Characterization and Reclamation of Frozen Garbage in Managed FaaS Workloads

Ziming Zhao
Institute of Parallel and Distributed Systems, SEIEE,
Shanghai Jiao Tong University
Shanghai, China
dumplings_ming@sjtu.edu.cn

Mingyu Wu
Institute of Parallel and Distributed Systems, SEIEE,
Shanghai Jiao Tong University
Shanghai AI Laboratory
Shanghai, China
mingyuwu@sjtu.edu.cn

Haibo Chen
Institute of Parallel and Distributed Systems, SEIEE,
Shanghai Jiao Tong University
Engineering Research Center for Domain-specific
Operating Systems, Ministry of Education, China
Shanghai, China
haibochen@sjtu.edu.cn

Binyu Zang
Institute of Parallel and Distributed Systems, SEIEE,
Shanghai Jiao Tong University
Engineering Research Center for Domain-specific
Operating Systems, Ministry of Education, China
Shanghai, China
byzang@sjtu.edu.cn

Abstract

FaaS (function-as-a-service) is becoming a popular workload in cloud environments due to its virtues such as auto-scaling and pay-as-you-go. High-level languages like JavaScript and Java are commonly used in FaaS for programmability, but their managed runtimes complicate memory management in the cloud. This paper first observes the issue of *frozen garbage*, which is caused by freezing cached function instances where their threads have been paused but the unused memory (e.g., garbage) is not reclaimed due to the semantic gap between FaaS and the managed runtime. This paper presents the first characterization of the negative effects induced by frozen garbage with various functions, which uncovers that it can occupy more than half of FaaS instances' memory resources on average. To this end, this paper proposes *Desiccant*, a freeze-aware memory manager for managed workloads in FaaS, which reclaims idle memory resources consumed by frozen garbage from managed runtime instances and thus notably improves memory efficiency. The evaluation on various FaaS workloads shows that *Desiccant* can reduce FaaS functions' peak memory consumption by up to 6.72×. Such saved memory consumption allows caching more FaaS instances to reduce the frequency of cold boots (creating instances before function execution) and p99 latency by up to 4.49× and 37.5%, respectively.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Runtime environments**.

Keywords: Language Runtime, Function-as-a-Service, Garbage Collection

ACM Reference Format:
Ziming Zhao, Mingyu Wu, Haibo Chen, and Binyu Zang. 2024. Characterization and Reclamation of Frozen Garbage in Managed FaaS Workloads. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3627703.3629579

1 Introduction

Function-as-a-Service (FaaS) is a promising cloud service to realize *serverless computing*, where user-uploaded applications (or *functions*) are deployed, scaled, and managed by the cloud. Since functions are usually fine-grained code snippets, they are usually written in high-level languages like JavaScript, Python, and Java for ease of development [5], which mandate the use of the managed runtimes (like OpenJDK's HotSpot JVM and V8 engine).

Mainstream cloud vendors have proposed their own FaaS platforms [7, 28, 30, 38] to enable fine-grained and on-demand function execution. Upon receiving requests, FaaS platforms launch *instances* (usually containers or virtual machines) to execute individual functions. To mitigate the performance overhead of repetitively creating instances, mainstream FaaS platforms keep instances alive after a function has finished. According to prior studies [53], such instances can be kept from minutes to hours so that they can be reused when facing the same type of requests. Since those idle instances may still consume CPU resources without processing any requests (e.g., background tasks to check incoming network packets), mainstream FaaS platforms *freeze* them by avoiding scheduling their corresponding threads until the next request.

Although efficient for saving CPU cycles, this paper observes that the freeze semantics can cause problems for functions written in managed languages. Managed runtimes usually leverage their garbage collection (GC) mechanism for memory management, which reclaims dead objects (or garbage) when memory resources become scarce. However, if a FaaS instance is frozen, GC has no opportunities for memory reclamation, which leads to wasted memory. If the amount of wasted memory is large, the FaaS platform has to evict (usually destroy) idle instances to make space for new requests, which can lead to more instance creations and larger latency for function execution. This paper then presents the first characterization of *frozen garbage* by studying the memory behavior of various functions written in two popular languages (JavaScript and Java) in the FaaS scenario. We find that all functions introduce frozen garbage during their execution, which contributes to more than half of the memory consumption on average. Meanwhile, we analyze the memory management mechanisms inside mainstream language runtimes and uncover that blindly conducting GC is also not enough to resolve the frozen garbage problem.

To this end, this work proposes *Desiccant*, a freeze-aware memory manager to release memory resources consumed by frozen garbage and thus allow better memory efficiency for the FaaS platform. The design of *Desiccant* mainly contains three parts. First, it becomes activated only when the overall memory resources become scarce and provides an *activation threshold* that dynamically changes according to metrics like the frequency of instance eviction. Second, it interacts with instances and the FaaS platform to receive per-instance execution behaviors (namely *profiles*) to select the most cost-effective ones for memory reclamation. Lastly, it coordinates with existing GC algorithms in language runtimes to reclaim frozen garbage and releases free memory resources to the FaaS platform. Meanwhile, *Desiccant* also provides optimizations to mitigate performance degradation after GC and reduce memory overhead caused by shared libraries.

Desiccant is implemented on OpenWhisk [6], a popular open-source FaaS platform. It supports both JavaScript and Java and only requires minor modifications to the language runtimes. Our evaluation on various FaaS functions shows that *Desiccant* can efficiently reclaim memory resources consumed by frozen garbage and introduce up to 6.72× reduction in memory consumption. When evaluated under production traces [43], *Desiccant* can reduce the frequency of cold boots (instance creation before function execution) by up to 4.49× and improve the functions' p99 latency by up to 37.5%. We also evaluate *Desiccant* on Lambda, a commercial FaaS platform provided by AWS, and the results are similar.

To summarize, the contributions of *Desiccant* mainly include:

- The first observation and characterization that unveil the *frozen garbage* problem (§ 2) and its negative effects in the FaaS scenario (§ 3).
- *Desiccant*, a freeze-aware memory manager to reclaim frozen garbage in managed FaaS workloads and improve memory efficiency (§ 4).
- A comprehensive analysis with both various FaaS functions and production traces to show the performance benefits introduced by *Desiccant* (§ 5).

2 Frozen Garbage in Managed FaaS Workloads

2.1 The freeze semantics in FaaS

To achieve fine-grained resource management, FaaS platforms execute functions in separate *instances*. Currently, instances are usually containers or (lightweight) virtual machines. Since functions are short-running and triggered by external events (like upcoming requests), an instance becomes idle when the function exits. If FaaS platforms directly destroy those idle instances after functions exit, function execution would suffer from frequent instance creations (namely *cold boot* in the FaaS scenario) and larger latency. To this end, idle instances are usually kept alive by FaaS platforms, and the maximum duration can be as long as several hours [53]. Commercial FaaS platforms like AWS Lambda also allow users to explicitly configure the number of idle instances to improve performance [9]. Those idle instances are utilized to execute subsequent invocations in order to achieve lower latency. Although such a mechanism causes functions to not start from a clean state and may violate the "functional semantics" of FaaS, it is still widely employed for performance considerations. Restoring from function checkpoints or forking from existing instances can also be applied to achieve better performance. Although their resource consumption is lower, they might suffer from additional execution overhead. For example, according to our experiment, the recently released AWS SnapStart takes over 100ms to restore a snapshot to a Java function instance before execution.

However, those instances are not completely idle in the operating system's view. Although no functions are executed, some background threads in the FaaS instance can still be active. For example, the Just-In-Time (JIT) compiling threads in managed runtimes can continue running for code optimizations. As for application threads (usually referred to as *mutators*), they may periodically send heartbeats to other endpoints in the background. Those threads can silently steal CPU slices and affect the execution of other active instances.

To this end, FaaS platforms adopt the *freeze* semantics to manage idle instances. Taking OpenWhisk, a popular open-source FaaS platform, as an example: when an instance (container) finishes executing a function, OpenWhisk soon freezes the instance with the *pause* instruction in Docker so that all its threads are paused and cannot be executed. Upon receiving the same type of request, OpenWhisk scans the frozen instances and thaws the corresponding one (if any) to launch a function and process the request.

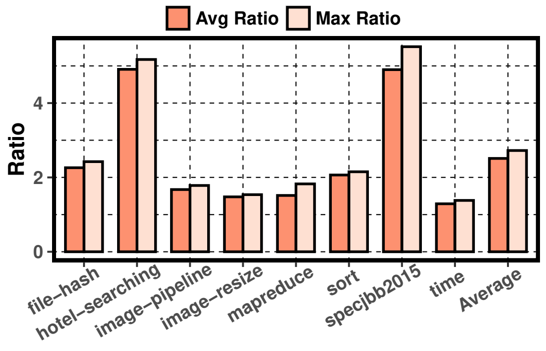
Note that the freeze semantics is not bound to OpenWhisk. We have investigated commercial FaaS platforms by splitting our uploaded functions into two parts: foreground and background. The foreground task is identified as the real function by the FaaS platform: when it finishes, the FaaS platform stops charging from the function. Meanwhile, the background task is implicitly specified, which periodically sends heartbeats marked with a unique function ID to a long-running server. In AWS Lambda, a mainstream FaaS platform, we find that the background task keeps sending heartbeats for about 100ms after the foreground one finishes. Afterward, the sending behavior stops, but it is resumed if a function of the same type is received by the instance (identified by the ID in heartbeats). The observation suggests that the instance is not destroyed by Lambda: it is only frozen to wait for new requests (consistent with Lambda's early report [1]). We also observe similar results in other FaaS platforms such as IBM Cloud Functions [30] and Alibaba Cloud Function Compute [18].

The freeze semantics make the execution of FaaS instances *intermittent*: the status of instances frequently switches between *frozen* and *running*. Although this pattern can save CPU resources, it is unaware of the memory efficiency issue, which is of great significance in data centers due to the high financial cost associated with memory [22].

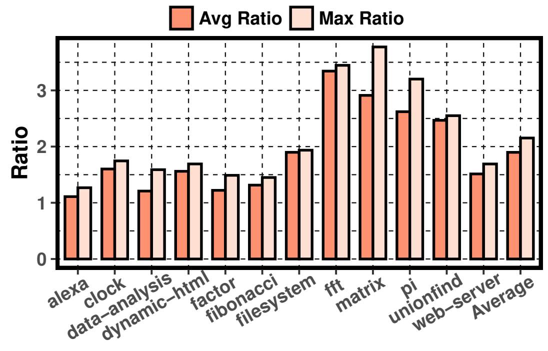
2.2 Frozen garbage

FaaS functions are usually written in high-level languages like JavaScript, Java, and Python to leverage their functionality and portability. Those languages use managed runtimes, which contain their own memory managers. Take Java as an example: users are required to allocate memory resources when creating the Java virtual machine (JVM) so that the JVM can manage user data as objects in a heap during application execution. When memory resources become scarce, JVMs initiate a garbage collection (GC) to reclaim memory for later reuse. Other managed runtimes, such as the V8 engine for JavaScript, also rely on GC for memory management.

Unfortunately, the GC phase is not aware of the freeze semantics in FaaS; it is still triggered only when free memory in a FaaS instance becomes scarce. Therefore, if an instance has finished executing a function and thus becomes frozen by the FaaS platform, it can contain dead objects which have not been collected by GC. Those dead objects cannot be removed unless the instance is resumed to execute more functions. We refer to those objects as *frozen garbage*. Since the FaaS platform cannot reuse the memory consumed by the frozen garbage, it has to destroy the whole instance for memory reclamation, which can lead to more instance creations and worse function execution latency. To our best knowledge, this work is the first one to unveil and analyze the frozen garbage problem.



(a) Java functions



(b) JavaScript functions

Figure 1. The ratios for frozen garbage

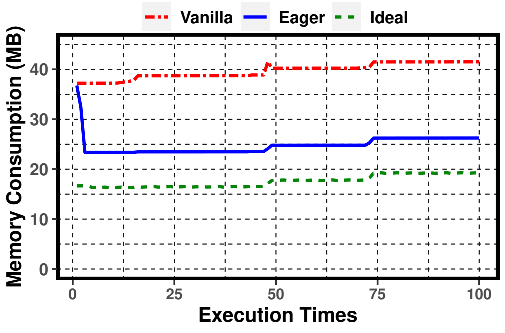
3 Characterization of Frozen Garbage

3.1 Memory inefficiency with frozen garbage

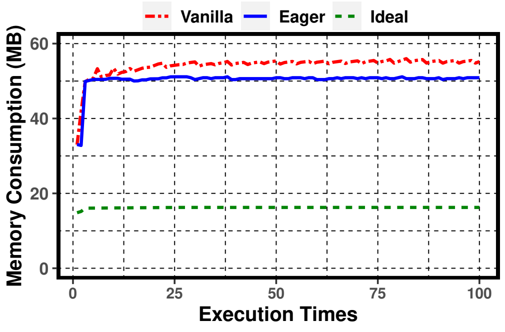
To study how frozen garbage affects the memory efficiency of managed FaaS workloads, we analyze the memory behaviors of various FaaS functions in two languages commonly used in FaaS development (JavaScript and Java), whose detailed descriptions are shown in Table 1. We collect all functions written in Java and JavaScript from multiple FaaS benchmarks [2–4, 11, 19, 37, 57] and also transform other event-based, highly-parallel applications (like microservices [24, 26]) into FaaS functions to further enrich the test suite. We leverage OpenWhisk [6], a commonly-used open-source FaaS platform, to execute functions, and the instances we use are Docker containers (the version is 20.10.9). To reflect the real-world execution on commercial FaaS platforms like Lambda, we apply Lambda's configurations for language runtimes to OpenWhisk. The version for Java is OpenJDK 8u322-GA (HotSpot JVM), which is the same as that in Lambda (Amazon Corretto [10], the JDK used in Lambda, also shows an OpenJDK version). As for JavaScript, the version for Node.js is 14.20.0 (one of the versions supported on Lambda) and the corresponding V8 version is 8.4.371.23-node.87. The memory budget for all FaaS instances is 256MB (the default value in OpenWhisk), and we leverage the runtime options in Lambda (e.g., the heap size) to configure language runtimes in OpenWhisk.

To calculate the amount of frozen garbage, we first iteratively execute functions 100 times in the same instance and collect the memory consumption every time a function exits. Since some applications are actually function chains that invoke multiple functions, we execute each function in separate containers and collect their accumulated memory consumption. We use USS (Unique Set Size, including *private_dirty* and *private_clean*) to measure the memory consumption of FaaS instances, which exclude shared libraries (e.g., *libjvm.so* for Java) since they are shared by multiple FaaS instances with the same language. Other metrics (e.g., the working set size) are not evaluated as they do not make much sense in the frozen scenario. To compare against the baseline, we also calculate an *ideal* case that only keeps useful memory contents (e.g., live objects) in the instance[1]. Figure 1 uses two ratios to represent the gap: *avg_ratio* and *max_ratio* standing for the average and maximum ratio between the memory consumption of the real execution and that of the ideal one. The results uncover that all functions regardless of programming languages generate frozen garbage. For some functions, the ratio can be quite large. For example, the maximum ratio for hotel-searching is larger than 5, which suggests more than 80% of its memory only contains frozen garbage. Meanwhile, the average number of maximum ratios for all Java functions is 2.72 (63.2% is occupied by frozen garbage). As for JavaScript functions, the number is 2.15 (53.5%). The results suggest that FaaS instances can contain a significant amount of frozen garbage. The frozen garbage consumes memory resources and reduces memory efficiency, thereby limiting the number of instances that can be cached within a fixed amount of memory in the FaaS framework, leading to more cold boots, lower peak throughput, and worse tail latency.

[1]The ideal memory consumption is measured at the end of each function execution because that is the point at which instances are frozen.



(a) file-hash (Java)



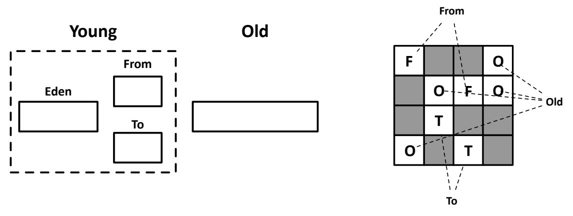
(b) fft (JavaScript)

Figure 2. The memory consumption curves for two representative functions

3.2 Is eager GC helpful?

Since the ideal case suggests the number of useful bytes in the heap is restricted, a straightforward method would be eagerly triggering GC after each function exits for memory reclamation. Unfortunately, this mechanism is actually not enough to reduce memory consumption, let alone its overhead induced by more collections. To show its effect, we choose two representative functions (respectively written in Java and JavaScript) and show their memory consumption curves while processing all 100 functions. Figure 2 shows that the eager GC mechanism does reduce the memory consumption of FaaS instances, but the reduction rate varies and fails to reach the ideal curve. For the *fft* function in JavaScript, eager GC only slightly reduces the memory consumption and is still far away from the ideal one. The reason eager GC is not enough varies in different language runtimes, so we discuss them separately below.

3.2.1 HotSpot (Java). Since the Lambda platform always uses the serial GC (found by dumping the runtime options inside Lambda's FaaS instances) for Java functions, we mainly study this algorithm. The serial GC embraces a generational heap design, which contains two generations: *young* and *old* (shown in Figure 3a). It also contains two GC



(a) HotSpot (Serial GC) **(b)** V8

Figure 3. The heap layout in different language virtual machines

cycles: *young GC* collects the young generation while *old GC* reclaims memory in both two generations. The young generation contains three spaces: the *eden space* serving memory allocation requests from mutators; the *from space* storing objects surviving at least one young GC; the *to space* serving as the destination for live objects in a young GC cycle. After each young GC cycle, the role of from space and to space switches. If an object has survived several GC cycles, it is *promoted* to the old generation.

The resizing phase in the HotSpot JVM is usually triggered after an old GC cycle, which contains resizing for both two generations. For the young generation, the size is mainly determined by the old generation size. As for the old, the size is mainly affected by its free ratio (the number of free bytes over the heap size), and the JVM ensures that the free ratio always stays within a predefined range.

Therefore, forced collections at the exit point of a function are quite helpful in controlling the heap size. First, applications can only use interfaces like *system.gc()*, which always leads to an old GC cycle and thus triggers resizing. Second, since all temporary objects become garbage after a function exits, the free ratio is usually large and the heap size can be shrunk. In the file-hash function, the whole heap size is fixed to 7.88MB thanks to the old GC after each function invocation, so the overall memory consumption also decreases compared with the vanilla baseline.

In HotSpot, heap expanding and shrinking are achieved via *mmap* since it can clear the physical pages mapped to the given virtual address range. The JVM maps more pages as usable to expand while marking pages as inaccessible (*PROT_NONE*) to shrink. Therefore, controlling the heap size does help reduce the memory consumption of a JVM (as shown in Figure 1a). However, all pages inside the heap are still managed by the JVM and do not return to OS even though they are actually free memory (e.g., pages in the to space when young GC is inactive). This design is reasonable because those pages are soon used by mutators and subsequent GC cycles, and reclaiming their physical memory would introduce more page faults and performance slowdown. However, a frozen instance cannot use those pages until it is resumed, rendering them wasted. Taking file-hash as an example, the live byte size after GC is actually 1.07MB, which means that 86.4% of heap memory contains only free pages and can be released.

3.2.2 V8 (JavaScript) .

The V8 engine also leverages a generational design but with several differences compared with the serial GC in HotSpot. For the young generation, V8 does not have an eden space; the from space serves allocation requests instead[2]. Meanwhile, all spaces are organized as discontinuous chunks (256KB), while OpenJDK's serial GC uses consecutive spaces (shown in Figure 3b). Note that V8's heap also contains other spaces (e.g., *code space* storing code generated by the JIT compiler), but their memory consumption is trivial and we omit them in this work.

Analog to HotSpot, V8 also has a heap resizing phase triggered after old GC cycles. Its policy for the old generation is quite simple: the generation expands when no free chunks are available and shrinks after GC generates free chunks. As for the young generation, the policies for expanding and shrinking are separately considered: shrinking is triggered after GC while expanding is before GC. If the accumulated live bytes found in GC since the last expansion exceeds the current young generation size, the generation is doubled. Meanwhile, if the allocation rate becomes lower than a given threshold, the generation size is reduced to twice the live byte size. This design is not friendly to the intermittent execution pattern in FaaS. Suppose an instance is frequently used only for a while, the allocation rate is high and the young generation size is repetitively doubled. Even though collections can be triggered during the execution, the young generation cannot be shrunk. This is the case in the fft function (Figure 2b): with more collections, the heap is still large due to frequent memory allocations. For the vanilla setting, the actual heap size is enlarged to 41.40MB mainly because of the expansion of the young generation (32MB, the upper bound for a 256MB heap). Due to the high allocation rate, the young generation size remains the same after eager GC, which can explain the curves in Figure 2b. However, if the function is no longer invoked, the free pages inside the heap are frozen.

V8 is more aggressive in releasing memory resources to OS. If shrinking is required, V8 also releases free pages in the to space since they are not used until the next GC cycle. However, since shrinking is not triggered when the allocation rate is high, it cannot help the corresponding instance to reduce its memory consumption.

3.2.3 Summary.

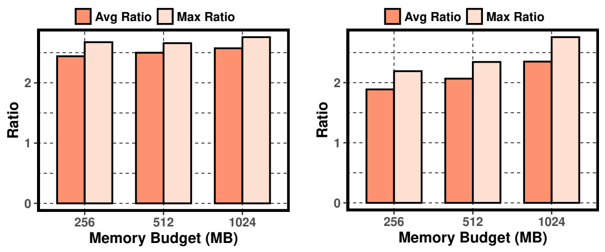
We have shown that eager GC is not effective in reducing memory consumption for FaaS workloads, but the reason is different for two language runtimes. For OpenJDK's HotSpot JVM, GC is only used for

[2]In V8, The role of from space and to space is somewhat different from HotSpot. To avoid misunderstanding, we always assume the from space stores live objects when GC is inactive.

resizing the heap instead of *releasing* memory resources, so the memory consumption is still high even when the heap mainly contains many free pages. For V8, although its resizing phase also releases memory to OS, its policy is not friendly to the intermittent execution pattern in FaaS and thus never shrinks before freezing. Therefore, one should provide runtime-specific designs to reclaim memory from instances using different languages. Meanwhile, the reclamation should also consider the performance overhead (like page faults) and remain adaptable to various GC algorithms and language runtimes.

3.3 The effect of heap size

Since the memory budget is directly related to the available CPU resources granted to a FaaS instance, a user may require more memory resources (resulting in a larger heap) for more CPU slices and better performance. Therefore, we also study the memory efficiency of FaaS instances when the heap size varies. Figure 4 shows the average number of average ratio and maximum ratio for all functions written in Java and JavaScript, respectively. For Java, the average number only slightly increases, which suggests that the HotSpot JVM is able to control the heap size regardless of the configuration. As for JavaScript, the ratios become larger as the heap size increases. For functions like fft, the average ratio increases from 3.27× in the default setting to 7.11× for the 1GB setting. Therefore, one can expect more frozen garbage with a larger heap setting for a part of FaaS applications.



(a) Java functions **(b)** JavaScript functions

Figure 4. The average number of ratios under different memory settings

4 Design of *Desiccant*

4.1 Overview

According to the analysis, this work provides *Desiccant*, a freeze-aware memory manager to reclaim idle memory resources in managed runtimes. During FaaS execution, *Desiccant* coordinates with language runtime instances to reclaim frozen garbage and release free memory to the operating system. Note that the process of *Desiccant* is part of the FaaS framework's resource management mechanism and therefore should not be billed to FaaS users. Meanwhile, *Desiccant* remains *non-intrusive* to most parts of the FaaS platform and language runtimes, which makes it adaptable to other platforms and programming languages. *Desiccant*'s design mainly contains the following three parts (introduced later).

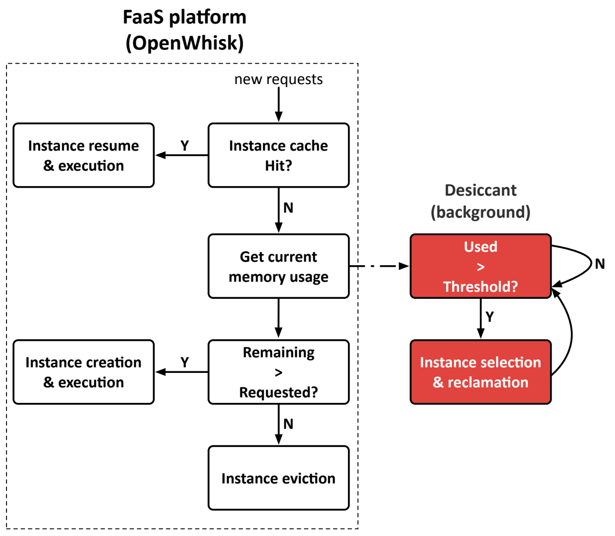


Figure 5. *Desiccant*'s Integration with OpenWhisk

- **Activation.** To reduce performance overhead, *Desiccant* is merely activated under memory pressure.
- **Instance selection.** According to Section 2.2, the memory behavior of functions varies, and *Desiccant* prefers to reclaim those containing the most frozen garbage.
- **Instance reclamation.** Since the original reclamation mechanisms are not effective, *Desiccant* modifies them with minor efforts to release memory.

4.2 Activation

To minimize CPU consumption, *Desiccant* is not activated until the portion of used memory of frozen instances exceeds a threshold. When *Desiccant* finishes reclamation and the amount of used memory has dropped below the threshold, it becomes inactive again. This design can be smoothly integrated with existing instance management mechanisms in FaaS platforms. For example, OpenWhisk monitors the accumulated memory consumption of all frozen instances and evicts (destroys) them when the remaining free memory is not enough to launch new instances. As shown in Figure 5, *Desiccant* can serve as a background sweeper for OpenWhisk. Since OpenWhisk already knows the current memory usage, *Desiccant* can directly leverage it to check if memory reclamation is necessary. Meanwhile, the eviction policy in OpenWhisk is orthogonal to *Desiccant*: when OpenWhisk determines to evict an instance, it does not need to consider if the instance is under memory reclamation. Due to the stateless nature of FaaS instances, direct evictions do not introduce correctness issues. This design simplifies the coordination between *Desiccant* and OpenWhisk and makes *Desiccant* less intrusive to FaaS platforms. Other policies (e.g., activating memory reclamation when idle computation resources are available) might further enhance the performance of *Desiccant*, and we left the design of more advanced policies as our future work.

4.3 Instance selection

When *Desiccant* is activated, it first needs to select frozen instances for reclamation. *Desiccant* follows two principles for instance selection. First, *Desiccant* prefers instances that have become frozen for long, as they continuously waste memory resources. Second, *Desiccant* should choose instances with the best reclamation efficiency (i.e., releasing the most memory given the same amount of CPU time slices). For the first principle, *Desiccant* provides a *timeout* value, and only instances whose freeze time exceeds the timeout are considered. As for the second one, *Desiccant* relies on both language runtime instances and the FaaS platform to collect memory and CPU-related information (namely *profiles*) after each reclamation succeeds. With those profiles, *Desiccant* can calculate the expected reclamation throughput (the amount of reclaimed memory in a fixed time interval) for all candidates and select those with the largest throughput for reclamation (details in Section 4.5).

4.4 GC-based instance reclamation

Figure 6 shows the workflow of *Desiccant*'s reclamation phase. After selecting suitable instances, *Desiccant* first informs the FaaS platform of the identifier of instances to be reclaimed. The FaaS platform then interacts with the instance by invoking the *reclaim* interface. Since FaaS platforms have already established a connection with instances to send user requests, this step only needs to add a new *reclaim* API which should be implemented by different language runtimes.

HotSpot. Since OpenJDK's original GC interface is in the System module (*System.gc()*), we add a *reclaim* interface in the same module. The pseudocode of the reclamation is shown in Algorithm 1. When *reclaim* is invoked, it first uses the GC interface to reclaim frozen garbage in the whole heap (line 1-5). Afterward, it relies on the JVM's resizing policy to shrink its heap size if possible (line 6-9). Lastly, we modify HotSpot's releasing policy so that all free memory pages are returned to OS. This includes the whole from space (not used until the next GC) and free memory in the eden space, to space, and the old generation (line 10-15).

V8. The reclamation phase in V8 is similar. Since it already provides *global.gc* interface, we add *global.reclaim* to first conduct GC and rely on the resizing policy to shrink. If shrinking is required, as V8 automatically releases memory

Algorithm 1 reclamation in HotSpot

```
1:  # for all generations in the heap
2:  for $gen \leftarrow generations$ do
3:      # perform GC on the generation
4:      $gen.do\_collection()$
5:  end for
6:  # resize heap
7:  for $gen \leftarrow generations$ do
8:      $gen.resize()$
9:  end for
10: # reclaim memory from each space
11: for $gen \leftarrow generations$ do
12:     for $space \leftarrow gen.spaces()$ do
13:         $mmap(space.top(), space.end() - space.top(), ....)$
14:     end for
15: end for
```

resources in the from space and the old generation, we only need to further release free memory in the to space. Otherwise, the reclaim interface releases free pages in all spaces. Note that chunks in V8 contain self-described metadata on their first page (4KB), which cannot be released. Nevertheless, unmapping other pages in the chunk already releases most memory resources (98.4%).

When reclamation is finished, the language runtime collects its own memory information and sends the memory profile to the FaaS platform. The FaaS platform further extends the profile by calculating CPU time consumed by the reclamation phase and sends the combined one to *Desiccant* for more accurate instance selection. Since *Desiccant* reuses the GC algorithms inside language runtimes, the required modifications are minor. We only require 69 LoCs modifications to HotSpot and 170 LoCs to V8, which makes *Desiccant* easily adapted to other GC algorithms and language runtimes.

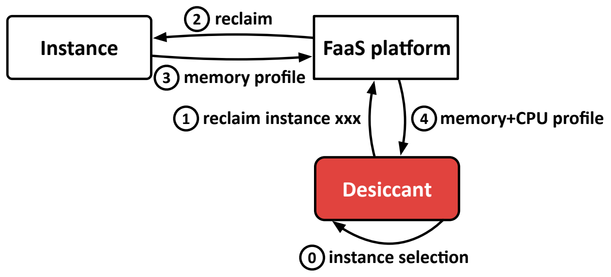


Figure 6. The reclamation workflow

4.5 Policies used in *Desiccant*

4.5.1 Dynamic activation threshold. Since FaaS functions have various memory demands and execution behaviors, the activation threshold is hard to be statically determined. If the threshold is set too small, *Desiccant* is frequently triggered and consumes considerable CPU slices. On the contrary, a too-large threshold would make FaaS platforms suffer from insufficient memory and frequent instance evictions. To this end, *Desiccant* proposes to dynamically change the threshold by monitoring instance evictions. If the FaaS platform starts to evict instances, *Desiccant* immediately lowers the threshold to a predefined one (60% by default) and thus releases more memory resources. Otherwise, *Desiccant* gradually increases the threshold to reduce its performance overhead.

4.5.2 Profile collection and selection policy. The last remaining problem is how to design policies so that *Desiccant* can accurately select instances with the best reclamation throughput. *Desiccant*'s policy mainly relies on two observations. First, due to the stateless nature of FaaS, the number of live bytes in a heap remains quite stable when each function exits (Figure 2 shows similar results). Second, since mainstream collectors use tracing-based algorithms to find all live objects and reclaim others, their cost is proportional to the number of live objects. Those two observations suggest both the in-heap live byte size and reclamation time is stable if the reclamation phase is triggered after the exit point of functions. According to the observations, *Desiccant* provides the estimation algorithm below.

In-heap live bytes. For ease of GC analysis, language runtimes provide interfaces to query the number of live bytes in the heap. Therefore, the *reclaim* interface can invoke them to collect the number and send it to the FaaS platform as the memory profile. The profile is then sent to *Desiccant*, and an instance's expected in-heap live bytes are estimated as the average of all profiles related to it.

CPU time. *Desiccant* relies on FaaS platforms to collect the CPU time, which is calculated with the elapsed time and the number of CPUs used by reclamation. To avoid interferences with active FaaS instances, the FaaS platform only uses idle CPU resources for reclamation and decreases the number of CPU slices when newly-coming functions require execution. Therefore, FaaS platforms should memorize the number of CPUs used during the whole reclamation phase and leverage it to calculate the accumulated CPU time. For example, suppose the reclamation takes 10ms to finish (collected by the platform), and its corresponding control group (cgroup) has 0.5 CPUs in the first 3ms and 0.25 in the rest, then its accumulated CPU time is 3.25ms (0.5*3+0.25*7). The accumulated time is then written to the CPU profile so that *Desiccant* can calculate the average time for each instance.

Estimated reclamation throughput. To estimate the expected reclamation throughput, *Desiccant* needs to know each instance's current in-heap memory consumption. For V8, this implementation is quite simple as it has maintained variables to memorize consumed memory resources. As for HotSpot, we require each FaaS instance to report its heap's address range after creation. Since a JVM's address range for its heap never changes, the FaaS platform can use *pmap* to collect its memory consumption within the range. With the in-heap memory consumption statistics, *Desiccant* can estimate the throughput with the formula:

$$Throughput_{Estimated} = \frac{(Mem_{heap} - Estimated\_live\_bytes)}{Estimated\_CPU\_time}$$

Afterward, *Desiccant* sorts instances with the estimated throughput and selects the highest ones to reclaim.

Handling new instances. Since *Desiccant* maintains profiles for each instance, a challenge is how to predict the memory behavior for newly-created instances. As instances running the same function share similar memory behaviors (especially the live bytes), *Desiccant* first searches for existing instances with the same function type and leverages their profiles for estimation. If no instances can be found, *Desiccant* adopts the average throughput of all pre-calculated instances for estimation. Once the instance is reclaimed, *Desiccant* can collect its profile and subsequent estimations become more accurate. When an instance is destroyed by the FaaS platform, its profiles are also abandoned to reduce the memory overhead.

4.6 Optimizations for shared libraries

Language runtimes require memory-consuming shared libraries (e.g., *libjvm.so* in HotSpot and *node* in V8) for execution. Since FaaS frameworks usually provide the same language runtime for functions in the same language, those libraries can be shared among containers and their memory consumption is amortized. However, if the libraries are used by only one function, the memory overhead is still considerable. To this end, *Desiccant* provides an optimization to unmap libraries if they are only used by one frozen instance. This is achieved by searching the per-process *smaps* file for memory ranges that are (1) private to the current process, (2) not modified, and (3) mapped from files. Afterward, *Desiccant* marks those memory ranges as non-present via *mmap* system calls. Note that those libraries are also included in the USS part if they are not shared among instances, so *Desiccant* does not need to modify its policy in Section 4.5 to support this optimization.

4.7 Avoiding aggressive collections

The GC interface provided by V8 suggests a global and thorough collection, which aggressively reclaims all collectible objects (including those pointed by weak references) and causes significant performance degradation due to JIT code deoptimizations for subsequent function executions. Since *Desiccant* reclaims memory resources only in a best-effort fashion, it does not require such aggressive collections. To this end, *Desiccant* slightly modifies the GC interface of V8 so that it can receive an option to avoid collecting objects referred to by weak references. This modification only adds 7 LoCs to V8 but improves function performance after reclamation.

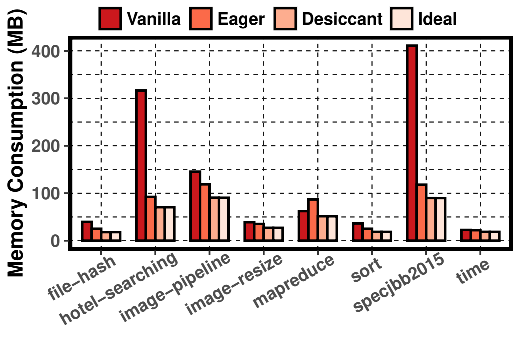
5 Evaluation

5.1 Experiment setup

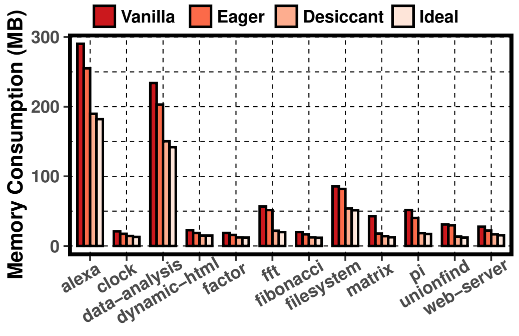
Desiccant is implemented and integrated with OpenWhisk, a commonly-used open-source FaaS platform (the version is 20.11). All FaaS functions used in the evaluation are listed in Table 1. We mainly use a server with dual Intel Xeon Gold 6138 CPUs and 128GB DRAM to launch containers as FaaS instances. Meanwhile, we compare *Desiccant* with two baselines: a *vanilla* one that executes FaaS functions without considering the freeze semantics and an *eager* one that triggers GC after every function exits.

5.2 Single function evaluation

We first study how *Desiccant* helps improve the memory efficiency for single functions. Each function instance has 256MB memory resources and 0.14 CPU (according to the configuration in commercial FaaS platforms [8, 47]). For all functions, we still execute them for 100 times in the same instance and freeze them. We assume the memory resources become scarce so that *Desiccant* is activated to resume frozen garbage from frozen instances. The memory consumption is compared with the aforementioned baselines and the results are shown in Figure 7. Compared with the vanilla baseline, *Desiccant* can significantly reduce the memory consumption of FaaS instances: the reduction after 100 iterations ranging from 1.21× to 4.57× for Java functions (2.78× on average) and from 1.51× to 3.04× for JavaScript functions (1.93× on average). When compared with the eager GC mechanism, *Desiccant* still reduces the memory consumption for all functions (1.36× on average for Java and 1.55× for JavaScript). Alternative solutions include not freezing functions after execution or relying on the swapping mechanism of the operating system to reclaim memory. The non-freezing solution yields similar results to the vanilla baseline since frequent function execution also hinders potential background garbage collection operations. The swapping solution suffers from higher execution overhead after reclamation as demonstrated in Section 5.6. The results of *Desiccant* are close to the ideal baseline (0.1% on average for Java functions and 6.4% on average for JavaScript functions) in Section 3.1 and indicate *Desiccant* is effective in releasing free memory consumed by frozen garbage. The reduction varies among functions due to different memory behaviors. For JavaScript functions, the improvement is mainly related to the allocation rate: functions with more active allocation behaviors frequently double their young generations, which cannot be optimized by the eager GC mechanism. As for Java functions, they usually have large demands for memory during the first execution (due to initialization), which significantly enlarges the heap size. Since the frequency of old GC is quite small, the heap does not shrink until all functions exit. Meanwhile, both eager GC and *Desiccant* are helpful in triggering the shrinking phase and thus reduce the memory consumption, while *Desiccant* further releases free memory pages inside the heap to OS. The difference between *Desiccant* and the ideal case stems from different reasons. For Java functions, it mainly arises from the page alignment overhead during reclamation. For JavaScript functions, it mainly results from the unreclaimed fragmented free memory released by the mark-sweep algorithm and can be eliminated by further integrating reclamation with the V8 engine (e.g., reclaiming them with the help of V8's free list).



(a) Java functions



(b) JavaScript functions

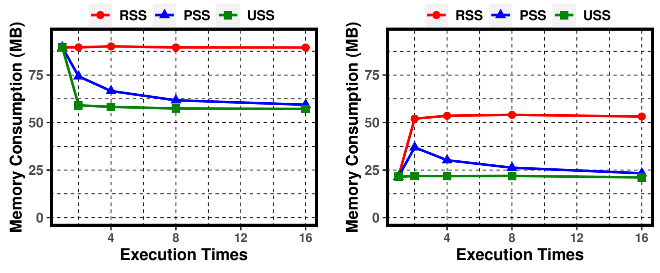
Figure 7. Single instance's memory consumption after repetitive function executions

Table 1. Evaluated FaaS functions in this work. For chained functions, the number of functions are denoted after their names

Language	Function name	Description
Java	time	Returning current time
	sort	Sorting an array of integers
	file-hash	Calculating the hash value for a file
	image-resize	Resizing an image
	image-pipeline (4)	Processing an image with four consecutive functions
	hotel-searching (3)	Searching hotels with preferences
	mapreduce (2)	Counting words in a map-reduce fashion
	specjbb2015 (3)	The purchasing transaction in a simulated supermarket
JavaScript	clock	Returning the executed time of current process
	dynamic-html	Generating a HTML file randomly
	factor	Calculating the factorization for a large integer
	fft	Fast Fourier transform
	fibonacci	Calculating the nth value in a Fibonacci sequence
	filesystem	Accessing the file system
	matrix	Matrix multiplication
	pi	Calculating pi with a given number of iterations
	unionfind	Executing operations over a union-find disjoint set
	web-server	Launching a web server and processing requests
	data-analysis (6)	Analyzing data in a database
	alexa (8)	Interacting with smart-home devices

The results for mapreduce are interesting because the eager baseline has an even larger memory consumption compared with the vanilla one. This is because the eager GC mechanism is unaware of the chain semantics in FaaS workloads. Since the mapper function needs to transform intermediate data into the reducer one, the data cannot be reclaimed by GC even after the function exits, which leads to higher memory consumption. In contrast, *Desiccant* does not have this problem because it only reclaims inactive (frozen) instances whose intermediate data has been transferred and thus can be collected.

Effects on other metrics. We also measure the overall memory consumption with other metrics, RSS (Resident Set Size) and PSS (Proportional Set Size). The evaluation is conducted by launching multiple instances for the same function (using *fft* as an example, other functions have similar results). As shown in Figure 8, when the number of containers is one, both RSS and PSS are improved by 4.16× thanks to the *Desiccant*'s in-heap reclamation and unmap optimization. When the number of concurrent instances increases, the RSS value for each instance remains the same while the PSS gradually approaches USS as libraries are shared among instances. The results suggest *Desiccant* can improve the system's overall memory consumption for various scenarios.



(a) Vanilla **(b)** Desiccant

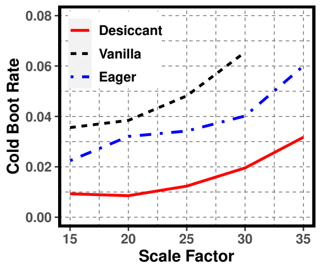
Figure 8. Improvement for per-instance RSS and PSS

5.3 Performance on production traces

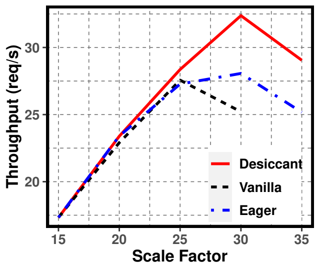
We also leverage production traces collected from Azure Functions [43] for evaluation. The traces contain thousands of functions (each differentiated by a unique ID) deployed on Azure, along with their inter-arrival time, duration, and the allocated virtual memory. However, the traces do not include the actual function code, making it impractical to directly replay the trace for evaluating Desiccant's performance. Therefore, we choose 20 functions from the trace whose execution time is closest to those listed in Table 1. For chained functions, we also select one function from the trace whose execution time is close to the overall time for the whole chain. Later, we invoke the functions in Table 1 based on the inter-arrival patterns of the selected functions recorded in the trace. Each function instance still has 256MB memory for execution, and we configure the instance cache (the cache used to keep frozen instances alive in memory) size to 2GB. To show the results under different workloads, we use a *scale factor* to proportionally reduce the inter-arrival time for all functions. For example, if the scale factor is 10, the inter-arrival time for functions is ten times smaller than that in the original traces. Before replaying the trace, we first warm up the system with a fixed scale factor (15) for 60 seconds, and the replay time is fixed to 180 seconds.

Figure 9 first shows the performance of *Desiccant* under different scale factors when compared with the vanilla and eager configuration. As the results in Figure 9a suggest, *Desiccant* can reduce the cold boot rate by up to 4.49× compared to the vanilla baseline (up to 3.75× compared to the eager baseline), thanks to its memory reclamation mechanism on frozen garbage. A lower cold boot rate also contributes to better throughput: Figure 9b shows that the average throughput on Azure traces can reach 32.37 requests per second with *Desiccant*, which is 17.4% and 15.3% better than that in the vanilla and eager setting, respectively. Meanwhile, the lower cold boot rate induced by *Desiccant* can result in lower execution time, which is also helpful in reducing the financial cost of FaaS functions borne by FaaS users. Besides, by reducing the cold boot rate, *Desiccant* also has a lower CPU utilization as a cold boot introduces heavy CPU overhead (collected by the *sar* tool, shown in Figure 9c). When the scale factor is between 15 and 30, *Desiccant* reduces FaaS instances' accumulated CPU utilization (averaged in 180 seconds) by 6.9% (vanilla) and 9.2% (eager) on average. The eager baseline occupies more CPU when the scale factor is low due to frequent GCs, suggesting the importance of reclaiming only when necessary, as is done by *Desiccant*. As for reclamation, we find that the frozen garbage reclamation phase introduced by *Desiccant* only induces up to 6.2% CPU overhead, which is trivial in the overall CPU utilization. Triggering GC only before freezing a function can reduce the CPU overhead in the eager baseline, but it will still suffer from a larger cold boot rate under high throughput when compared to *Desiccant* due to the worse reclamation efficiency.

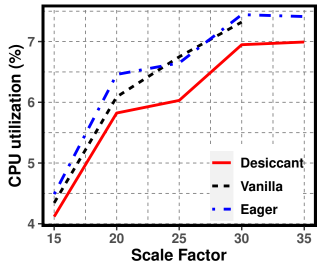
Figure 10 further illustrates the tail latency of function execution with two different scale factors. Since *Desiccant* is helpful in reducing the cold boot rate, it reaches better tail latency when compared with the vanilla one and the eager one regardless of scale factors. When under a medium scale factor (15), *Desiccant* achieves 33.1%, 9.8%, and 37.5% improvement on p90, p95, and p99 latency, respectively compared with the vanilla baseline (33.0%, 7.5%, and 27.9% compared with the eager baseline). As for higher throughput (the scale factor is 25), the results are 39.9%, 36.3%, and 0.1% compared to the vanilla baseline and 36.4%, 16.9%, and 0.1% compared to the eager baseline. The tail latency gap generally increases with a larger scale factor because the baselines' frequency of cold boot increases. Nevertheless, *Desiccant* does not show a significant improvement in p99 latency when the scale factor is 25. This is because the throughput is quite large and available CPU resources can be exhausted, which makes new requests wait for seconds and dominates the p99 latency.



(a) Cold boot rate

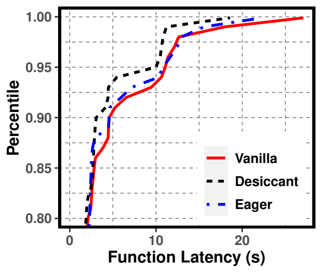


(b) Throughput

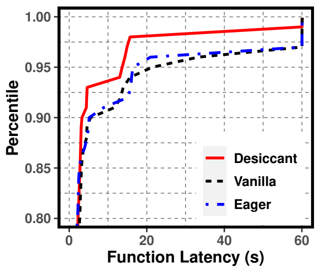


(c) CPU utilization

Figure 9. Performance on Azure traces



(a) scale factor=15 (medium)

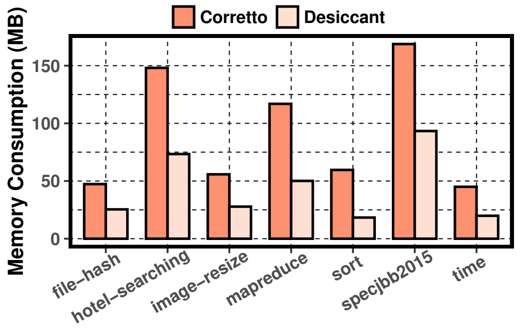


(b) scale factor=25 (high)

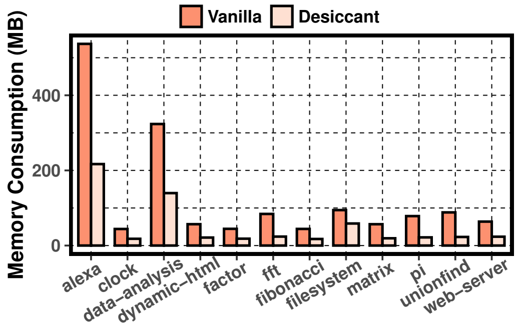
Figure 10. Tail latency for different scale factors

5.4 Results on Lambda

To confirm *Desiccant*'s performance is not bound to the OpenWhisk platform, we further evaluate functions on AWS Lambda. Since we have no permission to modify the Lambda platform, we use the following strategy to evaluate *Desiccant*. First, we pack functions into container images for deployment, which contain the function's related files, modified language runtimes and a web server for communication. Second, we still assume that the functions are iteratively executed for 100 times, so we manually send a special invocation to trigger a reclamation when all execution has finished. Note that since the image-pipeline function contains external calls to other processes, which are not supported by the vanilla Corretto image, we provide the results for the other six Java functions.



(a) Java functions



(b) JavaScript functions

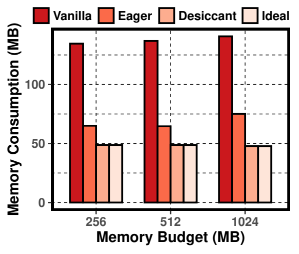
Figure 11. Memory efficiency of functions on AWS Lambda

As shown in Figure 11, since the images are different from those running in OpenWhisk, the memory behavior of *Desiccant* also changes. Nevertheless, *Desiccant* still introduces 2.08× improvement on average for Java functions (2.76× for JavaScript functions). Since Lambda does not allow sharing libraries among multiple FaaS instances, the unmapping optimization becomes more effective. The results suggest that the frozen garbage problem also exists on Lambda and *Desiccant* is still effective in reclaiming frozen garbage on a commercial FaaS platform.

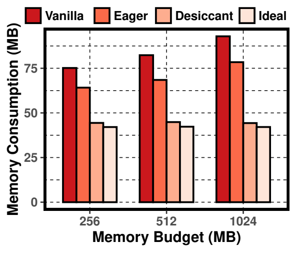
Discussion on GC algorithms. We have observed that Lambda leverages serial GC for all functions regardless of their configurations, so we mainly study this algorithm in this work. However, serial GC is a simple algorithm that only leverages one GC thread for collection. Since the number of CPUs granted for each function instance is known before execution, we suggest FaaS platforms provide an adaptive configuration where parallel collection algorithms can be used for instances with abundant CPU resources. Meanwhile, since mainstream GC algorithms are tracing-based, *Desiccant* can be easily integrated with them by reusing their collection phases.

5.5 Results on different memory settings

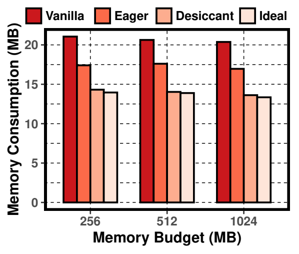
Figure 12 shows instances' memory consumption after 100 function executions when the memory budget varies. We first average the memory consumption among Java (Figure 12a) and JavaScript (Figure 12b) functions. Similar to the analysis results in Figure 4, the reduction in average memory consumption against the vanilla baseline does not change much with a larger heap setting (from 2.75× to 2.94×). However, the reduction is larger in JavaScript functions (from 1.69× to 2.10×). Although most functions written in JavaScript have a stable memory consumption regardless of the configured heap size (such as clock shown in Figure 12c), exceptions like fft show dramatically larger memory consumption for both the vanilla and the eager baseline (Figure 12d). We have analyzed the heap size of fft and observed that its young generation size continuously increases to 128MB for the 1GB setting and never shrinks. Since *Desiccant*'s memory consumption always remains stable thanks to its reclamation mechanism, its improvement against the vanilla baseline can reach 6.72× for fft's 1GB configuration (5.50× for the eager baseline).



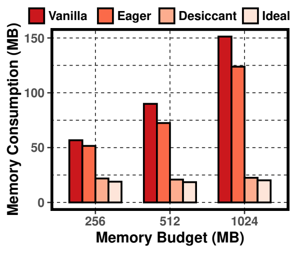
(a) average (Java)



(b) average (JavaScript)



(c) clock



(d) fft

Figure 12. Memory consumption under different memory settings

5.6 Execution overhead after reclamation

Since *Desiccant* releases memory to the operating system, a function's subsequent executions may suffer from more page faults and performance slowdown. To understand the performance overhead, we assume each function is iteratively executed for 130 times, reclaimed, and resumed to execute for 10 times (we use more execution times than other experiments because function performance is heavily interfered with JIT compilation when executing for 100 times). The average latency after reclamation is then compared with the latency averaged over the last ten executions before reclamation. As the results shown in Figure 13 suggest, the execution overhead introduced by *Desiccant* is 8.3% on average. The overhead is acceptable considering *Desiccant* is able to reduce the cold boot rate, which has much larger effects on the execution latency. Reclaiming memory with the swapping mechanism comes with a higher execution overhead after reclamation (2.37 times slower for the *sort* function when reclaiming the same amount of memory as *Desiccant*), as it lacks runtime semantic guidance and may swap out pages used later. Meanwhile, avoiding aggressive collections (proposed in Section 4.7 can prevent two JavaScript functions (*data-analysis* and *unionfind*) from significant performance slowdown (2.14× and 1.74×, respectively).

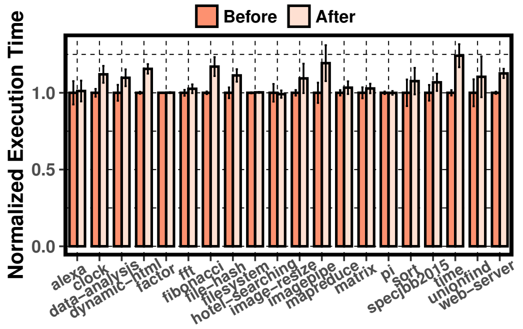


Figure 13. The execution overhead introduced by *Desiccant*

6 Related Work

6.1 Instance management in FaaS

Cold boot is a notorious problem in FaaS as it introduces large tail latency to function execution. To avoid cold boot, a straightforward method is to cache used instances and reuse them when receiving the same type of function. This method has been used by both commercial and open-source FaaS frameworks [46, 53]. However, cached instances also introduce large memory consumption. To mitigate this problem, prior work provides various policies to reduce the cost of cached instances. Shahrad et al. [43] leverage a hybrid histogram policy to identify applications invocation patterns so that an instance can be reloaded and pre-warmed right before a function needs it. FaaSCache [23] borrows ideas from traditional caching policies to maintain idle containers, while IceBreaker [42] allows keeping warm containers live on low-end servers to reduce the cost. Their warm-up policies are orthogonal to *Desiccant*, and *Desiccant*'s memory reclamation policy can further improve the memory efficiency in their systems.

Instead of pre-warming, another line of work chooses to create general-purpose instances (or *snapshots*) which can be used by different functions. OpenWhisk launches pre-warm containers with pre-installed language runtimes like Node.js [46]. SOCK [41] optimizes python functions by launching instances with pre-installed popular python packages. Catalyzer [20] and SEUSS [16] create instances by forking from existing ones. ReplayableJVM [52] launches Java functions with existing JVM snapshots. Pagurus [32] repurposes existing instances so that they can be reused by other functions. REAP [48] loads snapshots from disks and prefetches frequently-accessed pages according to prior knowledge of functions' memory access patterns. *Desiccant* mainly focuses on memory reclamation from existing FaaS instances and does not require modifications to FaaS instances like containers.

6.2 Language runtime optimizations for FaaS

Since managed languages are widely used in the FaaS scenario, their underlying managed runtimes are optimized or retrofitted for fine-grained function execution. Photons [21] and Flock [60] allow multiple instances to run together in the same JVM to reduce the frequency of cold boot. Shredder [58] and CloudFlare Workers [12] support running multiple JavaScript functions in the same V8 engine with the *context* abstraction. Faasm [44] provides *Faaslet* to execute functions compiled into WebAssembly [29] and stateful support to efficiently share data among functions. JWarmup [59] harmonizes features in JVMs to improve the startup time, while GraalVM Native Image [54] compiles Java applications and the underlying JVM into executables for fast boot. BeeHive [61] provides shared memory support to automatically offload functions to FaaS platforms for resource elasticity. JITServer [31] turns the dynamic compilation module into an external service to reduce the application startup time, while Ignite [17] allows sharing JITted code among Java functions to skip the warmup phase after instance creation. *Desiccant* optimizes language runtimes to release unused memory to OS, but its modifications are minor to remain adaptable to various GC algorithms and managed runtimes.

6.3 Garbage collection optimizations

Garbage collection (GC) is an important module in managed runtimes, and its performance is critical to applications. To this end, prior work has proposed GC optimizations for various scenarios. Yak [39, 40] and Broom [27] are optimized for big-data scenarios and leverage epoch-based mechanisms to manage objects with the same lifecycle together. NG2C [13–15] has a similar goal but uses pre-tenuring to avoid frequent data copy on long-lived objects. Yang et al. [56] propose isolating read and write operations in GC considering the bandwidth issues on non-volatile memory devices. Wang et al. [34, 49, 50] uncover the GC bottleneck on a far memory setting and provides several optimizations. ElasticMem [51] dynamically changes the heap size limit to manage memory resources among multiple JVMs in a cloud environment. Taurus [35, 36] coordinates the GC behaviors among multiple JVMs for better application performance. M3 [33] instead coordinates GC with memory management mechanisms of other modules in the software stack for performance improvement. Platinum [55] optimizes GC in interactive services by isolating the execution of mutators and GC threads. Suo et al. [45] and NumaGiC [25] uncover GC performance issues related to NUMA and provide NUMA-friendly GC optimizations. *Desiccant* does not provide optimizations on GC algorithms, but it manually chooses triggering points for GC (after functions exit) and leverages GC to reclaim and release frozen garbage for better memory efficiency.

7 Discussion: Applying *Desiccant* to other scenarios

Although *Desiccant* currently mainly focuses on two languages (Java and JavaScript) and the GC algorithms used in AWS Lambda, Other scenarios share the similar problem. Take the Python language Take the Python language, which is commonly used in the FaaS scenario, as an example. The mainstream CPython runtime manages memory in arenas of 256KB and only releases the entire memory of an arena when it becomes empty. Since CPython is not aware of freeze semantics, the memory in arenas is not returned to the OS when the instance should be frozen. Other languages (e.g., Go) and GC algorithms (e.g., G1GC) also face the same issues. In summary, the frozen garbage problem commonly exists in language runtimes and GC algorithms whose memory management mechanism does not promptly return the memory to the OS when it becomes available.

To apply *Desiccant* and address the frozen garbage problem, a language runtime or GC algorithm must have a memory management mechanism that satisfies the following requirements: 1) It should have the ability to estimate the reclamation throughput. 2) It should be able to determine which region of the memory is free (not occupied by live objects). Consequently, *Desiccant* can use the estimated reclamation throughput to select appropriate instances for reclamation and return the free memory to the OS with the assistance of the language runtime. Therefore, except for the HotSpot JVM and the V8 engine, *Desiccant* can also be applied to other scenarios. For instance, for the CPython runtime, *Desiccant* can record the average collection time and the number of live objects after each garbage collection, and can collect the addresses of all arenas and determine their physical memory usage with the pmap tool to calculate the total physical memory occupied by the heap. Later, *Desiccant* can utilize the formula described in Section 4.5 to calculate the estimated reclamation throughput, thereby selecting the most valuable instance for reclamation. To reclaim memory, *Desiccant* can leverage CPython's mark-sweep garbage collector and internal data structures (e.g., free list) to identify free memory regions and release them back to the operating system using the mmap operation. For the Go runtime, as its heap is located in several contiguous memory ranges, *Desiccant* can employ similar methods to estimate the efficiency of reclamation. Subsequently, *Desiccant* can utilize Go's internal data structures to identify free regions and perform reclamation accordingly for the selected instance. For the G1GC, despite having a different GC algorithm compared to the Serial GC, it is still based on the HotSpot JVM and fulfills the aforementioned requirements, making it compatible with *Desiccant*. we intend to extend the application of *Desiccant* to more scenarios in the future.

8 Conclusion

FaaS (function-as-a-service) is becoming a promising computing paradigm in the cloud environment. Since most functions are written in high-level managed languages like JavaScript and Java for ease of development, their memory resources are managed by the underlying language runtimes. Unfortunately, the runtimes are unaware of the freeze semantics in FaaS and thus contribute to the frozen garbage problem and lead to memory waste. To this end, this work proposes *Desiccant*, which reclaims free memory from frozen FaaS instances to improve memory efficiency and reduce the frequency of instance re-creation. The evaluation results on various FaaS workloads show that *Desiccant* can reduce functions' memory consumption by up to 6.72×.

Acknowledgments

We sincerely thank the anonymous EuroSys'24 reviewers for their insightful suggestions. This work was supported in part by the National Natural Science Foundation of China (No. 62202295, 61925206, 62172272). Corresponding author: Mingyu Wu (mingyuwu@sjtu.edu.cn).

References

[1] Understanding container reuse in aws lambda. https://aws.amazon.com/cn/blogs/compute/container-reuse-in-lambda/, 2014.
[2] Faastest. https://github.com/nuweba/faasbenchmark, 2020.
[3] Serverlessbench. https://serverlessbench.systems/en-us/, 2020.
[4] The faasdom benchmark suite. https://github.com/faas-benchmarking/faasdom, 2021.
[5] The state of serverless. https://www.datadoghq.com/state-of-serverless/, 2022.
[6] Apache OpenWhisk. Apache openwhisk - open source serverless cloud platform. https://openwhisk.apache.org/, 2020.

[7] AWS. Aws lambda. https://aws.amazon.com/lambda/, 2023.

[8] AWS. Configuring lambda function options. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html, 2023.

[9] AWS. Configuring provisioned concurrency. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html, 2023.

[10] AWS. Configuring provisioned concurrency. https://aws.amazon.com/corretto/, 2023.

[11] Timon Back and Vasilios Andrikopoulos. Using a microbenchmark to compare function as a service solutions. In *ESOCC*, volume 11116 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2018.

[12] Zack Bloom. Cloud computing without containers. https://blog.cloudflare.com/cloud-computing-without-containers/, 2018.

[13] Rodrigo Bruno and Paulo Ferreira. Polm2: automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 147–160. ACM, 2017.

[14] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. Ng2c: pretenuring garbage collection with dynamic generations for hotspot big data applications. *ACM SIGPLAN Notices*, 52(9):2–13, 2017.

[15] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 28. ACM, 2019.

[16] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[17] João Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From warm to hot starts: leveraging runtimes for the serverless era. In *HotOS*, pages 58–64. ACM, 2021.

[18] Alibaba Cloud. Function compute. https://www.alibabacloud.com/product/function-compute, 2023.

[19] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: a serverless benchmark suite for function-as-a-service computing. In *Middleware*, pages 64–78. ACM, 2021.

[20] Dong Du, Tianyi Yu, Yubin Xia, 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*. ACM, 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] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting vms in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 583–594, 2022.

[23] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS*, pages 386–400. ACM, 2021.

[24] 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. ACM, 2019.

[25] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. Numagic: a garbage collector for big data on big numa machines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 661–673. ACM, 2015.

[26] GitHub - delimitrou/DeathStarBench. Open-source benchmark suite for cloud microservices. https://github.com/delimitrou/DeathStarBench.

[27] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[28] Google. Cloud functions - google cloud. https://cloud.google.com/functions/, 2022.

[29] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[30] IBM. Ibm cloud functions. https://www.ibm.com/cloud/functions, 2023.

[31] Alexy Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. Jitserver: Disaggregated caching JIT compiler for the JVM in the cloud. In *USENIX Annual Technical Conference*, pages 869–884. USENIX Association, 2022.

[32] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In *USENIX Annual Technical Conference*, pages 69–84. USENIX Association, 2022.

[33] David Lion, Adrian Chiu, and Ding Yuan. M3: end-to-end memory management in elastic system software stacks. In *EuroSys*, pages 507–522. ACM, 2021.

[34] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. Mako: a low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, pages 92–107. ACM, 2022.

[35] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *ACM SIGOPS Operating Systems Review*, 50(2):457–471, 2016.

[36] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[37] Pascal Maissen, Pascal Felber, Peter G. Kropf, and Valerio Schiavoni. Faasdom: a benchmark suite for serverless computing. In *DEBS*, pages 73–84. ACM, 2020.

[38] Microsoft. Azure functions. https://azure.microsoft.com/services/functions/, 2023.

[39] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016.

[40] Khanh Duc Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 675, 2015.

[41] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 57–69, Berkeley, CA, USA, 2018. USENIX Association.

[42] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *ASPLOS*, pages 753–767. ACM, 2022.

[43] 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.

[44] 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. USENIX Association, 2020.

[45] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference*, page 35. ACM, 2018.

[46] Markus Thömmes. Squeezing the milliseconds: How to make serverless platforms blazing fast, 2017.

[47] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud. In *SoCC*, pages 78–93. ACM, 2022.

[48] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS*, pages 559–572. ACM, 2021.

[49] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *OSDI*, pages 261–280. USENIX Association, 2020.

[50] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In *OSDI*, pages 35–53. USENIX Association, 2022.

[51] Jingjing Wang and Magdalena Balazinska. Elastic memory management for cloud data analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 745–758, Santa Clara, CA, 2017. USENIX Association.

[52] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[53] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 133–145, Berkeley, CA, USA, 2018. USENIX Association.

[54] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[55] Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, Haibing Guan, Sanhong Li, Chuansheng Lu, and Tongbao Zhang. Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 159–172, 2020.

[56] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. Bridging the performance gap for copy-based garbage collectors atop non-volatile memory. In *EuroSys*, pages 343–358. ACM, 2021.

[57] 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.

[58] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12. ACM, 2019.

[59] Yifei Zhang, Tianxiao Gu, Xiaolin Zheng, Lei Yu, Wei Kuai, and Sanhong Li. Towards a serverless java runtime. In *ASE*, pages 1156–1160. IEEE, 2021.

[60] Ziming Zhao, Mingyu Wu, Xujie Cao, Haibo Chen, and Binyu Zang. Flock: Towards multitasking virtual machines for function-as-a-service. *IEEE Trans. Computers*, 72(11):3153–3166, 2023.

[61] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. Beehive: Sub-second elasticity for web services with semi-faas execution. In *ASPLOS (2)*, pages 74–87. ACM, 2023.

A Artifact Appendix

A.1 Abstract

This artifact contains the source code of Desiccant and scripts to run the main experiments. Desiccant is proposed to optimize the memory efficiency of managed FaaS workloads. The design of Desiccant is mainly based on the observation of the frozen garbage problem, where the FaaS framework freezes cached function instances while leaving unused memory unreclaimed. To address this issue, Desiccant reclaims frozen garbage with the help of the language runtime, thereby improving memory efficiency and benefiting end-to-end performance under a fixed memory budget.

A.2 Description & Requirements

A.2.1 How to access. The artifact is publicly available at https://doi.org/10.5281/zenodo.10103366.

A.2.2 Hardware dependencies. We evaluate Desiccant on a server with dual Intel Xeon Gold 6138 CPUs. We use CPU 0-19 for the execution of the FaaS framework and test scripts, and CPU 20-39 to execute FaaS functions. Other x86_64 hardware can also run Desiccant by changing the CPU binding configuration as described in the README.md of the artifact.

A.2.3 Software dependencies. We evaluate Deisccant on Ubuntu 20.04, but higher versions may also be acceptable.

A.2.4 Benchmarks. Desiccant is evaluated based on the Azure Functions Trace Dataset available at https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md, as well as a set of FaaS functions described in the paper. We have included all of them in the artifact.

A.3 Set-up

The detailed set-up guide can be found at https://github.com/SJTU-IPADS/Desiccant-artifacts#3-setup.

A.4 Evaluation workflow

This artifact contains two major claims (C1, C2), which are proven by five experiments (E1-E5), as listed in Table 2.

A.4.1 Major Claims.

- (C1): Desiccant can reclaim frozen garbage from functions under different enviornments and memory configurations. This is proven by the experiments described in Section 5.2 (E1, E2), 5.4 (E4) and 5.5 (E5), whose results are shown in Figure 7, Figure 8, Figure 11 and Figure 12.
- (C2): Desiccant can imporve end-to-end function execution performance under a fixed memory upper bound. This is proven by the experiments described in Section 5.3 (E3), whose results are shown in Figure 9 and Figure 10.

A.4.2 Experiments. Our experiments have been automated using scripts. For each mentioned figure, the artifact contains automated scripts that can directly reproduce the data similar to that presented in the paper. To run the experiment, navigate to the corresponding directory and execute the command `./run.sh`. To parse the result, execute `./parse.sh` in the same directory. The script will output the parsed result to stdout in CSV format, whose caption and data would be similar to what is shown in the corresponding figure. The mapping relationship of directories and figures is shown in Table 2.

Table 2. The mapping relationship of the artifact

Directory	Experiment	Figure	Claim
./exp/fig7	E1	Figure 7	C1
./exp/fig8	E2	Figure 8	C1
./exp/fig9,10	E3	Figure 9 & Figure 10	C2
./exp/fig11	E4	Figure 11	C1
./exp/fig12	E5	Figure 12	C1