BalloonJVM: Dynamically Resizable Heap for FaaS

Abraham Chan, Kai-Ting Amy Wang, Vineet Kumar
Huawei Technologies, Markham, Canada
Email: {abraham.chan, kai.ting.wang}@huawei.com, {vineet.kumar}@mail.mcgill.ca

***Abstract*—Serverless computing, or more specifically, Function as a Service (FaaS), offers the ability for software developers to quickly deploy their applications to the public without worrying about custom server architecture. However, developers using FaaS services must be cautious not to exceed their container memory limits. For FaaS developers using Java, a spontaneous out of memory exception could terminate their application. This could prompt some developers to consider scalability rather than focusing on functionality, reducing the advantage of FaaS. In this paper, we present BalloonJVM, which applies ballooning, a memory reclamation technique, to dynamically resize the heap for Java FaaS applications, deployed on Huawei Cloud's FunctionStage system. We explore the challenges of configuring BalloonJVM for production and outline opportunities for improving both developer and service provider flexibility.**

Keywords–Ballooning; Function-as-a-Service; Serverless; Runtime environment; JVM Configuration.

I. INTRODUCTION

The Function as a Service (FaaS) programming model runs user-defined code in a process, typically a high-level language runtime, inside an operating-system-level container. FaaS is built upon the serverless architecture, which allows developers to deploy their applications on the public cloud in lieu of custom servers. A growing number of developers and companies are choosing to deploy their applications in this model to avoid the expenses of setting up and maintaining custom server infrastructure [1][2]. FaaS also offers the added advantage of billing developers only for the usage incurred.

Today, FaaS developers must carefully craft their functions so that its runtime memory usage is within the memory limit of its container, enforced through Linux's *cgroups* feature [3]. Exceeding the cgroups limit terminates the application abruptly. The developer must relaunch the application with the next large sized container. Such abrupt termination is unwarranted. Both the developer and service provider could benefit if it were possible to dynamically increase the heap size for a memory needy application while charging for the enlarged container.

Many service providers, including Huawei, use Oracle's Java Virtual Machine (JVM) to execute Java programs on FaaS. Oracle JVM contains a maximum heap option to control the application's memory usage, similar to the cgroups resource limit. Typically, a JVM running inside a 128MB container is started with a maximum heap setting of `-Xmx=128M`. Dynamically resizing the JVM heap is not supported in Oracle JDK. JVM throws an unrecoverable Out-Of-Memory (OOM) exception when the heap usage exceeds the maximum size.

If high memory limits were pre-allocated to applications, this could impact both the service provider and FaaS developers negatively. Higher pre-allocated memory for applications could diminish the number of FaaS applications runnable concurrently on a shared cloud infrastructure, reducing the service provider's profitability. On the other hand, FaaS developers could pay more for unused memory resources.

Ballooning is a memory reclamation technqiue, used by hypervisors to leverage unused memory by guest Virtual Machines (VMs) [4]. Each guest VM is allotted a large memory, but the guest only uses a portion of that memory in practice. The remaining memory space can be filled with balloons, which are pre-occupied memory spaces to an application (i.e., guest VM, JVM), but are actually empty memory spaces to the operating system (OS). This means the host OS is free to use the memory reclaimed through the balloons. When a guest VM requires more memory, the host can free the balloons inserted in that guest VM.

In this paper, we adopt ballooning for FaaS and expose it as a set of Java Application Programming Interfaces (APIs). We present *BalloonJVM*, a modified Java FaaS framework that calls ballooning APIs when invoking JVM to achieve dynamic memory adjustment. BalloonJVM is deployed on Huawei Cloud FunctionStage [5], a FaaS platform allowing user defined functions to be invoked on-demand. BalloonJVM is built on top of our prior work, ReplayableJVM [6], which features a checkpoint and restore framework that enables JVM to launch from an existing image to avoid its cold startup time. BalloonJVM can be launched with a larger maximum heap size than initially required (i.e., `-Xmx=512M` when only 128M is needed). Then, BalloonJVM inserts balloons at initialization and free balloons as additional runtime memory is required - creating the effect of dynamic memory resizing. This offers FaaS developers more flexibility over conventional fixed heap JVMs. Our approach does not modify JVM internals since maintaining a custom JVM build is expensive. The incorporation of BalloonJVM will provide an extra option to many FunctionStage users worldwide.

In summary, we make the following contributions in this paper.

- We present BalloonJVM, a FaaS framework with a resizable JVM heap, by developing a set of novel Java APIs that adapt ballooning for FaaS.
- We make recommendations of deployment configurations of BalloonJVM based on a runtime and memory analysis using eight representative FaaS applications.
- We ensure that BalloonJVM contains properly pinned balloons, such that no memory spikes occur as object memory is shifted around in the heap.

The remainder of the paper is organized as follows. In Section II, we offer a motivating example of how BalloonJVM helps FaaS developers. Then, in Section III, we outline our implementation of ballooning and while in Section IV, we describe GC principles that impact BalloonJVM. In Section V, we evaluate the feasibility of BalloonJVM using FaaS benchmarks, and in Section VI, we discuss the implications and limitations of our work. Later, we discuss related work in Section VII. Finally, in Section VIII, we conclude the paper.

 ISBN: 978-1-61208-703-0

II. Motivating Example

Consider a Java application on FaaS that provides a simple Key-Value (KV) store. Each request to the insertion function of the KV store allocates memory to insert a new object into an underlying hash map. Figure 1 shows the occupied heap memory compared to the total heap. Eventually, the memory allocated for objects in the KV store will reach the maximum heap size. In a regular JVM instance without ballooning, the application encounters an OOM exception. BalloonJVM ensures that a balloon, if one remains, is released before an OOM occurs. This increases the maximum heap available to the application, thus, evading the OOM.

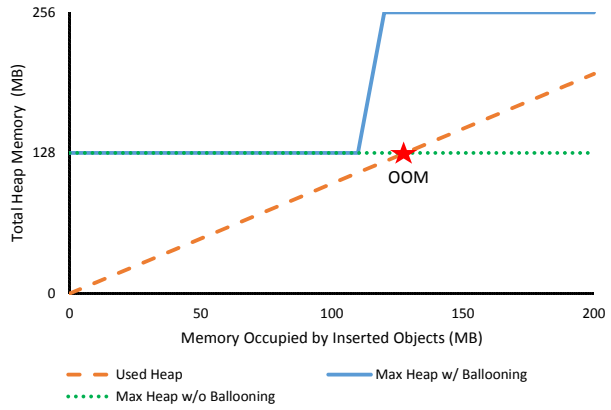

Figure 1. Avoiding an OOM exception with ballooning.

Without ballooning, the developer must ensure that the memory occupied by the KV store does not reach capacity in order to avoid a service disruption. To achieve this, the developer could either stop accepting new data, distribute insertions into another instance or write to a remote database. All of these options may prove to be more costly to the developer of a small upstart app than paying marginally more for dynamically increased heap space. BalloonJVM provides this latter option to developers, who wish to avoid infrastructure considerations for a simple deployment on FaaS.

III. Ballooning

BalloonJVM uses a variation of the memory ballooning technique, presented by OSv [7]. Unlike OSv, all of our ballooning features are exposed as a set of Java APIs, which are called by BalloonJVM to achieve ballooning. Our solution consists of two parts: *balloon insertion* and *balloon deletion*. BalloonJVM inserts balloons during the initialization of the JVM FaaS instance, while deleting balloons during the execution of JVM, between FaaS invocations.

A. Balloon Insertion

Balloon insertion is divided into two APIs: balloon *inflation* and *deflation*. Balloon inflation is the creation of the balloons in the JVM heap and unmapping them from the OS memory space. Balloon deflation involves the deallocation of OS memory occupied by balloons and returning it to the OS. Figure 2 shows balloon inflation in the first process, followed by balloon deflation in the second process. Balloons are implemented as a two dimensional Java byte array and are inflated and deflated natively through the Java Native Interface (JNI).

Balloon Inflation. Each balloon is created by allocating a single dimension array of a given balloon size in a 2D byte array. The memory held by the balloons is unmapped between JVM and the OS using the `munmap` system call, invoked through JNI. While the byte array represents used memory space to both JVM and the OS, JVM can no longer reference the balloon memory legally.

Balloon Deflation. After GC, each balloon is deallocated using the `madvise` system call with `MADV_DONTNEED` advice, through JNI. This advises the OS that the memory space occupied by the balloon is no longer needed in the near future. The OS has become aware that the balloon is free space.

After Balloon Insertion. At the end of balloon insertion, JVM holds references to inserted balloons and still thinks the balloon occupy their equivalent OS memory. However, through compacting, JVM will not touch the balloons during GC.

Compacting the Balloons. Once the balloons are inserted and deflated, it is important that the balloons are not moved by the Garbage Collector (GC) unless the corresponding JVM reference is also deleted. Otherwise, the mapped out pages may be mapped back in, resulting in a sudden jump in resident set size (RSS) and may lead to a JVM crash. To overcome this, we ensure that the balloons are inserted at the beginning of the old generation and compacted before deflation. We explicitly call GC multiple times to compact the inserted balloons and tenure them to the old generation. We verify that GC is actually invoked by analyzing the output of `jstat`, a JVM statistics monitoring tool. Additionally, the inflation and deflation of the balloons is implemented as a static block so that it executes before JVM runs `main()`, ensuring that the balloons are inserted before other objects are present. Note that our particular implementation is suitable for *Serial GC* and may not work for other garbage collectors.

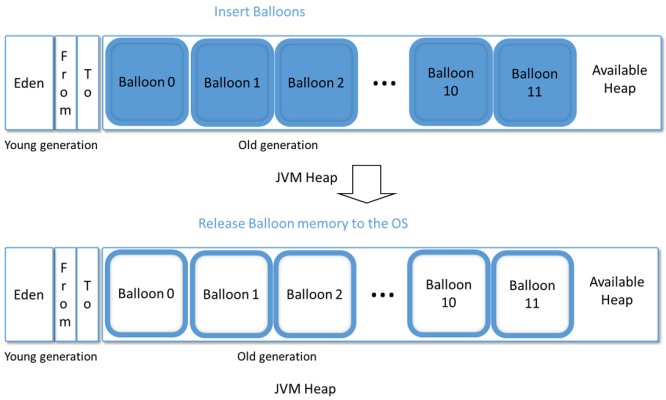

Figure 2. Balloon insertion.

B. Balloon Deletion

Our balloon deletion API calculates the amount of ballooned memory to release based on the size and number of balloons inserted. We also provide an option, Pre-Balloon Memory Utilization Ratio (PMUR), to control the memory used before balloons are deleted.

Implementation. The balloon deletion API is implemented by deleting the JVM reference to the balloon. This will trigger a GC, which frees up the Java heap space and allow JVM to reclaim memory from the OS. One shortcoming of balloon deletion, including our implementation, is the impact on JVM performance from the relative sizing of the old and

 ISBN: 978-1-61208-703-0

young generation heap space [8]. In Section V, we empirically explore the feasibility of using different configurations on BalloonJVM to minimize this impact.

Size and Number of Balloons. Our insertion API provides options for the number of balloons to be inserted and the size of each balloon. We accept balloons of any size, as long as all balloons for a single configuration are equally sized. From this point onward, we refer to the current JVM maximum heap size as the container size in MB. We also represent this using the variable, C. For instance, we choose a balloon size of 128MB and insert 11 equally sized balloons in a max heap of 1.5GB. This sample configuration is aimed to support a container size of 128MB, with an eventual allowance to 1.5GB as heap usage grows. When JVM is initially started with a 128MB container, no balloons are deleted. When a 256MB container is needed, one balloon is deleted, and eventually for the 1.5GB container, all balloons are deleted.

Equation (1) determines the number of balloons for deletion, B_{del}, where C is the initial container size, H is the eventual max heap, B_{ins} is the number of balloons inserted, and S is the balloon size.

$$B_{del} = \max(\left\lceil \frac{C - (H - B_{ins} \times S)}{S} \right\rceil, 0) \quad (1)$$

Pre-Balloon Memory Utilization Ratio (PMUR). The PMUR is defined as the ratio between the memory used and the total heap memory available before a balloon is released, ranging from 0 to 1. If PMUR is close to 1, an OOM may occur before a balloon is released. If PMUR is too low, the developer will be forced to pay for a larger heap space, when free heap space is still available. We find that a value around 0.85 works best, through experimentation described in Section V-G.

IV. Garbage Collection (GC)

We observe that by tuning the generational heap, BalloonJVM can reduce time-consuming GCs, especially Full GCs.

Generation GC. Generational GC separates the heap into a new and old generation. Newly allocated objects that survive several rounds of GCs are tenured to the old generation [9]. Separate algorithms can be deployed for young and old objects to maximize the efficiency of the GC.

New Generation (NewGen). This section of heap is where all new objects are stored. It is further divided into the eden and survivor spaces. In this paper, we refer to the NewGen as the combination of the eden and survivor spaces. The eden space hosts the newly allocated objects before any GC occurs, while a pair of survivor spaces host objects that survive at least one GC, awaiting promotion to the old generation. BalloonJVM uses the parameters, `NewSize` and `MaxNewSize`, in Oracle JVM to control the NewGen size.

Old Generation (OldGen). This section of heap hosts objects that survived enough GCs to be considered old objects. GC events occur less frequently in the OldGen compared to the NewGen. BalloonJVM always ensures that balloons are tenured to the OldGen to exploit this property.

Young GC (YGC) and Full GC (FGC). *YGCs* clean up the new generation. Since objects in the new generation build up quickly, YGCs occur relatively frequently and its algorithms optimize for speed. *FGCs* clean up both the old and new generations. In contrast to YGCs, they occur infrequently - this allows its algorithms to optimize for space over speed. While both FGCs and YGCs consume execution time, FGCs typically take longer to run than YGCs.

GC Algorithm. There are several GC algorithms offered by Java 8, which BalloonJVM uses, but all of them are generational GCs. BalloonJVM uses Serial GC and we found that it compacts balloons sufficiently. Serial GC exhibits a stop-the-world behaviour, meaning it pauses the operation of the application. It is typically used for smaller heaps (i.e., heaps of 1.5GB or smaller) while faster algorithms like Parallel GC are used for large heaps [9]. Serial GC avoids synchronization overhead for tracking live objects, required in Parallel GC.

V. Evaluation

We evaluate BalloonJVM with respect to these questions.

1) Is it feasible for one configuration to support all containers?
2) How do we choose a NewGen size for BalloonJVM?
3) What is the feasibility of using two configurations?
4) Does BalloonJVM ensure that balloons are pinned?

A. Experimental Setup

The experiments are performed on an Intel Xeon CPU E5-2687W, which is a SandyBridge EP @ 3.0 GHz machine with 12 cores and with HyperThreading enabled. It has 30MB of L3 cache and 256GB of memory. Ubuntu 16.04 is used as the base OS together with Docker 1.12.6.

For the remainder of this paper, *DefaultJVM* refers to Oracle HotSpot 64-Bit Server VM version 1.8.0_151 with a fixed new to old generation heap ratio of 1:2, running on Huawei FunctionStage. We use DefaultJVM as our baseline as it is the most prevalent default configuration for JVM on the cloud [8]. *BalloonJVM* is DefaultJVM with ballooning enabled and a variable NewGen size. Both JVMs run in a cgroup, allowing the service provider to exploit namespace isolation, resource limitation, and checkpoint/restore [6]. Otherwise, a JVM in a cgroup behaves the same as a standalone JVM.

B. Benchmarks

We use eight different benchmarks that represent FaaS applications of varying workloads and domains [10]: `Allocation`, `DataFilter`, `Inverse`, `Sort`, `TF-IDF`, `ThumbNail`, `TimeStamp` and `Unzip`. We found a lack of benchmark suites for FaaS, so we manually adapted all of our benchmarks to lambda functions. Lambda functions for FaaS are typically self-contained, repetitive tasks that are triggered by external events and its execution cannot exceed a strict timeout. `Allocation` allocates 1MB of memory in a static array list for each service request. It represents the workload of memory intensive FaaS applications (i.e., a KV store). `DataFilter` filters an array of random words based on a search query, representing data querying. `Inverse` computes the inverse of a 9x9 matrix, used in machine learning. `Sort` sorts an array of random words alphabetically. `TF-IDF` computes the statistical importance of a word in relation to a document in a corpus. `Thumbnail` converts a JPEG photo into a thumbnail, representing multimedia processing applications. `TimeStamp` outputs the current datetime as a string. `Unzip` uncompresses a zip file, performing file I/O. `Inverse` and `ThumbNail` represent workload intensive applications while `TimeStamp` and `Unzip` represent light utility applications.

 ISBN: 978-1-61208-703-0

C. Metrics

We define a configuration as *feasible* if it has a low runtime overhead and a high *Actual Memory Utilization Ratio* (AMUR).

a) Runtime Overhead: We define this as the runtime performance overhead of BalloonJVM over DefaultJVM. FaaS functions deployed on BalloonJVM should not incur a high runtime overhead over a similar deployment in DefaultJVM. We measure the runtime duration of a benchmark function in nanoseconds, using `System.nanoTime()`. The runtime excludes the time taken for the FaaS framework to initialize since BalloonJVM uses a checkpoint and restore mechanism [6]. All of the runtime durations are averaged over 100 runs. The overhead is then computed by $(T_b - T_d)/T_d$, where T_b and T_d are the times taken to execute the same function in BalloonJVM and DefaultJVM, respectively.

b) Actual Memory Utilization Ratio (AMUR): The AMUR is defined as the percentage of actual used memory over the container memory size. The AMUR can help guide the selection of the PMUR, discussed in Section III-B. The actual used memory is measured by counting the maximum number of objects, with a size of 1MB each, which can be allocated before JVM throws an OOM exception. This metric provides an approximation of the total heap space utilization before either a balloon is freed or an OOM finally occurs when no further balloons can be freed. A larger AMUR frees fewer balloons, preserving server resources and allowing developers to be charged at a lower tier of memory usage.

D. Heap Flexibility at What Cost?

BalloonJVM provides memory benefits but at what overhead to DefaultJVM? To answer this, we measure the overhead of each request to `Allocation` until DefaultJVM reaches an OOM. We initialize BalloonJVM with a container size, C (MB) of 128, with a max heap of 512MB and DefaultJVM with a fixed heap of 128MB. We see in Figure 3 that the overhead is roughly 10% on average, but spikes at certain requests. We find that the spikes are correlated with GC events - the upwards spikes represent GCs invoked by BalloonJVM while the downward spikes represent GCs invoked by DefaultJVM. Hence, to improve the performance of BalloonJVM, we need to tune the heap parameters to reduce GCs. In this paper, we manually tune the heap using benchmark programs and determine the configuration's feasibility.

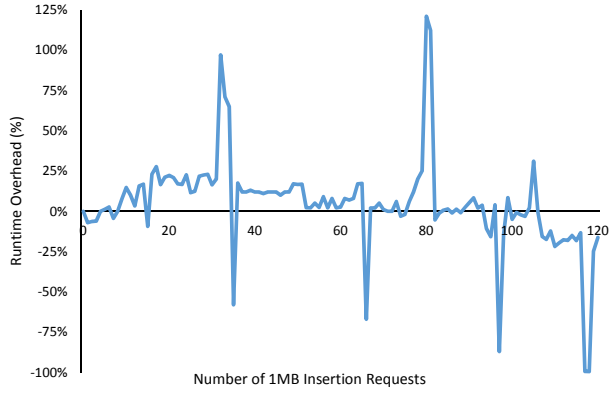

Figure 3. Overhead of BalloonJVM over DefaultJVM, at $C = 128$.

E. RQ1: Feasibility of a Single Configuration

In this section, we experimentally determine whether it is feasible to use a single max heap configuration, shown in Figure 4, to support C of 128, 256, 512, 1024 and 1536. To initialize our experiment, we allocate 110MB to the NewGen in order to maximize its use. In this single configuration, the eventual max heap size for BalloonJVM is 1536MB. The OldGen occupies the remainder of the heap, 1426MB, and is filled with 11 balloons of 128MB each. A NewGen of 110MB enables 18 MB of objects to be promoted to the OldGen for $C = 128$, while all 11 balloons remain. For $C = 256$, about 146MB of objects can be promoted to the OldGen with 10 balloons. Free OldGen space grows as balloons are released.

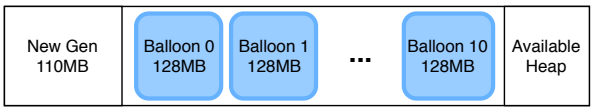

Figure 4. Single configuration, with an eventual max heap of 1.5GB.

We run the `Allocation` benchmark in DefaultJVM and BalloonJVM to measure the runtime overhead. As discussed in Section V-B, `Allocation` represents the most memory intensive application. As shown in Table I, the overheads for $C = 128, 1024, 1536$ exceed 10%. We then measure the YGC and FGC counts during the execution of the function in both JVMs using `jstat`. In Table I, we observe that `Allocation`, running in BalloonJVM, using $C = 1024$ and $C = 1536$, incur 9 and 15 extra YGCs respectively. $C = 128$ encounters an extra FGC. By inspecting the `jstat` output after each object inserted, we notice that the eden space is quickly exhausted. This is caused by insufficient memory for the NewGen, leading to frequent YGCs. Increasing the NewGen beyond 110MB is not practical to support the 128MB container, as there would be insufficient space for tenured objects and inserted balloons in the OldGen. Based on this observation, we conclude that a single configuration to support our range of container sizes is not feasible.

TABLE I. OVERHEADS AND GC COUNTS OF BALLOONJVM (B) VS DEFAULTJVM (D), USING ALLOCATION AND A SINGLE MAX HEAP.

C (MB)	Objs inserted	Overhead	YGC		FGC	
			D	B	D	B
128	100	13%	1	1	1	2
256	200	5%	2	2	1	1
512	400	6%	4	5	0	0
1024	800	21%	3	12	2	2
1536	1200	13%	3	18	4	2

F. RQ2: Choosing the NewGen Size

As shown in Figure 5, we partition the container sizes into two configuration groups, one for $C = 128, 256, 512$, and the other for $C = 1024, 1536$. For simplicity, we will refer to the former as Config A and the latter as Config B. Since a single configuration is not feasible, we wish to support the range of container sizes with as few partitions as possible, to reap the benefits of ballooning. It is important to note that the partition we made is only one possible combination - others may exist.

We explore the challenge of finding an appropriate NewGen size for the two configurations. For Config A, there is little leeway for choosing the NewGen size. A NewGen size

of 110MB is feasible, as it provides just enough OldGen space for tenured objects. For Config B, there is an opportunity to increase the NewGen size in a max heap of 1.5GB. We measure the runtime overheads using three different NewGen sizes across eight benchmarks in Figure 6 for the 1024MB and Figure 7 for the 1536MB container sizes respectively. We observe that a 400MB NewGen size offers the lowest overhead for `Allocation`, while 200MB gives the lowest overhead for other benchmarks. `Allocation` contains a lambda class member variable, whose reference is retained between invocations. Other benchmarks contain mostly transient objects, which are deleted after an invocation. Benchmarks with many transient objects may benefit from a smaller NewGen as a YGC is triggered earlier, and tenured objects are subject to infrequent FGCs. Alternatively, a larger NewGen for such benchmarks can result in slower YGCs, by traversing objects that should be tenured. Despite this, the configuration overheads differ less than 10% in the worst case or 5% on average.

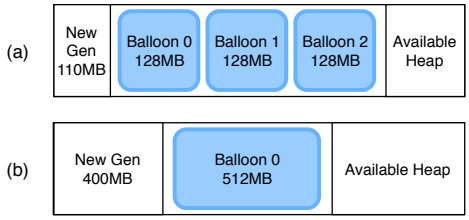

Figure 5. Two configurations approach. (a) Max heap of 512MB, with 3 balloons. (b) Max heap of 1.5GB, with 1 balloon.

TABLE II. NUMBER OF YGCS AND FGCS OF BALLOONJVM VS DEFAULTJVM, USING TWO CONFIGURATIONS.

Config	C (MB)	YGC		FGC	
		Default	Balloon	Default	Balloon
A	128	4	1	1	2
	256	4	3	3	2
	512	4	4	0	0
B	1024	3	3	2	2
	1536	3	4	4	3

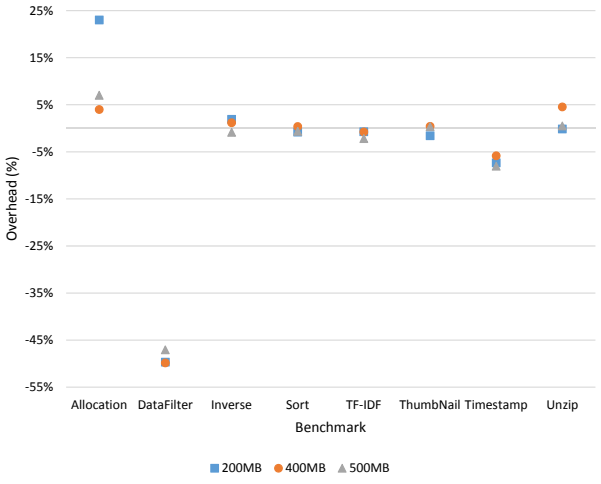

Figure 6. Performance overhead of BalloonJVM over DefaultJVM, with varying NewGen sizes, at C=1024MB.

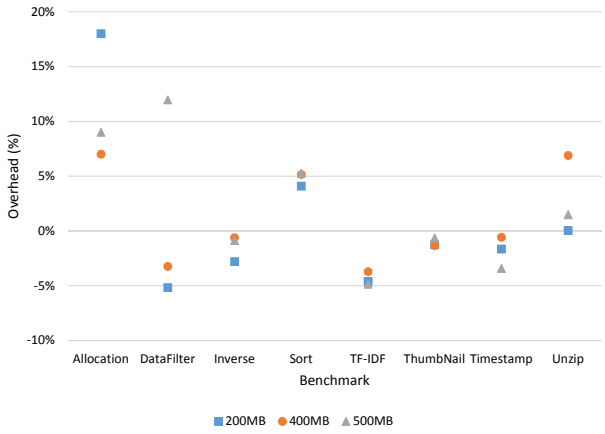

Figure 7. Performance overhead of BalloonJVM over DefaultJVM, with varying NewGen sizes, at C=1536MB.

G. RQ3: Feasibility of using Two Configurations

We evaluate the feasibility of using Config A and B with a NewGen of 400MB, for memory heavy workloads (i.e., `Allocation`).

First, we measure the AMURs of DefaultJVM and BalloonJVM as shown in Table III. We observe that BalloonJVM enables about the same amount of allocatable memory as DefaultJVM but the AMUR drops slightly in configurations with more balloons (i.e., $C = 128$ has 3 balloons vs. $C = 512$ has none). By analyzing the GC activity, we see that a FGC immediately follows a YGC whenever the OldGen is full, in this case, occupied by balloons. This bypasses the survivor space, reducing the overall usable heap memory.

Next, we measure the overhead of BalloonJVM against DefaultJVM across different container sizes using `Allocation`. To effectively evaluate BalloonJVM's performance, we condition `Allocation` to allocate a large number of objects, to approach its AMUR, and measure the overhead. As shown in Table IV, the overhead ranges from -56% to 14%. In some cases, benchmarks run faster in BalloonJVM than on DefaultJVM. In Table II, YGCs and FGCs drop sharply compared to the previous single configuration. Based on these observations, we conclude that this two configuration approach is feasible for allocation heavy workloads. However, the configuration feasibility does depend on the workload of the target applications.

TABLE III. AMUR OF BALLOONJVM AND DEFAULTJVM, USING TWO CONFIGURATIONS.

Config	C (MB)	Default	Balloon
A	128	94%	88%
	256	95%	93%
	512	97%	97%
B	1024	96%	95%
	1536	96%	97%

H. RQ4: Balloon Pinning

We evaluate BalloonJVM to determine whether the balloons inserted are properly pinned. If the balloons are not pinned, they may potentially move around the heap during GCs, causing the JVM RSS to jump abruptly. This will cause BalloonJVM to release balloons to compensate or crash JVM. We perform an experiment on only container sizes that have balloons. For Config A, this includes 128MB with 3

 ISBN: 978-1-61208-703-0

TABLE IV. OVERHEAD OF BALLOONJVM OVER DEFAULTJVM, USING TWO CONFIGURATIONS.

	C (MB)				
Config	**A**			**B**	
Benchmark	**128**	**256**	**512**	**1024**	**1536**
DataFilter	-56%	-11%	-11%	-50%	-3%
Inverse	-3%	-1%	-1%	1%	-1%
Sort	7%	5%	1%	0%	5%
TF-IDF	0%	-4%	-4%	-1%	-4%
ThumbNail	14%	-1%	1%	0%	-1%
Timestamp	4%	2%	5%	-6%	-1%
Unzip	4%	2%	2%	5%	7%

balloons and 256MB with 2 balloons. For Config B, this would be 1024MB with 1 balloon. For each eligible container, we allocate P MB of objects in the first request. In each subsequent request, we allocate A MB and randomly delete A MB of objects. The immediate allocation and deletion of large number of objects activates GC. We run the experiment on 128MB using $\{P = 95, A = 90\}$, 256MB using $\{P = 200, A = 150\}$, 1024MB using $\{P = 800, A = 500\}$ over 50k runs. We observe that the RSS remains consistent across all runs, showing that BalloonJVM's balloons are compact.

VI. DISCUSSION

A. Increased Flexibility

BalloonJVM offers increased flexibility to both the FaaS developer and cloud service provider. For the FaaS developer, BalloonJVM offers a safeguard when their application exceeds the initial maximum heap size. Developers may also deploy with a modest heap, and allow BalloonJVM to grow the heap as their application becomes more widely used. For the service provider, BalloonJVM allows improved resource sharing and adjustable price tiers. The memory occupied by balloons is directly returned to the OS, usable by other cloud applications. When multiple applications deployed on BalloonJVM release balloons, memory will be available on a first-come first-served basis. An OOM exception can occur if the actual memory is unavailable. Service providers can offer dynamic pricing where pricing jumps to the next tier when a balloon is freed.

B. Limitations

We identity three limitations to our work: the choice of GC algorithm, reinsertion of balloons, and the assumption of application functional correctness. We repeated our analysis by configuring both BalloonJVM and DefaultJVM to use Parallel GC, and found the runtime overheads to be feasible. However, we observe that RSS sharply increases as BalloonJVM fails to pin the balloons. This presents a challenge for deployment on large heaps where Parallel GC is desired over Serial GC. Secondly, BalloonJVM does not reinsert balloons after release, as we have not determined how to pin reinserted balloons when live data exists in the heap. Lastly, we assume that the FaaS application is functionally correct when requiring more memory. If the application erroneously consumes memory, BalloonJVM only delays its eventual failure through resizing.

VII. RELATED WORK

Ballooning is a widely used memory reclamation technique for VM memory management [4]. Ballooning to resize JVM's heap was first proposed in [7], which influenced the creation of BalloonJVM. However, our work differs in at least four major areas: we expose the features of ballooning as Java APIs rather than bundling it into an OS, we ensure that the inserted balloons are properly pinned by choosing a specific GC algorithm, we insert balloons before JVM is started without setting pressure criteria, and we implement ballooning for FaaS.

Salomie et al. [11] implement JVM ballooning by modifying the Parallel GC algorithm that is shipped with OpenJDK, requiring changes to the JVM internals. Hines et al. [12] present a framework called Ginkgo, which runs a background thread to monitor JVM heap usage and deletes or inserts balloons as needed. However, Java applications running on a Ginkgo resized JVM can experience high overhead as it does not consider the heap's generational nature.

VIII. CONCLUSION

Developers are choosing to deploy their applications on serverless architectures to avoid infrastructure costs. However, applications deployed on runtime environments like JVM are constrained by a maximum heap size. Developers either pay to overprovision or encounter a disruptive crash when the limit is exceeded. We present BalloonJVM, which utilizes ballooning, a memory reclamation technique, offering a dynamically resizable heap. Our results show that BalloonJVM can provide flexible memory benefits with less than 5% average overhead to typical FaaS applications, by carefully partitioning the generational heap. While BalloonJVM's ballooning implementation is specific for JVM, the concept can extend to other VM-based languages that constrain the heap size such as Node.js, another popular language for FaaS.

ACKNOWLEDGEMENT

We thank the anonymous reviewers and Tarek Abdelrahman for volunteering to provide feedback on our paper.

REFERENCES

[1] I. Baldini *et al.*, *Serverless Computing: Current Trends and Open Problems*. Springer, 2017, pp. 1–20.

[2] J. Jackson and L. Hecht, "TNS Guide to Serverless Technologies: The Best of FaaS and BaaS," http://thenewstack.io/guide-serverless-technologies-functions-backends-service, 2016, [Accessed: 21-Mar-2019].

[3] R. Buyya *et al.*, "A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade," *CoRR*, pp. 105:1–105:38, 2017.

[4] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.

[5] Huawei, "Functionstage," https://www.huaweicloud.com/en-us/product/functionstage.html, [Accessed: 21-Mar-2019].

[6] K. Wang, R. Ho, and P. Wu, "Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment," in *Proc. EuroSys'19*, 2019, pp. 39:1–39:16.

[7] A. Kivity *et al.*, "OSv: Optimizing the Operating System for Virtual Machines," in *Proc. USENIX ATC'14*, 2014, pp. 61–72.

[8] S. Sahin, W. Cao, Q. Zhang, and L. Liu, "JVM Configuration Management and Its Performance Impact for Big Data Applications," in *Proc. BigData Congress'16*, 2016, pp. 410–417.

[9] Sun Microsystems, "Memory Management in the Java HotSpot Virtual Machine," 2006.

[10] G. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research," 2017.

[11] T. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application Level Ballooning for Efficient Server Consolidation," in *Proc. EuroSys'13*, 2013, pp. 337–350.

[12] M. R. Hines, A. Gordon, M. Silva, D. D. Silva, K. Ryu, and M. Ben-Yehuda, "Applications Know Best: Performance-Driven Memory Overcommit with Ginkgo," in *Proc. CloudCom'11*, 2011, pp. 130–137.

 ISBN: 978-1-61208-703-0