to the number of invoked functions in t. However, the total number of dimensions searched in all the sub-problems across all the rounds, *i.e.,* $D_{SRE} \times N_{SRE} \times P_{num}$ is 10× smaller than the size of the original problem across $3N$ dimensions. SRE performs faster optimization because (1) it creates smaller sub-problems and does not optimize across all the dimensions of the original problem, and (2) it performs parallel optimizations across the sub-problems.

How does SRE perform sampling in an optimization round? Among the N functions invoked during interval t, SRE uses the percentage of times each of the functions was optimized previously (considering all of their invocations) to determine whether a function will be optimized in the current round. A lower value of this percentage increases the probability of a function's selection in the present optimization round. SRE uses probabilistic sampling to give all functions a fair opportunity to take part in the optimization rounds. This provides functions a probabilistic opportunity for re-optimization, upon being updated or invoked with different parameters. Fig. 4 shows the overview of CodeCrunch's SRE optimization in which CodeCrunch chooses to optimize functions 2 and 3 in the sub-problem space. Optimized solutions from previous rounds are selected for functions that are not chosen for optimization in the present round when reverting from the sub-problem space to the original problem space.

How is optimization performed on each of the sub-problems generated through SRE? Each sub-problem generated by SRE is optimized by estimating the service time of the functions in the sub-problem for different compression choices, keep-alive time, and processor type choices. Gradient descent is performed in the search space of a sub-problem and the optimization proceeds one step per round in the direction which minimizes the estimated mean service time of the functions in the sub-problem. The starting point of the gradient descent is the optimized solutions of the selected functions from previous rounds. If multiple solutions minimize the mean service time within a similar range (10% deviation), the solution that reduces the overall keep-alive memory consumption is chosen so that the excess keep-alive cost in the current interval can be credited to future intervals to generate S_t by optimization space generator.

Most serverless functions are invoked multiple times in production [61, 67]. From these invocations, CodeCrunch records the mean and standard deviation of the *local* and *global* invocation periods. Here, *local* refers to the last n_l invocations, and *global* refers to all the invocations of the functions. The estimated invocation period P_{est} of a function is calculated as:

$$P_{est} = \left(\frac{|L_m - G_m|}{max(L_m, G_m)}\right)(L_m + L_s) + \left(1 - \frac{|L_m - G_m|}{max(L_m, G_m)}\right)(G_m + G_s)$$

Here, L_m and G_m refer to the mean local and global invocation periods, respectively. L_s and G_s refer to the standard deviation of the local and global periods, respectively. The calculation of P_{est} has two parts – one of which focuses on the global metrics and another focuses on the local metrics. The equation combines the mean and standard deviation of local and global invocation periods. Considering more than one standard deviation slightly deteriorates the results of mean service time as the P_{est} estimation becomes less accurate by considering outlier cases.

As the difference between the local and global mean invocation period increases, more importance is placed on the local period to calculate P_{est}. Serverless functions typically exhibit periodic invocation patterns with occasionally changing periods and multiple invocation period frequencies [61, 67]. This is why we design CodeCrunch's invocation prediction estimate by taking into consideration both local (short-term) and global (long-term) periodic patterns.

Note that, the success of CodeCrunch does not depend on the exact estimation of P_{est}. Instead, it only provides an estimate that a function will undergo a warm start if it is re-invoked within the interval P_{est}. In CodeCrunch, P_{est} is used to estimate how good a solution of a sub-problem is in terms of reducing the mean service time of invoked functions. In

CodeCrunch's implementation, *local* refers to the last ten invocations, but CodeCrunch's effectiveness does not change by more than 2.6% when it varies from the last 2 to the last 100 invocations. The global invocation period is re-setted after every 1000 invocations.

Similarly, CodeCrunch keeps track of the service time of functions in ARM and x86 processors from past executions with cold starts, warm starts without compression, and warm starts with compression (including decompression time). For all neighboring solutions around the starting point of SRE's gradient descent in each sub-problem space, CodeCrunch determines whether the functions will undergo a cold start or a warm start (from P_{est}, following the keep-alive time of each function of the solution). From this, CodeCrunch estimates the execution time (from the choice of compression, choice of the processor in the solution, following past ARM and x86 execution time records). From this, SRE determines the solution that minimizes the mean service time of the functions included in the sub-problem (gradient descent takes a step toward the direction of the steepest descent).

The optimal solution of each sub-problem is combined to form the solution of the original problem, per round of optimization by SRE. The functions that are not chosen by SRE for optimization in the present round, retain their previously chosen choices of keep-alive time, processor type, and compression choice for execution. However, the probability of such functions of being sampled by SRE for optimization in future rounds increases.

Formulation of CodeCrunch's Optimization. To put the optimization pass together, CodeCrunch attempts to minimize the estimated mean service time of functions invoked in any interval t, under a keep-alive budget constraint. It performs the optimization by making function compression choice, keep-alive time choice, and processor type choice for function execution.

Essentially, CodeCrunch solves the following problem.

$$\underset{j \in S_t}{\operatorname{argmin}} \sum_{i=1}^{N} CS_i(j) + EX_i(j)$$

Here, j denotes the sample chosen from the set of all possible solutions in S_t. CS_i denotes the cold-start time and EX_i denotes the execution time of the i^{th}. The value of execution and cold-start time depends on the processor type (ARM or x86), and compression choice specified in sample j. Also, if the keep-alive time of sample j for the i^{th} function is greater than the P_{est} of the function, the function undergoes a warm start, and $CS_i = 0$.

We empirically observe that SRE mainly compresses functions during periods of high invocation load. Compression and keep-alive time (determined by P_{est} and K_t) directly impact the cold-start time ($CS(j)$) and processor type selection impacts the execution time ($EX(j)$). Next, we discuss the implementation of function compression.

3.2 Serverless Function Compression with CodeCrunch

CodeCrunch applies intelligent compression on functions selected for compression by SRE. To understand how CodeCrunch performs in-memory compression, we first discuss the file system details of serverless instances.

How are serverless function instances stored? *OverlayFS*, a modern union filesystem, is a widely-used storage driver in serverless function instances [20]. It uses a layered approach where the files from the lower layers can be reused by multiple function instances or containers executing in the upper layers, thus requiring less in-memory storage. In Docker, OverlayFS has two directories on a Linux host, unified via a union mount. In the lower directory (*lowerdir*), a Docker image of the function is stored, which contains the base operating system required to run the function, the necessary runtime environments and libraries, and the source code of the function. This layer is immutable. The container layer is present in the upper directory (*upperdir*). It runs the serverless function instance. It creates its own temporary files and contains the input files to execute a function. The container layer is writable. When a container is created, the driver of OverlayFS merges the two layers to create a container mount.

How does CodeCrunch compress the serverless function instance? A reasonable design to compress a function instance is to compress the entire filesystem (from the root) of the container layer in the form of a tar archive. However, this approach has three challenges. First, the service provider might not have access to the running container due to security/privacy policies. Second, it is challenging to decide which files to compress – for example, if we compress tar and its associated files, decompressing upon function invocation may not be possible. Third, this design only compresses the files in the container layer, with no effect on the image layer.

Therefore, CodeCrunch follows a different design for function compression (Fig. 5). After a function finishes execution in a container, CodeCrunch uses *docker commit* to save the container's file changes/settings into a new image. Then, CodeCrunch directly compresses the committed image into a tar archive. Thereafter, the original image of the function and the committed image are both removed from the docker directory in the host machine, freeing the memory space. Upon invocation, the archived image is decompressed, the image is registered in the docker directory of the host machine (via *docker build*) and a container is started (via *docker run*). CodeCrunch stores the compressed image in Docker's storage volume, which is configured to be a RAM/memory disk. In this way, CodeCrunch compresses the filesystem of a Docker image and keeps it in memory for faster start-up, instead of compressing the memory contents of a running function. The compressed docker image reduces the amount

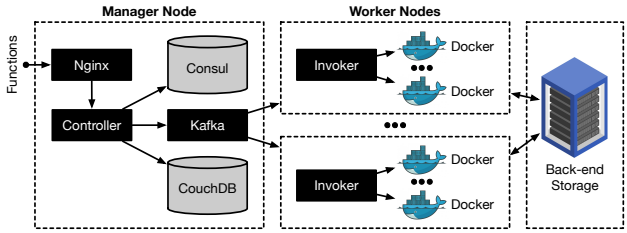

Figure 6. The manager node decides the keep-alive times and controls the execution of functions in the worker nodes.

of memory needed during the keep-alive period. The overhead to register the image and start a container is relatively negligible compared to a cold start. However, the decompression time is critical in CodeCrunch's case. Therefore, next, we discuss two important design considerations: *impact of decompression* and *selection of compression-friendly functions.*

How to determine which compressor to choose? For CodeCrunch's function compression to be effective, the decompression time should be lower than the regular cold start time of the function. CodeCrunch selects the *lz4* [12] compression algorithm as it provides a reasonable trade-off between decompression time and compression ratio. It provides over 2.5× compression ratio on average for our evaluated serverless functions, while the decompression cost is less than 35% of the average cold-start overhead. It is possible to achieve a higher compression ratio with alternative compression-focused algorithms (e.g., *xz* [15]), but they can increase the decompression time, and hence, negate the benefit of warm starts.

Which functions does CodeCrunch compress? As we saw earlier in Fig. 1, compression is not effective for all functions. This is especially true for functions that have large dependencies and hence, decompression time can eliminate the benefit of a warm start. Therefore, CodeCrunch's SRE learns the relative benefit of compression for each function over time and only compresses functions that benefit from it. Learning this behavior is relatively lightweight and robust to potentially changing function characteristics (e.g., input changes). This is because input changes often do not affect the container size and decompression time. Nevertheless, our evaluation (Sec. 5) shows our compression scheme's robustness against such changes. Next, we discuss how CodeCrunch's implementation integrates these design elements.

3.3 CodeCrunch's System Implementation

As visually depicted in Fig. 6, CodeCrunch's architecture consists of (a) a manager node, which makes the scheduling decisions regarding keeping functions alive, compressing, and executing them, and (b) the worker nodes, where the serverless functions are executed or kept alive in memory. CodeCrunch's architecture is based on that of OpenWhisk [64], a widely used, open-sourced serverless scheduler.

The first component of the manager node is an *Nginx Front-end* [52]. It exposes the public-facing HTTP endpoints to the clients and serves as a reverse proxy for API and SSL termination to offload function requests to the next layers in the pipeline. The *controller* is the next component in OpenWhisk, residing in the manager node, which CodeCrunch modifies to optimize for function service time. With every function request, the modified controller attaches a *json* tag to specify the scheduling decisions for the invoked functions.

Recall that CodeCrunch uses a mix of server types (x86 and ARM) to exploit the trade-offs between performance and keep-alive costs. Next, *CouchDB* [8] provides the load level on the worker nodes to the controller. CouchDB receives this information from *Consul* [50], which acts as a distributed profiler to keep track of the health of the worker nodes and their resource usage. *Apache Kafka data pipeline* [40] sends messages from the controller to all the worker nodes. These messages are commands to execute a function, along with the CodeCrunch-determined keep-alive period.

Each worker node has an *invoker*, which prepares for execution by first downloading all the metadata of a serverless function along with its executable from a backend storage server. Then, it creates a docker image of the function. If the JSON tag of the function, as attached by the controller component described earlier, indicates it needs to be kept alive after execution, then the invoker uses *execve* system call to load and keep alive the application contents in the memory of the node. A *back-end storage server* contains all the metadata of serverless functions, which includes the executables, required libraries, runtime environment, and input files. Since serverless functions are stateless and cannot communicate directly with each other during their execution, this back-end storage server is also used as the medium of communication among the functions, as needed. Next, we summarize CodeCrunch.

3.4 CodeCrunch's Design and Implementation Summary

At every optimization interval, CodeCrunch's optimizer generates the set of samples with processor choice, compression choice, and keep-alive time to form the optimization space. Thereafter, CodeCrunch's SRE forms different sub-problems and estimates the best solutions from the optimal solutions of the sub-problems. CodeCrunch uses this to determine the keep-alive time, compression, and processor choice of the invoked functions. Since CodeCrunch's SRE minimizes service times, it chooses to perform function compression mainly during periods of high invocation load (function compression increases the service time of a function compared to an uncompressed warm start) when it is otherwise not possible to operate under the keep-alive budget. Next, we discuss the experimental methodology used to evaluate CodeCrunch.

4 CodeCrunch: Experimental Methodology

Cluster Setup. CodeCrunch spawns serverless functions on Docker [7] containers on ARM and x86 worker nodes, controlled by a manager node. In our setup, the worker nodes consist of Amazon m5 EC2 virtual machines as servers with x86 processors (usage cost: $0.384/hour) and t4g as servers with ARM processors (usage cost: $0.2688/hour). These node types closely represent the node options available on AWS Lambda in terms of x86 and ARM processors. We cannot directly use AWS Lambda offerings because we cannot control the keep-alive time on their x86 and ARM-based processors.

Both types of worker nodes have 8 cores and 32 GB of memory. They vary by processor speed, memory bandwidth, and I/O bandwidth, which makes the execution time of functions different on the two node types. In our experiments, we allocate the same total capital cost budget to both ARM and x86 servers, resulting in 18 ARM and 13 x86 nodes (similar to other state-of-the-art serverless schedulers [61]). In our evaluation, we found that the CodeCrunch's benefits are not sensitive to the cost ratios of the ARM and x86 servers, and CodeCrunch is effective across all different configurations of the number of ARM and x86 nodes.

When a function is kept alive in the memory of a server, it occupies all the other computing resources for the given amount of time, and hence, the keep-alive cost is directly proportional to the usage cost of a node per unit of time. We use the Amazon S3 bucket for designing the back-end storage as S3 provides low I/O latency during high request rates compared to other storage services like Amazon EFS. CodeCrunch uses an Intel Silver server with 20 logical cores, 46 GB memory, and 5 Gbps network bandwidth as the manager node. The choice of a powerful manager node allows us to implement a capable controller with high network bandwidth for low-latency message passing between the manager and worker nodes under high request loads.

Workloads and Function Invocation Trace. The serverless workloads used in CodeCrunch are taken from SeBS [16] and ServerlessBench benchmark suites [84]. These benchmarks perform a wide range of operations like image processing, linear algebra, data analytics, stream processing, online compilation, etc. Both of these benchmark suites are widely used by the serverless research community [9, 61, 83]. All the functions from the evaluated serverless benchmark suites can be executed on both ARM and x86 platforms. Many serverless functions are written in interpretable languages and run on both platforms, without compilation. If a case arises that a function can only run in one of the platforms, CodeCrunch focuses on performing keep-alive time and compression choice optimizations only. When executing serverless functions, we ensure that the invocation pattern of the serverless function represents the serverless invocation characteristics on the production-scale cloud platforms. To achieve this, we use Microsoft Azure Cloud Function trace [67], which is a widely used trace in serverless research, to invoke serverless functions with a production invocation pattern. Our production-grade trace consists of function invocation data of two weeks from Microsoft Cloud, with more than 200,000 unique functions. This data is sampled at an interval of one minute. For the experimental purpose, we uniformly divide the invocation of functions within each one-minute interval. Along with the invocation characteristics, this trace also provides the execution time and memory requirements of the functions. We use these values to find the nearest matching function from our benchmark pool to represent the corresponding serverless function.

Baseline. The technique in the **Serverless in the Wild (SitW) paper [67]** is chosen as our baseline as it is an open-source serverless workload manager which has been deployed in production on Microsoft Azure Cloud. SitW uses a combination of regression-based and histogram-based invocation probability determination for predicting which functions will be invoked. We duly underscore that SitW was designed for a homogeneous cluster, but we modified it to make it heterogeneity-aware, to improve its effectiveness (but kept the name the same for simplicity of reference and to acknowledge the practicality of the technique).

Other Relevant and Complementary Techniques. We demonstrate how CodeCrunch can enhance the effectiveness of other state-of-the-art serverless scheduling techniques.

(1) **FaasCache [24]:** FaasCache uses a greedy caching approach to predict and decide which functions to keep alive. It uses several serverless specific considerations like memory utilization, invocation frequency, and recency of execution to determine functions to keep alive. We also use the ARM or x86 processor affinity of functions and their different keep-alive costs to make FaasCache aware of both processor types.

(2) **IceBreaker [61]**: To the best of our knowledge, IceBreaker is the only existing method that uses a heterogeneous mix of servers for the execution of serverless functions. IceBreaker uses Fourier Transform to learn about function invocation periodicity and predict when to warm functions up. However, IceBreaker suffers from multiple challenges and practical limitations including high decision-making overhead, and profiling of all functions on different nodes. Also, IceBreaker does not perform function compression and scheduling in x86 and ARM processors.

(3) **Oracle.** This technique utilizes the future knowledge of all the function invocations collected offline. It keeps functions alive by appropriately choosing between x86 or ARM to minimize the mean service time. It is not feasible to implement Oracle in a real system. However, it serves as the upper limit of performance estimation, even when future information is known.

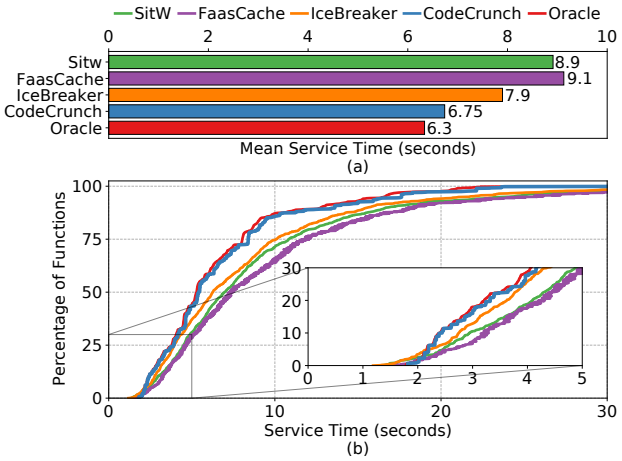

Figure 7. CodeCrunch improves service time over SitW and is close to the Oracle due to a higher rate of warm starts.

Figures of Merit. The primary metric is the **service time**, which is the sum of the execution time, cold-start time, and wait time before a function is executed (wait time occurs only when the cluster is fully occupied and has no more capacity to execute a function). The service time is strongly correlated with the **percentage of warm starts** and the **keep-alive cost**, which is the dollar amount spent by a cloud provider to keep a function alive. Typically, a higher percentage of warm starts leads to lower service time. However, a higher percentage of warm starts often requires a high keep-alive cost. The keep-alive cost is proportional to the keep-alive time and depends on the type of server used for function warm up (ARM or x86). CodeCrunch's design does not aggravate the interference concern compared to other methods. CodeCrunch does not require the co-location of multiple functions on the same core at the same time, to reap compression and processor heterogeneity benefits. Natural overlaps of decompression/compression with other function executions are accounted for in CodeCrunch's results. *In the evaluation, the keep-alive budget of CodeCrunch is normalized with respect to the keep-alive cost incurred by the baseline technique, SitW [67], for fair comparison.*

5 Evaluation and Analysis

Effectiveness of CodeCrunch. *CodeCrunch significantly outperforms the baseline and state-of-the-art serverless techniques in minimizing service times. CodeCrunch reduces the service time of almost all functions compared to competing techniques.* Fig. 7(a) shows that CodeCrunch improves the average service time by 32% compared to the baseline solution (SitW) and is within 6% of the Oracle solution. CodeCrunch's total keep-alive budget is the same as the total keep-alive cost expenditure of SitW – for fair comparison against SitW. This total keep-alive budget is divided equally among all periods of time across the two weeks of function invocations, to derive the keep-alive budget spending rate of CodeCrunch.

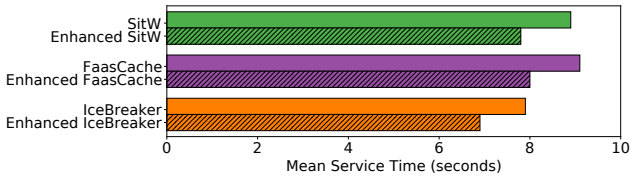

Figure 8. CodeCrunch's optimizations can further improve the effectiveness of existing techniques (enhanced means augmented with compression and x86/ ARM selection).

FaasCache and Icebreaker are other recent competing techniques that predict function invocation time based on their historic invocation pattern and warm-up costs. Unlike SitW, which primarily relies on the simplicity and practicality of the keep-alive strategy, FaasCache and Icebreaker require high-overhead intelligence to accurately predict the next invocation of the function. Since CodeCrunch keeps functions alive after invocation, it does not need to predict the invocation time. Because of the difference in approach, CodeCrunch is not necessarily competitive, but complementary. CodeCrunch leverages in-memory compression, and sequential random embedding to obtain a 6.75-second mean service time, compared to 9.1 seconds and 7.9 seconds by FaasCache and IceBreaker, respectively. This translates to over 34% and 17% improvement in the mean service time. The Oracle mean service time values for warm starts (uncompressed), warm starts (compressed), and cold starts (uncompressed) are 6.3 seconds, 6.99 seconds, and 10.2 seconds, respectively, when the best processor type choice is selected for each function execution. The mean decompression time of functions is 0.37 seconds, while the 75^{th} percentile and maximum decompression time are 0.52 and 0.68 seconds, respectively. The compression time is 1.57, 1.82, and 2.01 seconds for the mean, 75^{th} percentile, and maximum compression time, respectively. However, the compression time is not in the critical path of CodeCrunch's operation.

Next, Fig. 7(b) demonstrates that the benefits of CodeCrunch are not simply because of a few functions or long-running functions, CodeCrunch improves the service time of most functions throughout the invocation trace. This is because CodeCrunch achieves more warm starts than SitW due to scheduling among ARM and x86 processors, and in-memory function compression. For only 6% of function invocations, CodeCrunch has more service time than FaasCache and IceBreaker. These are the functions that are invoked infrequently with a high re-invocation period (> 60 minutes), and hence, as expected, CodeCrunch does not keep them alive to minimize keep-alive cost expenditure. CodeCrunch provides 8.6%, 12.1%, and 11.7% reduction of service time over Icebreaker, FaasCache, and SitW, respectively even for short-running functions (service time < 1 second). Next, we show that CodeCrunch can indeed be applied to further augment and enhance SitW, FaasCache, and IceBreaker.

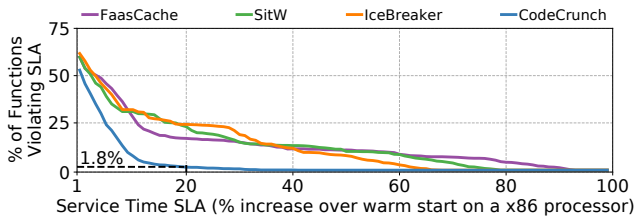

Figure 9. CodeCrunch's implementation can be modified to operate under a service time SLA.

To demonstrate this, we enhanced SitW, FaasCache, and IceBreaker by adding in-memory compression and choice of execution on both x86 or ARM processors as in CodeCrunch. Fig. 8 shows the results of adding these enhancements. Note that the original pre-warming/keep-alive decision-making was kept intact in the alternative state-of-the-art schemes (ARIMA in SitW, and Greedy caching in FaasCache, FFT in IceBreaker), while CodeCrunch uses SRE-based optimization. All three alternative state-of-the-art schemes consistently observe over 10% improvement in service time using two key components of CodeCrunch's design. Interestingly, we observe that "enhanced SitW" performs similarly or slightly better than recent techniques (IceBreaker and FaasCache). This shows that we enjoy better performance while maintaining the simplicity, production-level practicality, and effectiveness of SitW, without the complex overhead of more recent techniques such as IceBreaker, that rely on complex FFT-based prediction.

Can CodeCrunch operate following an SLA? CodeCrunch's original design goal is to minimize service time, which is the usual optimization metric of serverless schedulers due to the high variability in service time of production serverless functions [24, 60, 61, 67]. However, when CodeCrunch's design is modified to operate under an SLA, it still outperforms the competing techniques (Fig. 9). For example, when the SLA is 20% allowed percentage increase in service time of a function compared to a warm start (uncompressed) on an x86 processor, CodeCrunch violates the SLA for only 1.8% of the functions, while all other competing techniques violate it for more than 19% of the functions.

Reasons for CodeCrunch's effectiveness. Next, we analyze and demonstrate how the key ideas behind CodeCrunch, individually contribute toward its effectiveness. First, we analyze the effect of CodeCrunch's operation under a keep-alive budget constraint. From Fig. 10(a) we see that CodeCrunch achieves more percentage of warm starts than the baseline technique, SitW. This is because CodeCrunch is better at utilizing the available keep-alive budget than SitW. During periods of low function invocation load, CodeCrunch uses less keep-alive memory. This saves keep-alive costs. CodeCrunch uses the saved-up keep-alive cost to increase the keep-alive budget during periods of high invocation load, when more functions need to be kept alive (Fig. 10(b)). Just

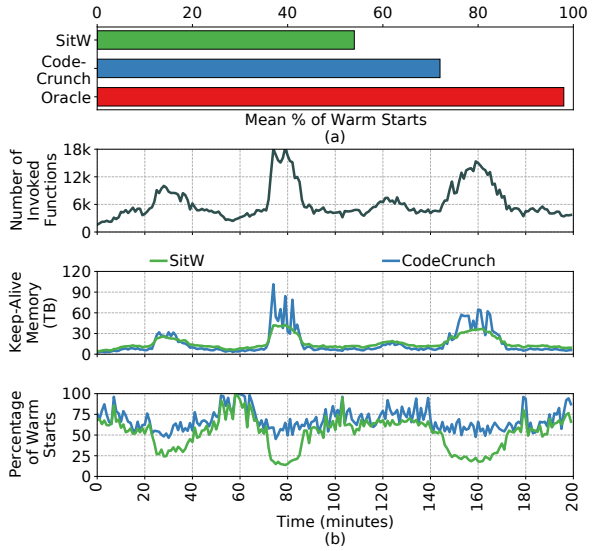

Figure 10. CodeCrunch's keep-alive budget creditor helps us achieve more warm starts, even during peak load periods.

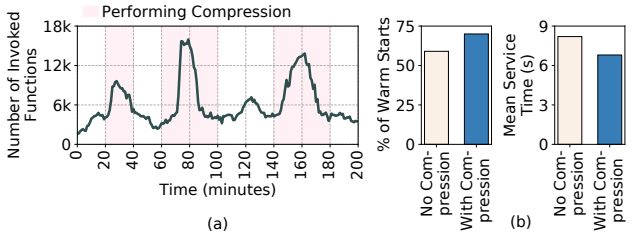

Figure 11. Compression of functions increases the percentage of warm starts during periods of high invocations.

effectively managing the available keep-alive budget helps CodeCrunch in achieving more warm starts than SitW by 18% points. The data points in Fig. 10(b) represent the total keep-alive memory budget for an entire minute, not per unit time (*i.e.*, a second). Also, per unit time, if the available keep-alive budget is more than the capacity of the available memory, CodeCrunch keeps alive functions only up to the capacity of the available memory.

Next, we analyze the role of compression and its impact on warm starts. Fig. 11(a) shows the three major periods of time (shaded regions), where CodeCrunch performs in-memory function compression due to high invocation load. By performing function compression, CodeCrunch increases the overall percentage of warm starts by more than 10% points, resulting in the improvement of service time (Fig. 11(b)).

Even when a serverless function instance (container/microVM) starts up very fast, CodeCrunch's compression provides significant benefits. This is because the cold-start time bottleneck continues to be large, and compression allows a longer keep-alive period within a keep-alive budget. For example, with Firecracker [3] microVMs (fast start-up time), CodeCrunch achieves a mean service time of 6.66 seconds with compression, and 8.05

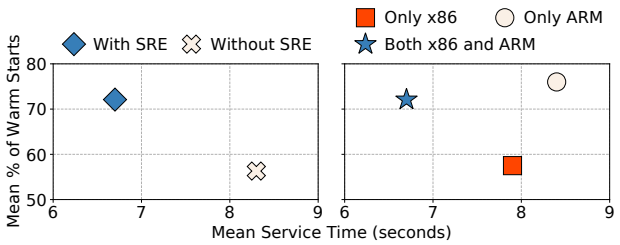

Figure 12. Using different kinds of processors and SRE helps CodeCrunch to minimize service times and cold starts.

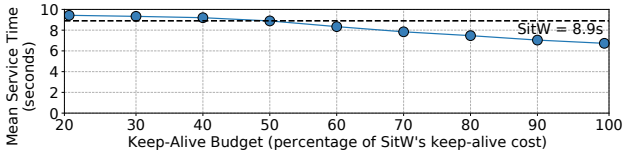

Figure 13. CodeCrunch improves the service time across different keep-alive budgets (the dashed line denotes that SitW achieves a mean service time of 8.9 seconds).

seconds without compression. With Docker containers, CodeCrunch achieves a mean service time of 6.75 seconds with compression, and 8.15 seconds without compression.

CodeCrunch's SRE speeds up the optimization process to find the optimal processor type (x86 or ARM), keep-alive time, and compression choice of functions. If instead of SRE's intelligent sampling, optimization is performed over all the invoked functions, the resultant solution is sub-optimal compared to SRE, within SRE's optimization time. This can be seen from Fig. 12, which shows that SRE helps CodeCrunch to achieve a service time improvement of 19%, by improving the percentage of warm starts. Fig. 12 also shows that CodeCrunch's choice of executing functions on both ARM and x86 processors actually helps it to reduce service time over homogeneous execution on only ARM or only x86 processors. This is because, under the same keep-alive budget, the number of functions that can be warmed up on servers with x86 processors is lower than the number of functions that can be warmed up on servers with ARM processors. At the same time, since a majority of functions achieve lower execution time on an x86 processor, only ARM execution increases the mean service time. CodeCrunch efficiently utilizes servers with both x86 and ARM processors, to improve service time. This shows that when CodeCrunch is run on a heterogeneous system, not only do the users save on service time and costs but also, the service providers can reduce their keep-alive cost and provide service to more users, increasing their revenue. In terms of absolute numbers, CodeCrunch achieves a mean service time of 6.75 seconds. CodeCrunch (without compression), CodeCrunch (with only x86 processors), CodeCrunch (with only ARM processors), and CodeCrunch with a fixed 10-minute keep-alive time policy achieves a mean service time of 8.15 seconds, 7.87 seconds, 8.4 seconds, and 7.38 seconds, respectively.

Overhead of CodeCrunch. *CodeCrunch's decision-making is practical even when the number of functions grows, due to CodeCrunch's sequential random embedding.* A major barrier to adopting function invocation prediction techniques, such as IceBreaker and FaasCache, is that these techniques need to capture the probability of invocations of all the functions to make warm-up decisions. As the number of functions grows in a production serverless system, this approach incurs high decision-making overhead. Typically, FaasCache has a lower overhead than IceBreaker. In contrast, CodeCrunch only makes scheduling, compression, and keep-alive decisions for the functions invoked at the current time interval. Hence, it has much lower decision-making overhead compared to the competing techniques – CodeCrunch spends only 4.52% of function service time to make decisions, while IceBreaker spends 30% of function service time, and FaasCache spends 21% of function service time to make scheduling decisions, for 10 million unique functions in the system. This overhead is similar to SitW, which also employs a keep-alive strategy. CodeCrunch provides significantly higher performance than practical techniques like SitW and high-overhead techniques such as IceBreaker and FaasCache. When only recently invoked functions (within the last 1 hour) are considered, Icebreaker and Faascache's overhead are 7.12%, and 5.06% of mean service time, respectively (similar to 4.52% for CodeCrunch). However, IceBreaker and Fasscache's service time increases by 5.21%, and 5.03% respectively - *i.e.*, the gap in effectiveness compared to CodeCrunch increases. We note that the decision-making overhead occurs at the time instant when warm-up decisions are being made, and hence, they do not directly affect the function scheduling and compression decisions (only keep-alive decisions). These overheads are accounted for in the evaluation.

Sensitivity to keep-alive budget. *CodeCrunch continues outperforming the baseline at different keep-alive budget targets.* Unlike existing techniques, CodeCrunch allows the service provider to set the per unit time keep-alive budget, thus allowing the provider to control the resources to be used for keeping alive functions. From Fig. 13, we observe CodeCrunch's performance under different keep-alive cost budgets. Note that, CodeCrunch observes almost similar performance as that of SitW in terms of service time, at 0.5 times the keep-alive cost expenditure of SitW. Also, CodeCrunch's achieved service time is only 5% more than SitW's mean service time at a keep-alive cost expenditure of just 25% of SitW's total keep-alive cost expenditure. These benefits are a result of the correct estimation of the function re-invocation period, function compression, and proper scheduling by CodeCrunch's SRE.

CodeCrunch is effective even when the pricing of the model is the same for both kinds of processors. Our results confirm this - the mean service time changes to 6.87s when

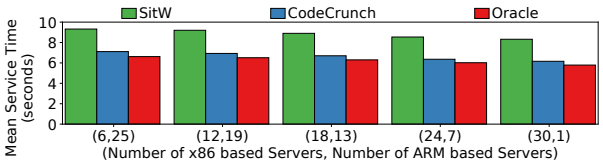

Figure 14. CodeCrunch improves service time for different numbers of x86 and ARM processors.

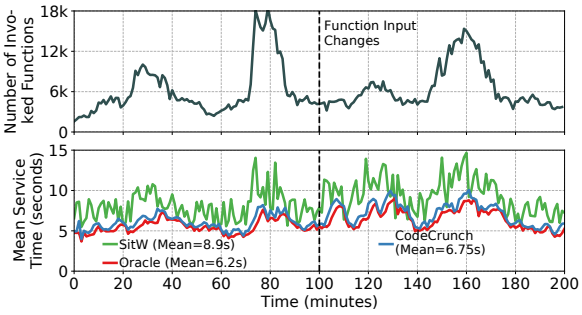

Figure 15. CodeCrunch adapts to input changes and invocation bursts compared to the baseline technique (SitW).

both processor type has the same price (it was 6.75s in the original pricing model). This is because scheduling, compression, and keep-alive time decisions are performed based on relative x86/ARM execution time, compression-friendliness, and invocation frequency, respectively. Only the keep-alive budget is affected by the pricing model.

Sensitivity to the number of x86 and ARM-based nodes. *CodeCrunch outperforms the baseline solution for different numbers of x86 and ARM-based nodes.* One of the main sources of CodeCrunch's improvement comes from proper function scheduling in Arm and x86-based processors. Even with varying numbers of x86 and Arm-based nodes for serverless execution, CodeCrunch's performance is closer to the Oracle solution, compared to SitW, as seen in Fig. 14. On average, CodeCrunch is 35% closer to Oracle, compared to SitW. In general, the service time increases with a decrease in the number of x86 nodes as more functions have lower execution times on an x86 node compared to an ARM node.

Effectiveness against varying input parameters and load. Fig. 15 demonstrates the effectiveness of CodeCrunch under varying input parameters and load conditions. Our experiment is conducted by changing function inputs at an arbitrarily chosen point and changing the load level. However, CodeCrunch is not informed of these changes. CodeCrunch detects these changes automatically and adjusts quickly and closely follows the Oracle solution's service time curve. In contrast, the baseline is not always able to follow the Oracle curve and experiences significant performance degradation during peak loads. CodeCrunch performs better because SRE probabilistically re-selects functions for optimization based on the characteristics of other invoked functions.

6 Related Work

Problems in serverless computing. Serverless computing is gaining interest due to its attractive benefits like elastic resource-unaware scheduling [14, 29, 30, 32, 63, 65, 75]. The general interest has led to several characterization and analysis related studies of serverless platforms [28, 42, 45, 76, 84]. They range from characterization of production serverless platforms to efficiency of specific types of workloads [25, 37–39, 44, 49, 60]. These works establish that (1) high cold start overhead is harmful as most serverless functions are short running [67, 79, 80], (2) sub-optimal scheduling of serverless functions results in slowdown due to resource contention [13, 43, 66], and (3) execution is costly due to resource cost and keep-alive cost [22, 28, 51, 70, 74].

Existing approaches. The existing techniques aim to solve these problems by developing serverless-specific kernels or execution environments to solve the inefficiencies in serverless [4, 18, 24, 27, 33, 49, 69–71, 73, 85]. This has led to works like Catalyzer [20], which develops sandboxing mechanisms for sub-millisecond function startup and strong isolation techniques to avoid resource contention. SOCK [53] caches frequently used files and libraries in memory to ensure a fast start-up. While these approaches are effective in speeding up serverless execution, often their effectiveness is application-specific [26, 35, 36, 47, 56]. For example, an I/O bound application will suffer from sandboxing as in Catalyzer, due to strong isolation between serverless function instances. Again, caching-based techniques cannot be used for applications with large dependencies as they increase the keep-alive cost [6, 10, 11, 19, 23, 55, 81]. This led to the development of serverless-specific scheduling techniques [18, 46, 68]. These techniques focus on developing prediction strategies to minimize serverless cold starts [21, 31, 71, 72, 77]. FaasCache [24] uses several concepts of memory caching to warm up serverless functions. SitW [67] and IceBreaker [61] learn invocation patterns over time to warm up functions. Orion [48] co-locates multiple concurrent invocations in a single VM and pre-warms them to reduce cold starts. These approaches have a large decision-making overhead and can increase the keep-alive cost without any bound [5, 9, 54, 58, 59, 61].

7 Conclusion

CodeCrunch provides a low-overhead and practical solution to improve serverless service time by effectively keeping functions alive, scheduling them on heterogeneous processors, and compressing them. CodeCrunch improves service time by more than 10% when combined with other serverless schedulers. We hope that CodeCrunch's design elements will help the service providers improve performance under limited resources or budget constraints.

Acknowledgement. We thank Rodrigo Fonseca (our shepherd), and the reviewers for their constructive feedback. This work is supported by NSF Award 1910601, 2124897, Northeastern University, and Rice University.

8 Artifact Appendix

8.1 Abstract

Serverless functions are kept alive in the memory of servers by service providers to achieve warm starts. Keeping alive functions consumes compute and memory resources, resulting in a keep-alive cost expenditure. Servers of different architectures (x86 or ARM) have different keep-alive cost expenditures, per unit time. CodeCrunch performs function compression to keep more functions alive within a given keep-alive budget. It also schedules these functions in x86 or ARM-based servers to optimize for serverless service time.

8.2 Artifact check-list

- **Algorithm:** Sequential Random Embedding
- **Program:** We use benchmarks from ServerlessBench and SeBS benchmark suites. All the evaluated benchmarks are included in the artifact.
- **Data set:** The invocation of serverless functions follow Microsoft Azure Trace. The trace used in CodeCrunch is included in the artifact.
- **Run-time environment:** Python 3.10
- **Hardware:** AWS EC2 servers: c5.4xlarge for the manager node, and multiple t4g.2xlarge and m5.2xlarge worker nodes.
- **Metrics:** Service time (cold-start time + execution time).
- **How much disk space required?:** 512 GB
- **How much time is needed to prepare workflow (approximately)?:** 2 hours
- **How much time is needed to complete experiments (approximately)?:** 1 day
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.8347244

8.3 Description

8.3.1 How to access. The artifact is located at: https://zenodo.org/doi/10.5281/zenodo.8347244.

8.3.2 Hardware dependencies. This artifact is evaluated in the AWS platform using EC2 servers. It requires the user to set up a manager node. All the worker nodes are automatically set up by CodeCrunch's controller running from the manager node. We have set the manager node to be a c5.4xlarge EC2, with 512 GB of storage. However, any other node or VM with similar bandwidth and memory capacity can be chosen.

8.3.3 Software dependencies. Python 3.10, pip3, awscli, boto3, lz4, and Docker are the required dependencies.

8.4 Installation

After spawning a manager node, perform the following steps to complete the installation process:
(1) Install *Python 3.10* and *pip3*.
(2) Install *awscli* in the manager node and set the *access key*, *secret access key*, and the *region name*.
(3) Install *boto3*, and *lz4* via *pip3*. Install *Docker* via *apt-get*.
(4) In AWS, set a security group with full access to all EC2 services. After setting the security group, you should have a security group ID.
(5) In AWS, create a key pair (resulting in a *.pem* file), which will be used to SSH to both manager and worker nodes. Copy this *.pem* file into the */home/ubuntu/* directory of the manager node.
(6) Extract all the contents of the artifact zip in the */home/ubuntu/* directory.
(7) Open the *main.py* file and go to the *main* function. Set the *User Account Details* part with the information gathered from the previous step. Set the values of the following variables: *region_name*, *key_pair_name*, *ssh_filename*, and *security_group_id*.
Follow the *README.md* file in the artifact for more information.

8.5 Evaluation and expected results

CodeCrunch generates an output file called, "service_time.txt", which captures the service time of all the functions being executed via CodeCrunch.

References

[1] AWS Lambda pricing, url: https://aws.amazon.com/lambda/pricing/.
[2] Lambda instruction set architectures, url: https://docs.aws.amazon.com/lambda/latest/dg/foundation-arch.html.
[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, pages 419–434, 2020.
[4] Siddharth Agarwal, Maria A Rodriguez, and Rajkumar Buyya. A reinforcement learning approach to reduce serverless function cold start frequency. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 797–803. IEEE, 2021.
[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards high-performance serverless computing. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 923–935, 2018.
[6] Waleed Ali, Siti Mariyam Shamsuddin, Abdul Samad Ismail, et al. A survey of web caching and prefetching. *Int. J. Advance. Soft Comput. Appl*, 3(1):18–44, 2011.
[7] Charles Anderson. Docker [software engineering]. *Ieee Software*, 32(3):102–c3, 2015.
[8] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide: time to relax.* " O'Reilly Media, Inc.", 2010.
[9] Ataollah Fatahi Baarzi, George Kesidis, Carlee Joe-Wong, and Mohammad Shahrad. On merits and viability of multi-cloud serverless. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 600–608, 2021.
[10] Abdullah Balamash and Marwan Krunz. An overview of web caching replacement algorithms. *IEEE Communications Surveys & Tutorials*, 6(2):44–56, 2004.
[11] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive ttl-based caching for content delivery. In *Proceedings of the 2017 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, pages 45–46, 2017.
[12] Peter Boncz, Thomas Neumann, and Viktor Leis. Fsst: fast random access string compression. *Proceedings of the VLDB Endowment*,

13(12):2649–2661, 2020.

[13] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the" micro" back in microservice. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 645–650, 2018.

[14] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[15] Rudi Cilibrasi and Paul MB Vitányi. Clustering by compression. *IEEE Transactions on Information theory*, 51(4):1523–1545, 2005.

[16] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021.

[17] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. *Advances in neural information processing systems*, 28, 2015.

[18] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, pages 356–370, 2020.

[19] Mostafa Dehghan, Laurent Massoulie, Don Towsley, Daniel Sadoc Menasche, and Yong Chiang Tay. A utility optimization approach to network cache design. *IEEE/ACM Transactions on Networking*, 27(3):1013–1027, 2019.

[20] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[21] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[22] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2020.

[23] Andrés Ferragut, Ismael Rodríguez, and Fernando Paganini. Optimizing ttl caches under heavy-tailed demands. *ACM SIGMETRICS Performance Evaluation Review*, 44(1):101–112, 2016.

[24] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[26] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 199–208. IEEE, 2019.

[27] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C Nachiappan, Mahmut Taylan Kandemir, and Chita R Das. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference*, pages 280–295, 2020.

[28] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidamabaram Nachiappan, Ram Srivatsa Kannan, Mahmut Taylan Kandemir, and Chita R Das. Characterizing bottlenecks in scheduling microservices on serverless platforms. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1197–1198. IEEE, 2020.

[29] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[30] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.

[31] Hongseok Jeon, Seungjae Shin, Chunglae Cho, and Seunghyun Yoon. Deep reinforcement learning for qos-aware package caching in serverless edge computing. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2021.

[32] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[33] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 158–164, 2019.

[34] Arzu Gorgulu Kakisim, Sibel Gulmez, and Ibrahim Sogukpinar. Sequential opcode embedding-based malware detection method. *Computers & Electrical Engineering*, 98:107703, 2022.

[35] George Kesidis. Overbooking microservices in the cloud. *arXiv preprint arXiv:1901.09842*, 2019.

[36] Young Ki Kim, M Reza HoseinyFarahabady, Young Choon Lee, and Albert Y Zomaya. Automated fine-grained cpu cap control in serverless computing platform. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2289–2301, 2020.

[37] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 759–773, 2018.

[38] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 789–794, 2018.

[39] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 427–444, 2018.

[40] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[41] Danielle Lambion, Robert Schmitz, Robert Cordingly, Navid Heydari, and Wes Lloyd. Characterizing x86 and arm serverless performance variation. 2022.

[42] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450. IEEE, 2018.

[43] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408. IEEE, 2020.

[44] Xiayue Charles Lin, Joseph E Gonzalez, and Joseph M Hellerstein. Serverless boom or bust? an analysis of economic incentives. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[45] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.

[46] Wes Lloyd, Minh Vu, Baojia Zhang, Olaf David, and George Leavesley. Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 195–200. IEEE, 2018.

[47] Kunal Mahajan, Daniel Figueiredo, Vishal Misra, and Dan Rubenstein. Optimal pricing for serverless computing. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[48] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[49] Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing*, 2020.

[50] Shivakant Mishra, Larry L Peterson, and Richard D Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87, 1993.

[51] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[52] Clément Nedelcu. *Nginx HTTP Server*. Packt Publishing, 2013.

[53] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 57–70, 2018.

[54] Panos Patros, Josef Spillner, Alessandro V Papadopoulos, Blesson Varghese, Omer Rana, and Schahram Dustdar. Toward sustainable serverless computing. *IEEE Internet Computing*, 25(6):42–50, 2021.

[55] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.

[56] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 193–206, 2019.

[57] Hong Qian, Yi-Qi Hu, and Yang Yu. Derivative-free optimization of high-dimensional non-convex functions by sequential random embeddings. In *IJCAI*, pages 1946–1952, 2016.

[58] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa $ t: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 122–137, 2021.

[59] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–17, 2021.

[60] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Mashup: making serverless computing useful for hpc workflows via hybrid execution. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 46–60, 2022.

[61] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[62] Gideon Schechtman. Random embeddings of euclidean spaces in sequence spaces. *Israel Journal of Mathematics*, 40(2):187–192, 1981.

[63] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J Yadwadkar, Raluca Ada Popa, Joseph E Gonzalez, Ion Stoica, and David A Patterson. What serverless computing is and should become: The next phase of cloud computing. *Communications of the ACM*, 64(5):76–84, 2021.

[64] Michele Sciabarrà. *LEARNING APACHE OPENWHISK: developing open serverless solutions*. O'Reilly Media, 2019.

[65] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: A survey of opportunities, challenges, and applications. *ACM Computing Surveys (CSUR)*, 2019.

[66] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1063–1075, 2019.

[67] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 205–218, 2020.

[68] Jiacheng Shen, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 194–204. IEEE, 2021.

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

[70] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, pages 1–13, 2020.

[71] Khondokar Solaiman and Muhammad Abdullah Adnan. Wlec: A not so cold architecture to mitigate cold start problem in serverless computing. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 144–153. IEEE, 2020.

[72] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–10. IEEE, 2020.

[73] Amoghvarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*, pages 19–24, 2019.

[74] Davide Taibi, Josef Spillner, and Konrad Wawruch. Serverless computing-where are we now, and where are we heading? *IEEE Software*, 38(1):25–31, 2020.

[75] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: Affordable, scalable, and accurate {GNN} training with distributed {CPU} servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 495–514, 2021.

[76] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021.

[77] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. Mitigating cold start problem in serverless computing: A reinforcement learning approach. *IEEE Internet of Things Journal*, 2022.

[78] Ashish Venkat and Dean M Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 121–132. IEEE, 2014.

[79] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. Faasnet: Scalable and fast provisioning of custom serverless containerruntimes at alibaba cloud function compute. *arXiv preprint arXiv:2105.11229*, 2021.

[80] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.

[81] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal prefetching without the off-chip metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 996–1008, 2019.

[82] Dong Xie, Yang Hu, and Li Qin. An evaluation of serverless computing on x86 and arm platforms: Performance and design implications. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 313–321. IEEE, 2021.

[83] Hanfei Yu, Athirai A Irissappane, Hao Wang, and Wes J Lloyd. Faasrank: Learning to schedule functions in serverless platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 31–40. IEEE, 2021.

[84] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 30–44, 2020.

[85] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204, 2020.