Dynamic Vertical Memory Scalability for OpenJDK Cloud Applications

Rodrigo Bruno
INESC-ID / IST - Técnico, University of Lisbon
Portugal
rbruno@gsd.inesc-id.pt

Paulo Ferreira
INESC-ID / IST - Técnico, University of Lisbon
Portugal
paulo.ferreira@inesc-id.pt

Ruslan Synytsky
Jelastic, Inc.
California, USA
rs@jelastic.com

Tetiana Fydorenchyk
Jelastic, Inc.
California, USA
tf@jelastic.com

Jia Rao
University of Texas at Arlington
Texas, USA
jia.rao@uta.edu

Hang Huang
Huazhong University of Science and Technology
China
u201217509@hust.edu.cn

Song Wu
Huazhong University of Science and Technology
China
wusong@hust.edu.cn

Abstract

The cloud is an increasingly popular platform to deploy applications as it lets cloud users to provide resources to their applications as needed. Furthermore, cloud providers are now starting to offer a "pay-as-you-use" model in which users are only charged for the resources that are really used instead of paying for a statically sized instance. This new model allows cloud users to save money, and cloud providers to better utilize their hardware.

However, applications running on top of runtime environments such as the Java Virtual Machine (JVM) cannot benefit from this new model because they cannot dynamically adapt the amount of used resources at runtime. In particular, if an application needs more memory than what was initially predicted at launch time, the JVM will not allow the application to grow its memory beyond the maximum value defined at launch time. In addition, the JVM will hold memory that is no longer being used by the application. This lack of dynamic vertical scalability completely prevents the benefits of the "pay-as-you-use" model, and forces users to over-provision resources, and to lose money on unused resources.

We propose a new JVM heap sizing strategy that allows the JVM to dynamically scale its memory utilization according to the application's needs. First, we provide a configurable limit on how much the application can grow its memory. This limit is dynamic and can be changed at runtime, as opposed to the current static limit that can only be set at launch time. Second, we adapt current Garbage Collection policies that control how much the heap can grow and shrink to better fit what is currently being used by the application.

The proposed solution is implemented in the OpenJDK 9 HotSpot JVM, the new release of OpenJDK. Changes were also introduced inside the Parallel Scavenge collector and the Garbage First collector (the new by-default collector in HotSpot). Evaluation experiments using real workloads and data show that, with negligible throughput and memory overhead, dynamic vertical memory scalability can be achieved. This allows users to save significant amounts of money by not paying for unused resources, and cloud providers to better utilize their physical machines.

CCS Concepts • **Software and its engineering → Memory management**; **Garbage collection**;

Keywords JVM, Garbage Collection, Container, Scalability

ACM Reference Format:
Rodrigo Bruno, Paulo Ferreira, Ruslan Synytsky, Tetiana Fydorenchyk, Jia Rao, Hang Huang, and Song Wu. 2018. Dynamic Vertical Memory Scalability for OpenJDK Cloud Applications. In *Proceedings of 2018 ACM SIGPLAN International Symposium on Memory Management (ISMM'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3210563.3210567

ISMM'18, June 18, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5801-9/18/06…$15.00
https://doi.org/10.1145/3210563.3210567

1 Introduction

The cloud is an increasingly popular platform to deploy applications. One of the main factors contributing to this popularity is the simplicity in how resources can be added (or removed) to an application as they are (or not) needed. Furthermore, cloud providers have started to offer a "pay-as-you-use" model instead of the traditional "pay-as-you-go" model.[1] The models "pay-as-you-use" and "pay-as-you-go" are different in the way resources are assigned to applications (note that an application might be running on top of a virtual machine [3] or inside a container [2]). In a "pay-as-you-use" model, cloud users only pay for the resources that are actually being used by their applications as opposed to the "pay-as-you-go" model, in which users pay for statically sized virtual machines or containers. When using the "pay-as-you-use" model, unused resources are given back to the virtualization or container engine, allowing other applications to take advantage of such resources. From now on, for simplicity, the term instance will be used when both a virtual machine or a container are equally applicable. Host engine will also be used when a virtualization engine (hypervisor) or a container engine are equally applicable.

The "pay-as-you-use" model is usually achieved using containers or virtual machine ballooning [14, 18]. For example, if an application needs to grow its memory, the host engine might grant more memory to the instance where the application is running. Once the application memory requirements decrease, the host engine is able to reclaim back the extra memory that is not being used by the application.

However, runtime environments such as the Java Virtual Machine (JVM) fail to dynamically adjust their resource requirements at runtime in a variety of workloads (more details in Section 2.1). In particular, if an application needs more memory than what was given to the JVM at launch time, the JVM does not allow the application to use more memory and therefore, the JVM must be re-launched with a higher memory limit. The opposite problem is also present, i.e., if an application is not using all the memory that once was used, it is not possible to control/force the JVM to give this memory back to the host engine.

In sum, JVM applications cannot dynamically scale their memory requirements. Once the JVM is launched, the memory limit is fixed and cannot be changed. This means that if the memory requirements change, the host engine would be able to scale the memory resources as the application needs but the JVM does not allow it due to architectural restrictions. This creates a gap between the real resource needs and what the customer is paying for.

[1]White papers presenting these concepts can be found at www.infoq.com/articles/java-cloud-cost-reduction and www.forbes.com/sites/forbestechcouncil/2018/03/28/deceptive-cloud-efficiency-do-you-really-pay-as-you-use.

The main goal of this work is to allow JVMs (the OpenJDK in particular) to dynamically scale their memory usage taking into consideration an application's needs. Thus, if an application needs more memory than what was initially predicted at launch time, the JVM should be able to grow its heap. On the other hand, if an application does not need all the memory that was once used, the heap should shrink and unused memory should be given back to the host engine. In addition, and since this problem comes from the lack of support for vertical memory scalability in the JVM, no changes to the host engine, operating system, or user application should be required for this solution to work. Finally, design and implementation changes introduced into the JVM should not compromise application throughput, and should not require any downtime/restart in order to scale the application memory.

The challenge of dynamically scaling memory is that current JVMs, the OpenJDK in particular, impose a fixed limit on how much memory can be used throughout the application execution. Changing the JVM to allow a dynamic memory limit is not trivial as the internal JVM data structures are setup to work with a static memory limit, and cannot be easily setup again without rebooting the entire JVM.

There are clearly several naive solutions that may seem to solve the problem. For example, simply rebooting the JVM to provide a higher memory limit is not adequate; the application needs to be re-launched and this process takes time, and stops any service provided by the application. Using many JVMs with small heap sizes that could be created or destroyed, as more memory is needed or not, is also not adequate; in fact, such a solution does not work in situations where individual tasks require more memory than what is available in each single JVM. Finally, another naive solution is to simply setup a JVM with a very large memory limit. This provides an application with enough memory to run its workload but leads to resource/money waste whenever the application is not using all the memory that was reserved. In conclusion, there is currently no solution to providing dynamic memory vertical scalability in the JVM.

To solve this problem, we propose a new JVM heap sizing strategy that allows available memory to scale up and down according to a real application needs. In order to do so, two important steps are required.

First, a new runtime configurable memory limit for the heap size is provided; this limit, named `CurrentMaxMemory`, defines how much memory an application can currently use. Contrary to the static memory limit defined at launch time, `CurrentMaxMemory` can be re-defined at runtime, and can also be programmatically adjusted. One clear advantage of using `CurrentMaxMemory` (instead of the original JVM static limit) is that cloud users can adapt the amount of memory given to the JVM along the way, and do not need to guess at launch time the application memory requirements.

The second step to achieve the proposed goals is to rethink the Garbage Collection (GC) policies that control how much and when the heap memory is given back to the host engine. In the proposed solution, it is essential that unused memory gets uncommitted and thus given back to the host engine as soon as possible. To do so, we introduce additional logic into the JVM to trigger a heap compaction whenever the amount of unused memory is significant (this is controlled by a configurable variable). The heap reduction can be triggered at any time, not only during a GC.

We implement and evaluate the above mentioned JVM changes to provide dynamic vertical memory scalability on the OpenJDK 9 HotSpot JVM, the current release of one of the most used JVM implementations. We also modified the Parallel Scavenge collector and the Garbage First collector, the new default GC algorithm, to change the policies that control how and when the heap can grow or shrink.

The proposed solution is evaluated using the DaCapo benchmark suite as well as real workloads and data provided by Jelastic.[2] Through several experiments, we show that with a very small memory footprint overhead, we provide dynamic memory scalability with negligible application throughput overhead. This allows users to save money that would otherwise be spent on unused resources.

The contributions of this work are threefold. First, it presents a novel heap sizing strategy that allows the memory used by the JVM to scale up and down, as opposed to the current static strategy. This enables JVM applications to take advantage of the "pay-as-you-use" model. Second, the proposed solution is evaluated and different aspects of our solution are exercised showing very promising results. Finally, all the code is open source and a patch is being prepared to be sent to the OpenJDK, allowing everyone to benefit from it.

2 Motivation

This section presents concepts that are important to understand the motivation and design decisions of this work. First, it explains how resources can be used in the cloud and their billing models, further motivating the problem solved in this work. Then, it describes the target applications, for which the solution proposed in this work is mostly beneficial. This section closes with a description of important memory management concepts and architectural principles of the JVM. These are specially important to understand the design decisions proposed in our solution.

2.1 Resource Scalability in the Cloud

Elasticity is one of the key features in cloud computing; it allows host engines to dynamically adjust the amount of allocated resources to meet changes in application's workload demands [1]. Such a feature is crucial for scalability which can be provided along two dimensions: horizontal (adjusting the number of instances), and vertical (adjusting the resources assigned to a single instance). In the context of this work, we focus on the second one.

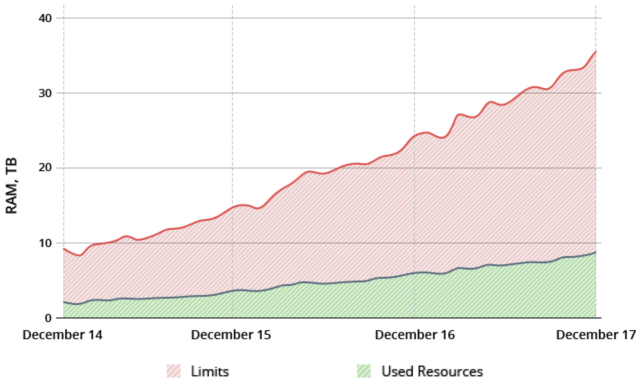

Figure 1. Jelastic Reserved vs Used Container Resources

Regardless of the scalability dimension used, cloud providers are currently enforcing one out of two different billing models: i) "pay-as-you-go", and ii) "pay-as-you-use". In the first, users are billed for statically reserved resources while in the second, users are billed for the actual used resources.

In this paper, we focus on the "pay-as-you-use" model, as there are a number of interesting use cases that benefit from it (more details in the next section). Figure 1 further motivates this problem by showing the difference between the used and reserved resources in Jelastic cloud for the last three years (from December 2014 to December 2017). By analyzing this chart, it is possible to observe that the difference between the reserved memory (Limits) and the actual used memory (Used Resources) increases through time. This means that the amount of unused memory, i.e., memory for which cloud users are paying but not using, is increasing. In December 2017, the amount of unused memory is above 26 TB in Jelastic cloud. In addition, the unused memory represents almost three times the amount of used memory (approximately 9 TB of used memory compared to 26 TB of unused memory). Therefore, there is an enormous potential to reduce the cost of cloud hosting for cloud users by using the "pay-as-you-use" model in which users to pay for the used resources (and not for the reserved resources).

Vertical scalability is a fundamental requirement for the "pay-as-you-use" billing model. In order to take advantage of it, both the cloud provider (i.e., the host engine) and the application running inside the instance must support vertical scalability. However, as described in Section 2.3, current runtime environments, specifically the OpenJDK HotSpot JVM, do not vertically scale.

[2]Jelastic is a decentralized multi-cloud provider that introduced the "pay-as-you-use" model. It can be reached at jelastic.com

Use Cases	Short Description
user sessions	user-based services save sessions (in memory) that timeout after some time
in-memory data	services that cache reads or consolidate writes that are flushed after some time
periodical jobs	services that run at particular times to retrieve, process, or store some data

Table 1. Examples of Target Use Cases

2.2 Target Cloud Applications

The solution proposed in this work targets applications that have periodical patters of memory usage, i.e., applications that can have periods of time with high memory usage and other periods of time with low memory usage. In this section, we analyze some real-world examples, based on Jelastic cloud provider, for which this class of applications represents a significant fraction of their client's applications.

Table 1 presents three examples of popular use cases in Jelastic cloud provider. The first example is related to user-based services that maintain in memory user sessions which timeout after some time. Web servers (such as Apache Tomcat) evidence this characteristic as most users of certain web sites are active during the day and, at night, almost no user is active. Therefore, most memory used during the day to hold user sessions is unused during the night. Another use case are services that keep data is in-memory to either provide fast reads or to consolidate writes (for example Hazelcast or Cassandra). This data is flushed after sometime and can lead to high amounts of memory being unused for longs periods of time. Finally, periodical jobs are another example of a very common use case. In such scenarios, schedulers can be used to launch services at specific times to retrieve, process, or store data. One popular example is periodical log processing.

To conclude, for all these use cases (that can hold unused memory for a long time), the "pay-as-you-use" model enables not only cloud users to save significants amount of resources (and therefore money), but also cloud providers to better use their machines (e.g., to support more applications in the same physical computer).

2.3 Memory Management in the JVM

Most runtime environments, the HotSpot JVM in particular, provide automatic memory management. This means that a user application does not need to explicitly allocate and deallocate memory. Instead, it is the responsibility of the runtime environment to manage memory (including its allocation and deallocation) providing it when the application requires so to hold some application data/objects.

Runtime environments (e.g., JVM) often provide memory to applications in the form of a heap abstraction. The heap, from an application point of view, is a continuous segment of memory that can be used to hold application objects. How the heap is implemented and managed is hidden from the application and is the responsibility of the Garbage Collector.

2.3.1 GC Data Structures

The Garbage Collector (component responsible for the GC) is a fundamental component in high level language runtime design. Among other tasks, the collector is responsible for reclaiming unused heap memory and thus making it available for future use. Unused heap memory is reclaimed during heap collections, during which the collector analyzes which objects are unreachable and thus need to be collected (and their memory freed).

In order to work efficiently, the collector maintains several auxiliary internal data structures that optimize the GC process. One of such data structures, for example, is the card marking table, which is used to keep track of pointers that cross heap sections.

These internal data structures are set up at JVM launch time and are prepared to deal with, at most, the heap size limit defined at launch time. Since i) these data structures are essential for the collector to work and, ii) they are continuously being read and updated by the collector, changing the amount of memory that these data structures must handle is not trivial. To do so, one would have to stop the whole JVM (including GC and application threads) to re-initialize these data structures. This would require significant engineering effort and would also lead to significant application downtimes. Therefore, currently, the only solution to change the heap size limit is to re-start the JVM, incurring into a significant application downtime.

2.3.2 Reserved vs Committed vs Used Memory

Users can only define the heap size limit at launch time and, at runtime, the application is assured to have a fixed memory area to place application objects (the heap). The application is also assured that there will be free space to allocate objects if the heap is not full with live objects (as the collector reuses memory that was occupied by unreachable objects).

However, the JVM grow or shrink the heap size at runtime (within the limits defined at launch time) according to different sizing policies. For example, if the amount of live objects keeps increasing and the current heap gets full, the collector will try to grow the heap (while remaining within the limit previously defined at launch time). On the other hand, if the used space is very low, the collector might shrink the heap during a collection. These heap operations (grow and shrink) will change the state of the heap memory.

In the JVM, heap memory can be in different states. We now present a simplified model, yet general enough to represent real implementations, consisting in three states:

- used, memory that is actually being used to hold application objects (which might be reachable/live or

unreachable/garbage). The used memory is a subset of the committed memory (defined in the next item);
- committed, memory that constitutes the actual heap. committed memory may contain live objects, unreachable objects (garbage waiting to be collected), or may be unused (free space for new application objects). The committed memory is a subset of the reserved memory (defined in the next item);
- reserved, memory whose address space is already reserved inside the JVM but may still be not committed in the JVM. Uncommitted memory (i.e., reserved memory that is not committed) does not have physical memory assigned to it.

Upon launch time, the JVM reserves enough memory to accommodate the maximum heap size defined by the user. The initial committed memory size, if not specified by the user, is computed through implementation specific GC heap sizing policies. The used memory is zero. Throughout an application execution, the committed memory (i.e., the heap size) may grow (up to the amount of reserved memory) or shrink depending on several factors such as increase or decrease of used memory.

These operations are controlled by different collector implementation specific heap sizing policies, and they are only executed when the heap is being collected. This leads to a significant problem for applications that do not trigger GCs during long periods of time (e.g., if applications are idle or do not allocate new objects). For these applications, there is no way to reduce the heap size even if the amount of unused memory is very high.

3 Dynamic Vertical Memory Scalability in the JVM

Dynamically adapting JVM's resources to better fit an application's needs is an increasingly important problem, essential to improve resource efficiency in the cloud. However, JVMs evidence some problems handling changes in the amount of resources given to them while running some classes of applications (such as the ones described in Section 2.2). Thus, we propose a re-design of the heap sizing strategy to allow JVMs to scale up and down the amount of memory handled by the JVM (and thus, the amount of memory available to the application). This new strategy consists of two main steps: i) define a dynamic maximum memory limit for JVM applications (see Section 3.1), and ii) adapt GC heap sizing policies to better fit the application needs (see Section 3.2).

3.1 Letting the Memory Heap Grow

In current JVM architectures, the size of the memory heap is statically limited by an upper bound, from now on named MaxMemory, defined at JVM launch time; this value affects how much memory is reserved and imposes a limit on how much memory can be committed (and therefore, used by the application). Committed memory starts, if not specified by the user at launch time, with a GC specific value that may depend on several external factors. Then, committed memory may grow if there is no space left to accommodate more live application objects. Once committed memory grows and fills all the reserved memory, no more heap growth is allowed and allocation errors will occur if more memory is necessary.

Algorithm 1 Set Current Maximum Heap Size

```
1:  procedure SET_CURRENT_MAX_MEMORY(new_max)
2:      committed_mem ← CommittedMemory
3:      reserved_mem ← MaxMemory
4:      if new_max > reserved_mem then
5:          return failure
6:      if new_max < committed_mem then
7:          trigger GC
8:          committed_mem ← CommittedMemory
9:          if new_max < committed_mem then
10:             return failure
11:     CurrentMaxMemory ← new_max
12:     return success
```

In order to allow the application to scale and to use more memory, the heap must keep growing. However, as discussed before (in Section 2.3.2), it is not trivial to increase the reserved memory at runtime (mainly due to difficulties with the resizing of GC internal data structures). To solve this problem, we propose a new dynamic limit on how much memory the application can use, named CurrentMaxMemory. This limit can be changed at runtime whenever the user decides that it is appropriate.

Increasing or decreasing this limit will result in more or less memory available for the heap. The committed memory can grow until it reaches CurrentMaxMemory. By definition, CurrentMaxMemory is a subset of MaxMemory (reserved memory) and contains CommittedMemory (see Expression 1).

$$\texttt{CommittedMemory} \subseteq \texttt{CurrentMaxMemory} \subseteq \texttt{MaxMemory} \tag{1}$$

The MaxMemory value must still be set (mainly because it is necessary to properly setup GC data structures) but it can be set conservatively to a very high value. This will only impact the reserved memory, which does not affect the instance memory utilization. It will also slightly increase the committed memory because larger (in terms of memory) GC internal data structures will be necessary to handle larger volumes of data. However, as shown in Section 5.4, this overhead is negligible and the committed memory overhead is hardly noticeable.

Algorithm 1 depicts how the CurrentMaxMemory value can be set at runtime. As previously explained, CurrentMaxMemory cannot be higher than MaxMemory, and thus the operation fails (line 5) if the new value is higher than MaxMemory

(reserved memory). On the other hand, if the new value is lower than CommittedMemory, we first need to try to reduce the committed memory so that the value of CommittedMemory is lower than the new value for CurrentMaxMemory. To do so, a GC cycle is triggered (line 7) and, after the cycle finishes, a new test is performed (line 9). If the new value for CurrentMaxMemory is still lower than CommittedMemory than the operation fails (line 10). Otherwise, a new value is assigned to CurrentMaxMemory.

By taking advantage of this operation, a user does not need to guess the application memory requirements at launch time, being able to control it (i.e., vertically scaling the JVM memory) by simply setting a new value for CurrentMaxMemory. This value can be also be changed programmatically.

3.2 Give Memory Back to the Host Engine

To be able to dynamically scale memory, the JVM must not only be capable of increasing its memory usage but must be also able to reduce it. Thus, when not using memory, the JVM must be able to free unused memory, and give it back to the host engine so that it can be used by other instances. We discuss in this section, how to properly scale down JVM memory usage.

The first step to scale down memory is to reduce the size of the JVM heap or, in other words, to reduce the size of the CommittedMemory. This operation usually occurs at the end of a GC cycle if the percentage of committed memory that contains no live objects (i.e., unused memory) is high. The problem, however, is that if no GC cycles are triggered (e.g., if an application does not need to allocate objects, or if an application is idle); in such a case, it is not possible to scale down memory and thus, memory is kept in the JVM although it is not being used.

To solve this problem, we propose the introduction of periodic memory scale down checks that verify if it is possible to scale down the JVM memory. If so, a GC cycle is triggered. The decision to trigger a GC cycle or not is based on two different factors: i) over committed memory (i.e., amount of committed memory that is not being used), and ii) time since the last GC. The goal is to reduce memory usage by uncommitting unused memory, but also not to disrupt the application execution by triggering very frequent collections.

Algorithm 2 presents a simplified version of the code that checks if a GC cycle should be triggered to resize the heap. This decision depends on two conditions: i) if the difference between the CommittedMemory and UsedMemory is above a specific threshold (MaxOverCommittedMemory), and ii) if the time since the last GC is above another specific threshold (MinTimeBetweenGCs).

In sum, if the over committed memory is high, it means that the JVM should scale down its memory. To avoid disrupting the application execution with potentially many GC calls to scale down memory, a scale down triggered GC cycle is only launched if no other GC cycle ran at least MinTimeBetweenGCs seconds ago (a configurable value).

Algorithm 2 Should Resize Heap Check

```
1:  procedure SHOULD_RESIZE_HEAP
2:      commit_mem ← CommittedMemory
3:      used_mem ← UsedMemory
4:      time_since_gc ← TimeSinceLastGC
5:      over_commit ← commit_mem − used_mem
6:      if over_commit < MaxOverCommittedMem then
7:          return false
8:      if time_since_gc < MinTimeBetweenGCs then
9:          return false
10:     return true
```

Both MaxOverCommittedMemory and MinTimeBetweenGCs are configurable at runtime. By controlling these two variables, users can control how aggressively the JVM will reduce its heap size. It is important to note that, the more aggressive the policies to scale down memory are, the more interference there will potentially be in the application execution (specially when the application is not idle). This topic is further discussed in Section 5.3).

3.3 Memory Vertical Scaling

In short, as described above, we propose two important changes to the JVM heap sizing strategy: i) introduce a configurable maximum memory limit, and ii) periodic heap resizing checks. These two features are essential for providing vertical memory scalability, and no feature could be discarded or replaced using already existing mechanisms inside the JVM.

On the one hand, the configurable maximum memory limit (CurrentMaxMemory) is essential to avoid guessing applications' memory requirements, and also to dynamically bound the memory usage. This dynamic limit could not be replaced by simply setting MaxMemory to a very high value as the user would not have any control on how much memory the application would really use, and therefore how much it would cost in the cloud.

On the other hand, periodic heap resizing checks are necessary to force the JVM to uncommit unnecessary memory. This is specially important for applications that might be idle for long periods of time, during which no GC runs and therefore, no heap resizing would be possible.

4 Implementation

The proposed ideas were implemented in the OpenJDK 9 HotSpot JVM, one of the most used industrial JVM implementations. In addition, we currently support two widely used OpenJDK collectors: i) Parallel Scavenge (PS), and ii) Garbage First (G1), the new by default and most advanced collector in HotSpot.

The provided implementations consist in several small but precise changes in the JVM code. These changes are sufficient to provide the features proposed in this paper. As the implementation is relatively contained, and does not change core algorithms (such as the collection algorithms), we envision that it would be portable to other collectors very easily. We are currently preparing the code to send a patch proposal to the OpenJDK HotSpot project.

In the rest of this section, we describe the two main implementation challenges of our solution: i) how to implement the dynamic memory limit (`CurrentMaxMemory`), and ii) how to implement the periodic heap resizing checks. We clearly indicate whenever the implementation is different between the two supported collectors (PS and G1).

4.1 Dynamic Memory Limit

As discussed in Section 2.3, the JVM allows the user to specify, at launch time, a number of configuration parameters, one of which, the maximum memory limit (`MaxMemory`). This limit is static and therefore cannot be changed at runtime.

To implement and set up the dynamic memory limit (i.e., the value of `CurrentMaxMemory`), we create a new JVM runtime variable which can be set either at launch time using the JVM launch arguments or changed at runtime using an OpenJDK tool named `jstat`. Since the `CurrentMaxMemory` value must respect the invariant presented in Section 3.1, every time a new value is requested, the JVM executes the code presented in Algorithm 1.

Besides assigning new values to `CurrentMaxMemory` we also had to modify the allocation paths and heap resizing policies (in both G1 and PS) to respect the invariant check. For example, the JVM will fail to grow the heap if the resulting heap size is larger than the value defined in `CurrentMaxMemory` even if the new size is below the `MaxMemory` value.

4.2 Heap Resizing Checks

As discussed in Section 3.2, it is also necessary to reduce the heap size (`committed` memory) to return unused memory back to the host engine in a timely manner. To do so, the code presented in Algorithm 2 must be executed frequently.

To avoid excessive performance overhead, we piggy-back the heap resize checks in the main loop of the JVM control thread. This control thread runs in an infinite loop which is iterated nearly once every second. Inside the loop, several internal checks are performed, and internal maintenance tasks may be triggered (such as a GC cycle). We modified the control thread loop to also include the heap resizing check. This ensures that our resizing check is executed frequently and with a small performance overhead by utilizing the existing JVM control thread mechanism.

4.3 Integration with Existing Heap Resizing Policies

Whenever the heap resizing check returns true, meaning that the heap should be resized to return memory to the host engine, a heap resizing operation is triggered. Currently, this operation is implemented through a full GC cycle (we are working so that in future, a full GC can be avoided). The way a full GC cycle leads to a heap resize is, however, different in the two supported collectors (G1 and PS).

In G1, a full collection leads inevitably to several heap ergonomic checks that will determine if the heap should grow or shrink. The thresholds used for these checks are tunable through several heap launch time arguments. In other words, no changes are introduced into G1 heap sizing code and it suffices to trigger a full collection cycle in order for the heap size to be adjusted.

PS, however, employs a different adaptive sizing algorithm to adjust the heap size based on feedbacks from previously completed collections. PS sets two targets for each GC: i) pause time, and ii) throughput. The pause time target sets an upper bound for the GC pause time; the throughput target specifies the desired ratio of GC time and the total execution time. Based on these two targets, the adaptive sizing algorithm shrinks the heap on two occasions. First, if the GC pause time exceeds the pause time target, PS shrinks the heap until the target is met. Second, if the throughput target is met, i.e., the proportion of GC time in the total time is less than 1%, PS shrinks the heap to save memory. To avoid abrupt changes to the heap size and performance fluctuations, PS uses the moving average of the pause times of recent GCs in the adaptive sizing algorithm.

Unlike the G1 collector, which resizes the heap immediately after a full GC reclaims memory, the PS collector relies on the adaptive sizing algorithm to adjust the heap size. There are several challenges in shrinking the heap in PS. First, since the heap resizing is based on the moving average of recent GC times, a single GC triggered by the change of `CurrentMaxMemory` may not lead to a heap size change. Second, PS divides the heap into the young and old generations. Heap resizing involves adjusting the sizes of the two generations and carefully dealing with their boundary. To enable timely heap resizing in PS, we bypass the adaptive sizing algorithm whenever the heap resizing check (Algorithm 2) returns true and forces a heap resizing.

5 Evaluation

This section presents evaluation results for the solution previously described. The main goals of this evaluation are the following: i) show that it is possible to reduce the JVM heap size (committed memory), and thus reduce the instance memory by utilizing the proposed solution (see Section 5.2); ii) show that this reduction in the JVM memory footprint does not impose a significant performance overhead for applications (see Section 5.3); iii) show that the memory overhead (for holding large GC data structures) associated to having a very large `MaxMemory` limit is negligible (see Section 5.4);

Table 2. DaCapo Benchmarks

Benchmark	# Iters	CMaxMem	MaxOCMem	MinTmGCs
avrora	5	32 MB	16 MB	10 sec
fop	200	512 MB	32 MB	10 sec
h2	5	1024 MB	256 MB	10 sec
jython	5	128 MB	32 MB	10 sec
luindex	100	256 MB	32 MB	10 sec
pmd	10	256 MB	32 MB	10 sec
sunflow	5	128 MB	16 MB	10 sec
tradebeans	5	512 MB	128 MB	10 sec
xalan	5	64 MB	16 MB	10 sec

iv) show how much cloud users can save by allowing JVM applications to scale memory vertically (see Section 5.5).

In each experiment, we show results for both our implementations and their respective baseline implementations. For simplicity, plots are labeled as follows (from left to right): G1 (unmodified Garbage First collector); VG1 (Vertical Garbage First, modified version of Garbage First); PS (unmodified Parallel Scavenge collector); VPS (Vertical Parallel Scavenge, modified version of Parallel Scavenge).

5.1 Evaluation Environment

In order to simulate a real cloud environment, we prepared a container engine installation which was used to deploy JVM applications in containers. The physical node that runs the container engine is equipped with an Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz, 32 GB DDR4 of RAM, and an SSD drive. The host OS runs Linux 4.9 and the container engine runs Docker 17.12. Each container is configured to have a limit of memory usage of 1 GB, and two CPU cores.

To exercise our solution, we take advantage of the widely used and studied DaCapo benchmark suite [4] (version 9.12). Each benchmark execution is performed in a single container in complete isolation (i.e., no other applications running in the same container and host OS).

Table 2 presents the benchmarks and configurations used. The table presents, for each benchmark, the number of iterations used to produce results, and the values for the variables: `CurrentMaxMemory`, `MaxOverCommittedMemory`, and `MinTimeBetweenGCs`. Other DaCapo benchmarks (`batik`, `eclipse`, `lusearch`, `tomcat`, and `tradesoap`) could not be used as they do not run in OpenJDK 9.

In our experiments, each benchmark runs for a number of iterations, in addition to warm-up iterations (which are not accounted for the results). Most benchmarks run for 5 iterations (after the warm-up iterations), which is enough to extract reliable statistics regarding the execution. Benchmarks with very short iteration execution times run for more iterations (the shorter the execution time is, more iterations are needed). This is necessary because a single GC cycle might increase the time of a single iteration by a large factor.

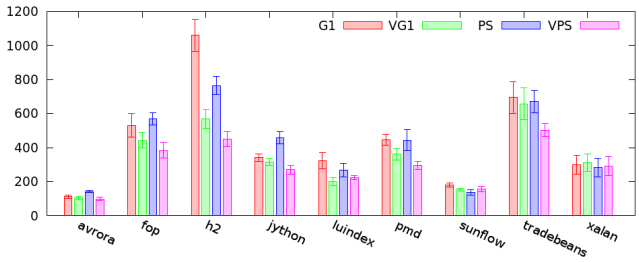

Figure 2. Container Memory Usage (MB)

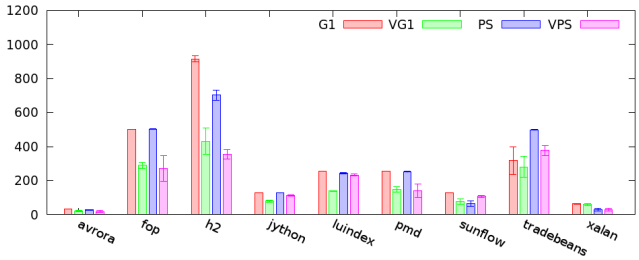

Figure 3. JVM Heap Size (MB)

We prepare each benchmark to run with different `CurrentMaxMemory` limits (heap size). Each limit is determined by running the same benchmark with different `CurrentMaxMemory` limits until the lowest limit with the highest throughput is found (i.e., we optimize for throughput and then try to reduce the footprint). Except in Section 5.4, all experiments are configured with `CurrentMaxMemory` equal to `MaxMemory`.

The `MaxOverCommittedMem` is set to either half or quarter of the current max heap size. We found that these values provide a good memory scalability while imposing a negligible throughput overhead (this tradeoff is discussed in Section 5.3). `MinTimeBetweenGCs` is set to 10 seconds, meaning that a heap sizing operation can not be started if a GC cycle ran less than 10 seconds ago. In a real scenario, this value would reflect the periodicity of the cloud provider's billing period (hourly, daily, etc).

5.2 Dynamic Memory Scalability

This section presents results on how much can the application memory footprint be reduced by employing the heap sizing strategy proposed in Section 3.2. This footprint change is presented from two perspectives: i) the container memory usage (see Figure 2), and ii) the JVM heap size or committed memory (see Figure 3). Both figures present the average and standard deviation values for their respective metric.

Looking at the container memory usage, it is possible to observe that most benchmarks greatly benefit from lower memory usage when VG1 or VPS are used (compared to G1 and PS respectively). Also from these results it is possible to conclude that the benefit is greater for benchmarks with

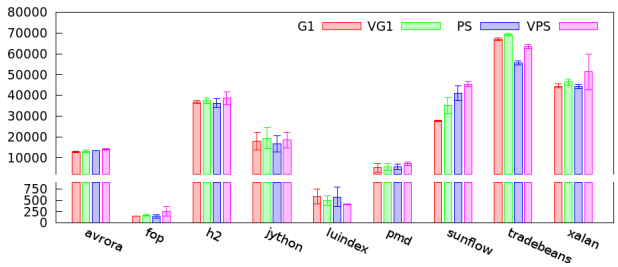

Figure 4. Execution Time (ms)

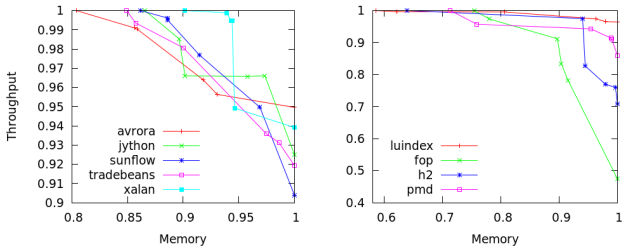

Figure 5. Throughput vs Memory Tradeoff

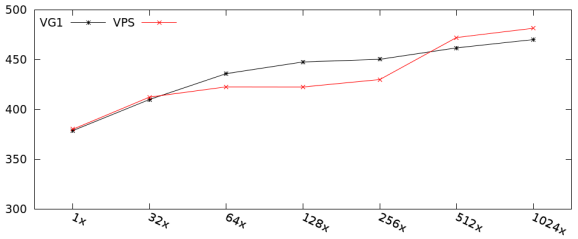

Figure 6. h2 Container Used Memory (MB) for Different Max Heap Limits

higher memory usage, i.e., there is more memory to save in applications which use more memory. Taking the h2 benchmark as an example, using VG1 or VPS instead of G1 or PS leads to 46.3% and 41.3% reduction in the container used memory (respectively). Another interesting fact is that both PS and VPS lead to smaller application memory footprint when compared to G1 and VG1, respectively. This is due to how PS is internally implemented.

The same conclusions taken from the container memory usage can also be drawn from the JVM heap size (presented in Figure 3). Both plots are highly correlated as the JVM heap size directly impacts the container memory usage. Using h2 as example, using VG1 or VPS instead of G1 or PS leads to 53.0 % and 49.6 % reduction in the JVM heap size.

5.3 Heap Resizing Performance Overhead

The reduction in the container used memory and JVM heap size comes from periodically checking if a heap resizing operation should be performed. This operation (currently implemented through a GC cycle) triggers periodic GC cycles which force the application to run with a smaller memory footprint. In this section we measure how much the throughput of the application is affected when our heap sizing approach is enforced.

Figure 4 presents the average and standard deviation for the execution time for each benchmark across G1, VG1, PS, and VPS. From the plot, it is possible to observe a slight increase in the execution time for both VG1 and VPS when compared to G1 and PS (respectively). Using h2 again as example, using VG1 or VPS instead of G1 or PS leads to a 2% and 6% execution time overhead respectively.

The memory footprint improvement and throughput overhead measured so far are directly related to the configuration used (see Table 2), in particular the value of the parameter `MaxOverCommittedMemory`. Figure 5 presents a VG1 throughput versus memory tradeoff to provide a better understanding of how much memory improvement can be achieved and at which cost in terms of throughput.

To build these plots, we ran each benchmark with different `MaxOverCommittedMemory` values: 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, and with no limit (i.e., equivalent to running G1). As we move `MaxOvercommittedMemory` to smaller values, the throughput decreases and the memory footprint is reduced.

Each plot compares the throughput and the container memory usage. Each axis is normalized to the best possible value (either the best throughput or the smallest memory footprint). For example, if a point is placed at 1.00 throughput and 0.80 memory, it means that the highest throughput is achieved when the memory footprint is 20 % higher than the smallest possible memory footprint.

Taking h2 as an example again, we can analyze the throughput evolution as we move towards smaller values of `MaxOverCommittedMemory` (i.e., moving from left to right). Considering the first two h2 points as (Throughput;Memory), we have (1.00;0.64) and (0.98;0.94). From these two points, it is possible to see that we can reduce the average memory utilization by 30% at a 2% throughput overhead.

In sum, from these plots, it is possible to perceive how the throughput of each benchmark behaves when a smaller value of `MaxOverCommittedMemory` is imposed. The interesting conclusion to take is that, for several benchmarks (mostly on the left-hand side of Figure 5), with less than 10% of throughput overhead, it is possible to reduce the memory footprint by up to 20%.

5.4 Internal Data Structures Overhead

As discussed in Section 3, setting the `MaxMemory` limit to a conservative very high value will force the JVM to setup larger GC internal data structures that need to be prepared to handle a large heap. This raises a potential problem as setting up a high `MaxMemory` value will result in larger GC internal data structures, which might require a lot of memory.

Figure 6 presents the container used memory for different values of `MaxMemory` for the `h2` benchmark. The value of `CurrentMaxMemory` is fixed across all runs, and is set to 1024MB. This experiment exercises values of `MaxMemory` starting at 1x the `CurrentMaxMemory` (1GB) until 1024x the `CurrentMaxMemory` (1TB).

From this experiment, it is possible to conclude that being conservative and setting `MaxMemory` to a very high value does not lead to an increased memory footprint. In `h2`, setting a heap max size 32x larger compared to the smallest memory footprint with highest throughout only adds 31.3MB to the container. In other words, increasing the `MaxMemory` by 32GB results in 31.3MB of increased container memory usage. We do not show the results for other benchmarks due to the lack of space. Nevertheless, the size of the GC internal data structures does not depend on the user application and thus, the trade-off between increased `MaxMemory` and extra data structure footprint is the same.

5.5 Real World Workload

For this final evaluation experiment, we perform an Amazon EC2 cost estimation (comparing the unmodified JVM to our solution) for a very common real-world workload (according to Jelastic logs). We prepared the following scenario based on real-world utilization in Jelastic cloud. We use a Tomcat webserver container with 4, 8, 16, and 32 GBs of RAM. The server is mostly accessed during the day. At night (approximately for 8 hours), there is almost no access to the server. User sessions (which occupy most of the memory) timeout after some time (10 minutes, in our experiment). As there is no user activity during the night, no GC is triggered and thus, the heap stays at full size all the time. When using the solution proposed in this work, the container usage drops to approximately 100 MB during the night. Figure 7 presents a plot showing a Tomcat webserver with 8 GBs of RAM for a 24 hour period. As described, VG1 (the proposed solution) is able to reduce the container memory to approximately 100 MBs for a period of 8 hours (10 to 18 hours in Figure 7) while G1 keeps full memory usage all the time.

We now calculate how much it would cost to deploy this workload on Amazon EC2 (assuming that Amazon EC2 supports resource elasticity, e.g., one could change the instance resources at runtime). If that is the case, we could host our Tomcat server during the day using an instance with more memory than the instance used during the night. In Table 3, we show the projected monthly cost for running Tomcat in an unmodified JVM, and in a JVM running our heap sizing approach (VJVM). We show results for Tomcat servers with 4 to 32 GBs of RAM. By analyzing the results in Table 3, it is possible to achieve, for this particular workload a cost reduction of up to 33%.

From the cloud provider's point of view, there are also benefits. Since the Tomcat server is now running in a much smaller instance (up to 64x smaller, for the 32 GB instance),

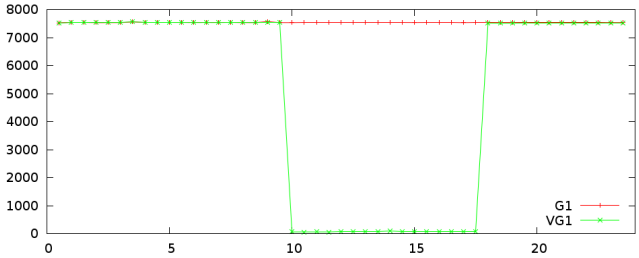

Figure 7. Tomcat Memory Usage (MB) during 24 hours

Table 3. Monthly Amazon EC2 Cost (USA Ohio Data Center)

Approach	Daily	Nightly	Total	Saving
4GB-JVM	23.01 $	11.53 $	34.00 $	
4GB-VJVM	23.01 $	1.44 $	24.44 $	29.40%
8GB-JVM	46.03 $	23.01 $	69.04 $	
8GB-VJVM	46.03 $	1.44 $	47.47 $	31.00%
16GB-JVM	92.06 $	46.03 $	138.00 $	
16GB-VJVM	92.06 $	1.44 $	93.50 $	32.60%
32GB-JVM	184.12 $	92.06 $	276.00 $	
32GB-VJVM	184.12 $	1.44 $	185.00 $	33.00%

and since memory is the limiting factor for oversubscribing [12], it is possible to collocate instances and reduce up to 64x the amount of hardware used to run the same instances.

6 Related Work

Applications have different memory requirements and even a single application often deals with different task sizes resulting in different memory demands throughout its execution [18]. The challenge is then to assign the application with the correct amount of memory such that: i) it is not penalized in terms of throughput due to lack of available memory, and ii) it does not lead to resource waste.

Other researchers have looked into the problem of determining the correct amount of memory to assign to a particular JVM application (or set of applications) from two different perspectives: i) assign memory to an instance, or ii) resize the JVM heap. In the next sections, both perspectives are discussed and compared to our approach.

6.1 Memory Balancing in Virtualized Environments

As discussed in Section 2.1, both virtual machines and containers support dynamic changes to the memory assigned to them. However, determining the real memory requirements at runtime is still an open problem.

Waldspurger et al. [14] propose a page sampling approach to infer the instance memory utilization. During a sampling interval, accesses to a set of random pages are monitored and, by the end of the sampling period, the page utilization is used as an approximation for the global memory utilization.

Zhou et al. [19] propose the use of a page miss ratio curve to dynamically track the working set size. This curve can be built using data from special hardware or statistics from the OS; the former tracks the miss ratio curve for the entire system, while the latter tracks it for individual applications. Jones et al. [11] infer memory pressure and determines the amount of extra memory required by an instance by monitoring disk I/O and inferring major page faults. Lu et al. [13] propose an LRU based miss ratio curve to estimate the memory requirements for each instance. Using their solution, there is a pool of memory which can be used to assign different amounts of memory to different instances. Memory accesses to the pool are tracked by the host engine. The work by Zhao et al. [18] dynamically adapts the memory assigned to each system VM by using an LRU predictor. To build such a predictor, the authors intercept memory accesses to a subset of memory pages. Finally, Caballer et al. [7] present a solution based on a memory over provisioning percentage. In this solution, memory usage is probed periodically and the amount of memory assigned to each instance is increased or decreased in order to allow a memory over provisioning percentage all the time.

6.2 Heap Sizing

Previous attempts to determine the optimal heap size have used techniques in which the size of the heap can be controlled in order to: i) allow the application to achieve target performance goals (such as throughput and/or pause times), and ii) avoid resource (memory) waste. This heap sizing problem can be seen as a trade-off between having a very large heap, which might trigger paging (due to limited memory in the host), and having a very small heap which decreases throughput due to an increased GC overhead. This trade-off is often modeled using a 'sweet-spot' curve [6, 15].

Although heap sizing is a well-studied problem, researchers are still looking for better approaches/trade-offs for this problem. Brecht et al. [6] propose a heuristic-based heap sizing mechanism for the Boehm collector [5]. Using this sizing mechanism the heap is allowed to grow by different amounts, depending on its current size and on a set of threshold values. The goal is to avoid both GC overhead (due to a small heap) and paging (due to a large heap). The heap size cannot, however, be reduced due to collector limitations [5].

Yang et al. [16, 17] take advantage of reuse distance histograms and a simple linear model of the required heap. In their approach, a JVM communicates with a Virtual Memory Manager (which is running in a modified OS) in order to acquire information about its own working set size, and the OS's available memory. With this information, the collector is able to make better decisions in order to avoid paging.

The Isla Vista [8] is a feedback-directed heap resizing mechanism that avoids GC-induced paging, using information from the OS. Costly GCs are avoided by increasing the heap size (while physical memory is available). When allocation stalls are detected, the heap size shrinks aggressively.

Hertz et al. [10] use a region of shared memory to allow executing instances to gather information on page faults and resident set size. This information is then used to coordinate collections and select the correct heap sizes. The cooperative aspects of the memory manager are encoded using a fixed set of rules, known as Poor Richard's memory manager. In White et al. [15] it is shown that control theory could be applied to model the heap sizing problem. The developed controller monitors short-term GC overhead and adjusts the heap size in order to achieve performance goals.

6.3 Discussion

When comparing previous approaches (determining the correct memory needs for an instance or a JVM), it is important to note that the proposed solution in this work is not meant to replace them. Instead, our solution attacks problems that are preventing both host engines and heap sizing policies from cooperating. Works similar to ours include solutions that try to make the host engines cooperate with the heap sizing engines [9, 16, 17]. However, such works require modifications to the host engines, something that is very hard to request in current cloud environments.

In sum, current JVM applications running on containers or virtual machines are not able to scale their memory requirements due to the lack of mechanisms inside the JVM that would allow the JVM and the host engine to exchange memory as required.

7 Conclusions

This work proposes a new heap sizing approach that allows the OpenJDK HotSpot JVM to vertically scale its memory resources. In particular, the proposed solution i) allows the cloud user to dynamically adjust the maximum memory limit for the JVM (without requiring the JVM to restart), and ii) timely releases unused memory back to the host engine.

We implement and evaluate our approach using the OpenJDK HotSpot JVM 9. Experiments include the DaCapo benchmark suite and the Tomcat webserver exercised with real world based workloads. Results are very promising as the footprint of most applications can be greatly reduced with a very small throughput overhead. We are preparing the code to send an OpenJDK HotSpot patch.[3]

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013, and through the FCT scholarships SFRH /BD/103745/2014 and SFRH/BSAB/135197/2017.

[3]The code and patch are available at github.com/jelastic/openjdk

References

[1] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. 2017. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Transactions on Services Computing* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSC.2017.2711009

[2] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, Berkeley, CA, USA, 45–58. http://dl.acm.org/citation.cfm?id=296806.296810

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[5] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820.

[6] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. 2006. Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications. *ACM Trans. Program. Lang. Syst.* 28, 5 (Sept. 2006), 908–941. https://doi.org/10.1145/1152649.1152652

[7] Miguel Caballer, Ignacio Blanquer, Germán Moltó, and Carlos de Alfonso. 2015. Dynamic Management of Virtual Infrastructures. *Journal of Grid Computing* 13, 1 (01 Mar 2015), 53–70. https://doi.org/10.1007/s10723-014-9296-5

[8] Chris Grzegorczyk, Sunil Soman, Chandra Krintz, and Rich Wolski. 2007. Isla Vista Heap Sizing: Using Feedback to Avoid Paging. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 325–340. https://doi.org/10.1109/CGO.2007.20

[9] Matthew Hertz, Yi Feng, and Emery D. Berger. 2005. Garbage Collection Without Paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 143–153. https://doi.org/10.1145/1065010.1065028

[10] Matthew Hertz, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Chen Ding, Xiaoming Gu, and Jonathan E. Bard. 2011. Waste Not, Want Not: Resource-based Garbage Collection in a Shared Environment. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/1993478.1993487

[11] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2006. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 14–24. https://doi.org/10.1145/1168857.1168861

[12] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. 2014. Group-based Memory Oversubscription for Virtualized Clouds. *J. Parallel Distrib. Comput.* 74, 4 (April 2014), 2241–2256. https://doi.org/10.1016/j.jpdc.2014.01.001

[13] Pin Lu and Kai Shen. 2007. Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association, Berkeley, CA, USA, Article 3, 15 pages. http://dl.acm.org/citation.cfm?id=1364385.1364388

[14] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 181–194. https://doi.org/10.1145/844128.844146

[15] David R. White, Jeremy Singer, Jonathan M. Aitken, and Richard E. Jones. 2013. Control Theory for Principled Heap Sizing. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2464157.2466481

[16] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2006. CRAMM: Virtual Memory Support for Garbage-collected Applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 103–116. http://dl.acm.org/citation.cfm?id=1298455.1298466

[17] Ting Yang, Matthew Hertz, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2004. Automatic Heap Sizing: Taking Real Memory into Account. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/1029873.1029881

[18] Weiming Zhao and Zhenlin Wang. 2009. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/1508293.1508297

[19] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/1024393.1024415