ALMA – GC-assisted JVM Live Migration for Java Server Applications

Rodrigo Bruno, Paulo Ferreira
INESC-ID / Instituto Superior Técnico, University of Lisbon
{rodrigo.bruno,paulo.ferreira}@inesc-id.pt

ABSTRACT

Live migration of Java Virtual Machines (JVMs) consumes significant amounts of time and resources, imposing relevant application performance overhead. This problem is specially hard when memory modified by applications changes faster than it can be transferred through the network (to a remote host). Current solutions to this problem resort to several techniques which depend on high-speed networks and application throttling, require lots of CPU time to compress memory, or need explicit assistance from the application. We propose a novel approach, Garbage Collector (GC) assisted JVM Live Migration for Java Server Applications (ALMA). ALMA makes a snapshot to be migrated containing a minimal amount of application state, by taking into account the amount of reachable memory (i.e. live data) detected by the GC. The main novelty of ALMA is the following: ALMA analyzes the JVM heap looking for regions in which a collection phase is advantageous w.r.t. the network bandwidth available (i.e. it pays to collect because a significant amount of memory will not be part of the snapshot). ALMA is implemented on OpenJDK 8 and extends CRIU (a Linux disk-based process checkpoint/restore tool) to support process live migration over the network. We evaluate ALMA using well-known JVM performance benchmarks (SPECjvm2008 and DaCapo), and by comparing it to other previous approaches. ALMA shows very good performance results.

CCS Concepts

•Networks → Cloud computing; •Software and its engineering → Virtual machines; Garbage collection; Process management;

Keywords

Virtual Machine Migration; Garbage Collection; Garbage First; Java Heap; OpenJDK; JVM; CRIU

Middleware'16, December 12-16, 2016, Trento, Italy
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4300-8/16/12...$15.00
DOI: http://dx.doi.org/10.1145/2988336.2988341

1. INTRODUCTION

Live migration of a running Java application entails the full transfer of its state to a remote location, where it is resumed, with minimal downtime. Live migration is useful for a number of situations, for example: replacing a crashed service [34] (fault tolerance), replicating an application for work distribution [42] (load balancing), application fast start-up [26], co-locating services to reduce resource usage [9, 35] (power saving), live service updates [19].

In this work we are interested in improving the live migration of Java server applications which do not rely on other local processes (running on the same node). For example, if two processes in the local node depend on each other, one cannot be migrated if the other isn't; in this case, a full system virtual machine[1] migration would be necessary (to migrate both processes at the same time). However, if the application to be migrated does not rely on other local processes to run, we can avoid a full system-VM migration (which migrates all the Operating System state and other processes that might be running) and, instead, perform a JVM migration (single process migration), which is a faster and less resource demanding alternative to migrate the application.

The efficiency of live migration can be measured in three metrics: i) **total migration time**, i.e., the amount of time since the migration starts until it finishes; ii) **application throughput**, i.e, the difference between the normal application throughput and the throughput experienced including a migration (note that application downtime is not the only factor affecting the application throughput, other factors such as application throttling or some application slowdown after migration also affect the throughput); iii) other **resources overhead** (e.g., network bandwidth, CPU). We target these three metrics with out system called ALMA, i.e. we aim at achieving Java application live migration with: i) minimal total migration time, ii) minimal application throughput impact, iii) minimal resource overhead (mainly CPU and network bandwidth). Additionally, we also want to avoid requiring programmer intervention/help for the migration, special hardware and modifications to the Operating System as all of these factors impose difficult obstacles to deploying and using the system.

To fulfill such requirements, the migration engine must be effective and efficient in several ways. To ensure a low total migration time and low resource overhead, the migration

[1]In this paper we differentiate a system-VM (e.g. Xen-based or similar) from the Java Virtual Machine (JVM) using the terms system-VM and JVM.

engine must minimize the data to transmit; this, it avoids transmitting data that is not necessary or that can be generated at the destination node. To achieve low application overhead (i.e., low throughput impact), the migration engine must: i) efficiently identify the process working set, and ii) avoid forcing the application to slowdown before or after the migration.

Keeping an overall low performance overhead is particularly challenging because the network bandwidth, which is typically used to transfer the JVM state to a remote computer, is very slow compared to the memory bandwidth within a single system [29]. Allocation intensive applications (i.e., applications with high memory demands) are even harder to migrate since more memory pages are constantly being dirtied.

A common approach to this problem is to take an initial snapshot of the underlying system-VM state while application is running (pre-copy [44, 7]). After the snapshot is transferred, the system-VM is suspended and an incremental snapshot is taken (containing only the differences since the last snapshot). This second snapshot is transferred and the system-VM is resumed at the destination host. Previous work try to improve this approach by using several techniques such as: application throttling [7] and use of high-speed networks [22], compress system-VM state [23], using application assistance or looking into the application state to reduce the amount of state to transfer [21, 19]. However, these solutions require special hardware, or impose high application overhead, or fail to completely determine the exact state of the application state (i.e., determine which memory pages are actually required to transfer) ending up transferring more data than needed, leading to longer migration downtimes.

Using GC to reduce the amount of data to transfer has been proposed in previous works ([21, 26]). However, they not take full advantage of the GC information available when performing the migration and therefore, force a heap collection even if not necessary; this adds more downtime to the application without significantly reducing the size of the snapshot.

In this work we investigate how much improvement can be attained by taking advantage of knowledge about the reachability of heap objects. This technique is not exclusive as other techniques can be combined to boost performance (for example snapshot compression).

In ALMA we are focused on migrating applications running on top of JVMs. We assume that a system-VM is already available at the destination node, ready to receive the JVM. By migrating only the JVM, we avoid all the Operating System state and other processes that might be running; this contributes to a small total migration time and low resource usage (since all snapshots only contain information about the JVM and application).

To minimize the size of each snapshot (that will be migrated), we force a migration-aware garbage collection before taking the snapshot of the process state. In ALMA we use the Garbage First (G1) GC, a region-based GC targeted to be used with large heaps while maintaining low pause times.[2]

We augment G1 by introducing a new GC policy to select which regions to collect when collecting the heap. The goal of the new policy is to select for collection only those regions whose amount of dead bytes divided by the GC time (for the particular region only) is lower than the available network bandwidth (see Section 3 for more details). This migration-aware GC policy introduces no application overhead since it uses only internal information already being gathered by the G1 collector (see Section 2 for details). By forcing a migration-aware GC before creating each application snapshot, we minimize the amount of dead data in the snapshot, thus substantially minimizing its size.

To implement ALMA we extended CRIU[3] (a checkpoint and restore tool for Linux processes that runs in userspace) with a Migration Controller; this controller interacts with the JVM and forwards its snapshots to the destination system-VM. The migration-aware GC policy is implemented inside the G1 GC in the Open Java Development Kit 8 (OpenJDK 8).[4] All the ALMA code is publicly available this allowing others to reuse the ALMA and perform experiments with it.[5]

We evaluate ALMA by using well-known benchmark suites based on real world Java applications (SPECjvm-2008 [39] and DaCapo [3]), and by comparing it to: i) an application-assisted approach called JAVMM [21] (a similar system we could find on the literature), ii) CRIU (writes snapshots to disk and then migrates them by means of NFS), and iii) ALMA-PS (our solution configured to work similarly to JAVMM [21], more details in Section 5). The evaluation results show that ALMA is able to deliver low total migration time, low resource overhead, and low application throughput impact. Compared to the results obtained in JAVMM [21] and CRIU, ALMA reduces the total migration time by 5.69 and 2.57 times, the used network bandwidth by 4.42 and 2.86 times, and the application downtime by 1.125 and 3.58 times, respectively.

In short, this work presents several contributions. First, it provides fast live migration for Java server applications that do not require full system-VM migration. For these applications, migrating only the JVM is sufficient. Second, it describes a G1 GC migration-aware policy implemented on OpenJDK 8. This policy uses internal information about each heap region to determine the best set of regions to collect based on the available network bandwidth; Third, it proposes JVM heap collection before each snapshot. To the best of our knowledge, this is the first system to employ a collection at each snapshot. This ensures that no snapshot contains more data than necessary (according to our migration-aware policy). Fourth, it provides a JVM live-migration tool that runs in user space implemented on top of CRIU which increases, in multiple ways, the performance of live JVM migration, compared to previous approaches. And finally, it presents a detailed performance evaluation comparing our solution to Hou [21], and CRIU.

The remaining of this document is organized as follows. Section 2 gives some background on the OpenJDK's heap, the G1 GC approach, and CRIU (a checkpoint and restore tool for Linux processes). Section 3 presents the design of

[2]Garbage First Garbage Collector is a region-based GC. It is the most recent GC implementation and it will be the next default collector in OpenJDK's HotSpot JVM.

[3]CRIU is an open-source project available at criu.org

[4]The Open Java Development Kit is available at openjdk.java.net

[5]The code is available at: https://github.com/rodrigo-bruno/ALMA-JMigration. Part of the developed code is being integrated in the main CRIU repository.

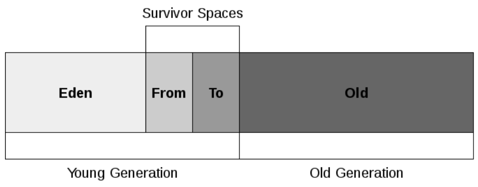


Figure 1: Parallel Scavenge GC Heap

ALMA followed by some implementation details on Section 4. In Section 5, we evaluate ALMA and analyze the results obtained. We conclude with an analysis of the current state-of-art (Section 6), and some conclusions (Section 7).

2. BACKGROUND

This section presents several basic concepts regarding the Java heap and the corresponding GCs; this is relevant for the next sections describing ALMA. Additionally, we also discuss CRIU, a Linux process checkpoint and restore tool since it is the basis for the implementation of the Migration Controller (presented in Section 4.2).

The Java heap is a continuous segment of memory where all application objects are allocated. In order to remove dead (unreachable) objects and free space for new ones, a GC is used. Different garbage collectors take different approaches to collect death objects and, most of the times, different collectors also organize the heap differently [24].

We target two important GC algorithms (currently available in the OpenJDK): i) Parallel Scavenge (PS), a simple generational GC which has two distinct memory spaces (generations), and ii) G1, a more recent generational GC which divides the heap in equally sized regions that can belong to any generation. Although ALMA is designed using G1 GC, we discuss both collectors (G1 and PS) to understand their key characteristics as they are both extensively discussed in in the Evaluation Section.

2.1 Parallel Scavenge

The idea behind generational GC algorithms [31] (based on the intuition that young objects die young) is not new. Therefore, through time, many generational algorithms have been developed ([13, 1, 38, 46, 14, 2] just to name a few). Although most of them share the basic principle of splitting the heap in young and old generations, the way to collect each generation differs a lot. We consider the OpenJDK 8's Parallel Scavenge collector because: i) it is the current default collector (and therefore it is probably one of the most used GC implementations), and ii) it was used in previous work [21].

This specific collector (PS for short) uses two different algorithms [17]: a copy collector [6] for the young generation, and a mark-and-sweep collector [33] for the old generation (see the PS heap layout in Figure 1).[6] Both generations are collected using multiple threads but are not concurrent with the mutator, i.e., both require stopping the application (forcing stop-the-world pauses).

[6]In generational GCs (and therefore in PS and G1, the heap is divided in two generations: young and old. The young generation is where all recently allocated objects live. As objects survive collections, older objects are promoted into the old generation.

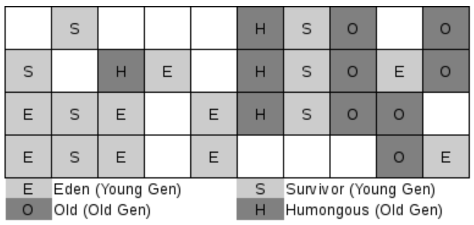


Figure 2: Garbage First GC Heap
(each square represents a region)

The young generation is divided into three areas (see Figure 1): *Eden*, *From*, and *To* spaces. The *Eden* is where all new objects are allocated. Once it is full, a minor collection takes place: when such collection starts, all live data inside the *Eden* space is copied to *From* space and all live data inside *To* space is copied to the old generation. Full GCs (i.e., all the heap is collected) are triggered when the heap is almost full and typically entail an extra heap compaction step.

This collector aims at achieving high throughput. However, this comes at the cost of forcing the mutator to stop while the garbage collector is performing a minor or a full collection. Increasing the size of the young generation will result in less frequent GCs but will increase the per-GC pause as the pause time is proportional to the number of live objects (that need to be copied away from *Eden* space).

2.2 Garbage First

Garbage First GC [12] is generational [31] and has a specific heap organization (see Figure 2). A G1 heap is divided into equally sized regions (the default region size for the OpenJDK 8 is 1024 Kilobytes) which can be in one of the following states: Eden (newly allocated objects), Survivor (objects moved from Eden in the latest collection), Old (objects that survived at least two collections), and Humongous (large objects). In order to maintain enough free space for new regions (necessary to allocate new objects when current young regions are full), periodic GCs are performed.

Similarly to other generational GCs, the G1 provides two types of collections: i) young collections, in which only young (eden and survivor) regions are selected to be collected, and ii) mixed collection, in which both young and old (including humongous) regions can be selected for collection. Both types of collections are triggered by different conditions (which are not relevant for now). The important idea is that all the live data in the set of regions selected for collection is copied to one or more regions. This results in more compacted heap regions.

Since G1 is thought for collecting large heaps while maintaining a low GC pause time, it must control how many regions will be collected at each collection. To do so, it relies on data gathered by the concurrent marking threads, which traverse the heap and, for each region, produce statistic information relevant for a possible collection of that region (e.g., predicted time to collect, number of reclaimable objects, etc). With this information, more particularly the time it will take to collect a particular region, the GC will select a set of regions, named *Collection Set*, to collect while keeping the GC pause lower than an user defined threshold. This particular metric (time to collect a region) is estimated based on several factors: number of live objects in the region, number of cached reference updates (dirty cards), previous collections, etc.

In ALMA, we take advantage of the information gathered by the concurrent marking threads to select an optimized *Collection Set*, according to the available network bandwidth (see Section 3.2). Since we are harnessing information that is already there (for GC purposes) we incur in no extra application overhead.

Note that this information (time necessary to collect each region, reclaimable space per region) does not exist in PS because it uses a heap with two long generations (as illustrated in Figure 1) as opposed to G1 which uses a heap with multiple regions (see Figure 2).

Regardless of the information provided by the collector, another key important reason for using G1 in ALMA is that G1 is being prepared to be the default GC for the next versions of the OpenJDK.

2.3 CRIU

CRIU [25] is a checkpoint and restore tool for Linux. Using CRIU, it is possible to freeze a process and checkpoint it to local disk as a collection of files. One can, later, use this collection of files (snapshot) to restore the application in the point it was frozen. CRIU is implemented in user space and not in the Linux kernel.

CRIU supports snapshoting processes and subprocesses, memory-mapped files, shared memory, open files, pipes, FIFOs, unix domain sockets, network sockets, signals, and more are still being implemented (the system is still under development). Currently, it is mostly used to support container [16] live migration.

This tool is used by ALMA as basis for the implementation of the Migration Controller (described in Section 3.1). To enable Java application live migrations, ALMA extends CRIU to allow snapshot data to be transferred to the destination site (instead of being saved to local disk). By default, CRIU supports remote migration by using an NFS share to send the snapshot files. More details on how ALMA extends CRIU are presented in Section 4.2.

3. JVM LIVE MIGRATION

In this section we start by giving a small introduction to the migration workflow and how ALMA minimizes the amount of data to transmit during the migration. Then, we describe ALMA's architecture, how it selects regions to collect before a migration, the migration worklow, and finally, a set of optimizations.

The JVM live migration uses the following workflow (this workflow is described in further detail in Section 3.3): i) the source site takes a snapshot of the JVM, and sends it to the destination site; ii) upon reception at the destination site, the source site stops the application and takes an incremental snapshot of the JVM and sends it to the destination site. Although the algorithm described in Section 3.3 works with any number of incremental snapshots, ALMAis implemented to use only one incremental snapshot (after the initial snapshot). This is discussed in greater detail in Section 3.3.

To reduce the amount of data to transfer when performing a JVM live migration while keeping a low overhead on the application throughput, ALMA analyzes the heap to discover heap regions with a *GC Rate* (amount of data that can be collected in some amount of time) that is superior to the network bandwidth; such regions will be collected to reduce their size.

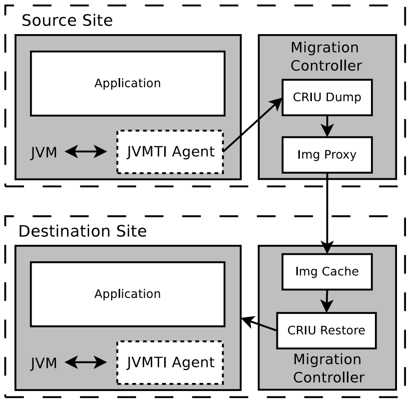


Figure 3: ALMA architecture.

3.1 Architecture

ALMA is composed by two components, each one used on both source and destination nodes/sites (see Figure 3): Migration Controller, and JVMTI agent (described below). Both destination and source sites are represented using dashed lines. Each process is represented with a gray background. JVMTI agents are represented by dotted lines.

The Migration Controller is responsible for: i) communicating with the local JVM at the source site to inform that a migration is being prepared (this will trigger the heap analysis and collection of the chosen regions); ii) looking into the local JVM process to save all the necessary information (in the source site) for the process to resume at the destination site (this includes page mappings, open files, threads, etc); note that apart from the first snapshot, only the differentials are transferred between the source and the destination sites; iii) transfer all the gathered process state data to the destination site; iv) bootstraping the process at the destination site, using the collected information by the Migration Controller at the source site. More details on how the Migration Controller is implemented (including a description of both Image Proxy, Image Cache, and CRIU) can be found in Section 4.2.

The JVM was modified to contain a migration aware G1 GC policy. This policy is used, when a migration starts, to determine the set of regions to consider for collection (more details in the next section). Note that we do not change or require any application-specific code. Only the JVM code is modified.[7]

To facilitate the communication between the Migration Controller and the JVM, we use a JVMTI agent, a simple pluggable component that accesses the internal JVM state[8]. This agent is responsible for: i) receiving requests from the Migration Controller to prepare the heap for a snapshot, (e.g., request to start a migration-aware GC), and ii) enumerating heap ranges of unused memory (that will be used to reduce the size of the snapshot, as described in Section 3.3).

[7]As discussed in Section 4.1, the changes to the JVM code are very small and therefore, are easily ported to other JVMs using a patch, for example.

[8]The JVMTI documentation is accessible at docs.oracle.com/javase/8/docs/technotes/guides/jvmti/

3.2 Heap Region Analysis

In order to reduce the amount of data to transfer, ALMA looks into the JVM heap for memory locations which are no longer reachable, i.e., garbage (thus containing only dead objects). To identify dead objects, one must scan/trace the entire heap and mark live objects, leaving dead objects unmarked (please note that we are focused in tracing collection [33] rather than reference counting [8] collection). This is a difficult task and many tracing GC implementations strive to reduce its negative effect on the performance of the application. Hence, we do not want to impose an extra overhead by using our own marking mechanism. Therefore, we rely on the marking operations performed by the G1 GC to analyze the heap, i.e., we neither modify the G1 GC marking operations to collect more data nor introduce new GC data structures.

As already said, the G1 GC (discussed in Section 2.2), periodically marks the heap and produces several metrics per heap region (resulting from the marking heap traversal) that allows ALMA to draw relevant conclusions leading to a minimal snapshot size. Two of the most important metrics are the following: i) an estimate of the amount of space the GC would be able to reclaim if a particular region is collected, and ii) an estimate of the time needed to collect a particular region.

With these estimates, ALMA decides, for each heap region, either to collect it, i.e., moving all live data to another region, or to avoid collecting it and thus not paying the time to do so. The set of regions selected for collection is called *Collection Set* (*CS* for short).

Thus, the total amount of heap data to transfer (i.e., to be included in the snapshot) is defined as the sum of the used space (i.e., allocated space, which might include reachable and unreachable data) of each region minus the reclaimable space (i.e., dead objects) in the regions in *CS* (see Eq. 1).

$$Data = \sum_{Heap} used(r) - \sum_{CS} dead(r) \tag{1}$$

Collecting a set of regions has a cost (time), which is defined in Eq. 2 as the sum of the cost of collecting each region in *CS*.

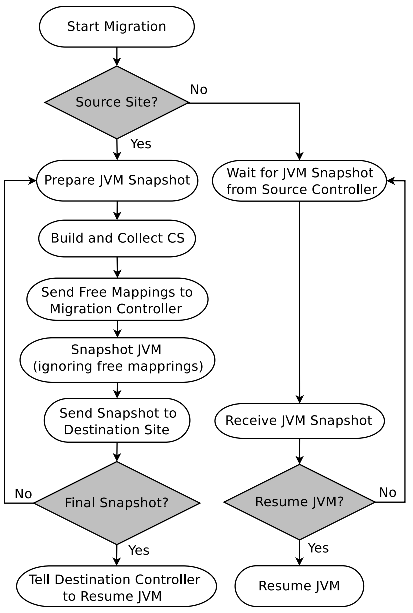
$$GCCost = \sum_{CS} cost(r) \tag{2}$$

We can now define the migration cost (in time) for migrating all heap regions (after each region r in *CS* has been collected) as the amount of data to transfer divided by the network bandwidth (which will be used to transfer the JVM) plus the cost of collecting the *CS* (see Eq. 3).

$$MigrationCost = \frac{Data}{NetBandwidth} + GCCost \tag{3}$$

Taking into account that we want to minimize the migration cost by properly selecting regions for the *CS*, we must minimize Eq. 3. In other words, we need to maximize the amount of reclaimable space and minimize the cost of collecting it. Hence, we define the ratio *GCRate* (see Eq. 4) as the amount of data reclaimed per amount of time for a region *r*.

$$GCRate(r) = \frac{dead(r)}{cost(r)} \tag{4}$$



Start Migration
Source Site?
No
Yes
Prepare JVM Snapshot
Build and Collect CS
Send Free Mappings to Migration Controller
Snapshot JVM (ignoring free mapprings)
Send Snapshot to Destination Site
Final Snapshot?
No
Yes
Tell Destination Controller to Resume JVM
Wait for JVM Snapshot from Source Controller
Receive JVM Snapshot
Resume JVM?
No
Yes
Resume JVM

Figure 4: ALMA's Migration Controller Workflow

With *GCRate* defined, we can estimate, for each region in the JVM heap, the *GCRate* and make sure that each region which has a *GCRate* superior to the network bandwidth is added to the *CS*. In other words, ALMA selects the regions that can have their size reduced and, as a result, transmitted faster than if that same region with its original size is transmitted. Thus, the *CS* is constructed as defined in Eq. 5: all regions whose *GCRate* is greater than the *NetBandwidth* are selected for collection.

$$CS = \{\forall r : GCRate(r) > NetBandwidth\} \tag{5}$$

3.3 Migration Workflow

Having explained how the heap is analyzed and prepared for migration, we now describe the live migration workflow; it starts when the migration request is issued, and finishes when the JVM is resumed at the destination site.

The flowchart in Figure 4 represents this process. To start, we launch two instances of the Migration Controller (`Start Migration`), one at each site. The controller spawned at the source site is then responsible for asking the JVM to prepare for a migration (`Prepare JVM Snapshot`). This request triggers a heap analysis, which results in the construction of the *CS*, which is then collected (`Build and Collect CS`). The request is then answered with a list of virtual memory ranges that contain no live data (`Send Free Mappings to Migration Controller`). Note that these virtual memory ranges can be as large as a full heap region, but can also be smaller. This ensures that only live data is transmitted and all other memory is skipped.

Also note that we guarantee that the virtual memory ranges marked as containing only dead objects are consistent with the real application state. ALMA does this by analyzing the heap memory and taking the process snapshot while the JVM is still inside the last stop-the-world pause after collecting the *CS*.

The next step (still at the source site) is to snapshot the JVM (`Snapshot JVM (ignoring free mappings)`). In this step, the Migration Controller looks into the process state and takes a snapshot of its memory, which is then forwarded to the destination site Migration Controller (`Send Snapshot to Destination Site`). This snapshot is incremental w.r.t. the previous one (except if this is the first snapshot).

Subsequent snapshots take the same approach until upon the last snapshot; then, the Migration Controller at the source site notifies the controller at the destination site to resume the JVM with all the state already forwarded (`Tell Destination Controller to Resume JVM` on the source site and `Wait for JVM Snapshot from Source Controller` at the destination site). At the destination site, the Migration Controller simply receives application snapshots (`Receive JVM Snapshot`), which are kept in memory, and waits for the resume JVM request (`Resume JVM`). Upon reception, it rebuilds the JVM and the process resumes.

This algorithm works with any number of snapshots. However, ALMA is configured by default to perform only two snapshots: one initial snapshot when the migration starts, and a second one (incremental with regards to the first one) when the initial snapshot arrives at the destination site. We found that having more than two snapshots does not reduce the application downtime (at least for the applications used in Section 5). Limiting ALMA to only two snapshots decreases the network bandwidth usage, and the total migration time. In addition, it turns migration more predictable, i.e., the Migration Controller does not take an arbitrary number of snapshots that will result in unpredictable total migration time and network bandwidth usage.

3.4 Optimizations

In order to improve the efficiency of the migration engine, ALMA employs several techniques to minimize the snapshot size and reduce the application overhead. For the rest of this section, we explore these optimizations: i) avoiding unnecessary collections when GCs triggered by the application are frequent; ii) avoid collecting regions included in previous snapshots to avoid increasing the size of the differential snapshot.

3.4.1 Avoid Unnecessary Collections

Depending on the mutator memory consumption rate, more or less GCs will be triggered. Applications that allocate memory very fast will most likely end up being collected much more often than applications that allocate much less memory.

We can take advantage of this fact in two ways. First, applications that allocate lots of memory will trigger GCs very often and ALMA can take advantage of these GCs to start a migration. In other words, instead of forcing a GC, ALMA can simply wait for the next application-triggered GC to start the snapshot cycle or start a forced GC after an used defined migration-timeout. Second, applications that allocate less memory will take longer to trigger a GC and will probably hit the migration-timeout most of the time. However, this is not a problem since these applications take longer to dirty memory and the migration engine can easily catch up with the memory changes.

3.4.2 Avoid GCs between Snapshots

Since G1 behaves just like a per-region copy collector (i.e., it copies the live content of one region to another upon collection), memory might get dirtied by the collector. This is particularly bad if the collector ends up copying live data around the heap because it breaks the benefits of using incremental snapshots.

To deal with this issue, ALMA prevents regions that had live data in the previous snapshot to be collected. By doing this, we prevent memory that was not filtered as garbage in the previous snapshot from being copied by the GC (this would create unnecessary incremental modifications between the previous and the next snapshot). Obviously, if the heap gets nearly full, we let the GC collect any regions. However, at this point, it probably means that most of the heap is dirtied anyway.

4. IMPLEMENTATION

In this section we describe some implementation details regarding our solution. First, we present how our migration aware GC is implemented using G1, and then we discuss the internal architecture of the Migration Controller: its internal components and their purpose.

4.1 Migration Aware GC

ALMA is implemented in the OpenJDK HotSpot JVM 8 by modifying its G1 collector. ALMA adds a migration aware policy which takes advantage of the already existing G1 data structures to perform the heap analysis. The modifications done into the JVM are small, about 50 lines of code changed/inserted. This means that it is easy to port this modifications to other JVMs, if needed.

The G1 GC uses, internally, data regarding each heap region (e.g., number of free bytes, used bytes, references from object outside the region to objects inside the region, etc). Such data is gathered by the concurrent marking threads, which scan the heap, marking all live objects. Based on these information, the G1 GC is able to tell how many live bytes reside inside a particular region. As time goes by and regions get collected, the G1 GC is also able to estimate the time it will take to collect a particular region. This estimate is based on several factors: e.g., previous collections, number of inter-region references to update, number of live objects.

Once a migration is about to start, we take advantage of this information maintained by G1 to compute the optimal set of memory pages to include in the snapshot. This heap analysis is ruled by the equations described in Section 3.2. In other words, only regions in which garbage can be collected faster than transmitted through the network are collected.

4.2 Migration Controller

The Migration Controller component used in ALMA is implemented using CRIU. We modified CRIU to: i) support live migration through the network (original CRIU writes the process state to disk and uses NFS to provide remote migration), and ii) filter free mappings (reported by the JVMTI Agent) from the snapshot. CRIU runs in userspace and therefore, there is no need neither to modify the kernel nor to load any extra modules. Note that CRIU already handles the migration of process's resources such as open files, subprocesses, locks, etc.

The original CRIU (i.e., without our modifications) writes locally a process snapshot to disk which can then be migrated using a NFS share. In addition, the original CRIU does not provide live migration, the user being responsible for requesting the restoration of the process at the destination site. We have also modified CRIU to wait and react to new process snapshots, and to restore a process as soon the last snapshot is transferred to the destination site.

ALMA' Migration Controller extends CRIU by adding two new components: the Image Proxy (runs at the source site), a component that forwards process snapshots to the destination site, and the Image Cache (runs at the destination site), a component that caches process snapshots in memory until ALMA restores the process. Both Image Cache and Image proxy are auxiliary components that act as an in-memory snapshot caches. The benefits from using such components is twofold. First, both components keep snapshots in memory, which is much faster than writing and reading from disk (even for SSDs). Second, since the Image Proxy proactively forwards the snapshot to the Image Cache, we can start restoring the process while CRIU is still finishing the creation the snapshot and while the snapshot is still being transferred; in other words, process restoration is concurrent with the last snapshot creation.

5. EVALUATION

This section describes the evaluation of ALMA. We use two benchmark suites, SPECjvm2008 [39] and DaCapo 9.12 [3], and evaluate ALMA against:

- **CRIU** (a checkpoint and restore tool for Linux processes); this solution uses NFS to transfer snapshots from the source site to the destination site; thus, it does not take into consideration unused or unreachable memory and therefore, snapshots all memory allocated to a particular process;

- **JAVMM** [21], a recent system with the same goal: migrate Java applications. We compare this system to ALMA because they share the same goal (migrate Java applications) and also try to use garbage collection for reducing the size of snapshots. However, authors decided to implement JAVMM through system-VM migration, as opposed to the other evaluated systems (CRIU, ALMA-PS, and ALMA) that only migrate a specific process (enclosing a JVM); this naturally results in more network bandwidth usage and increased total migration time given that the initial snapshot contains the state of all processes running on the system as well as the Linux kernel itself; for this solution, we present the results that we extracted from Hou [21].[9] This means that we only present results regarding the downtime, network utilization, and total migration time for the scimark, derby and crypto benchmark applications. For the other benchmark applications and other experiments, we do not show any results since we could not perform experiments with JAVMM;

- **ALMA-PS** which is the ALMA solution using the GC and tuning proposed in JAVMM [21]; in other words, ALMA-PS uses the Parallel Scavenge GC with 1GB for young generation and 1GB for old generation, and forces one minor (young) collection upon snapshot creation (settings described in Hou [21]). We use this system to: i) isolate the performance benefits of using JVM migration versus system-VM migration (comparing it to JAVMM) and, ii) measure the performance benefits of using ALMA's migration-aware GC policy versus using the regular not migration-aware GC policy. Note that by using 1GB for the young generation, ALMA-PS ensures that all benchmarks applications' working set fits in the young generation. This represents the best scenario for this collector. Using less memory for the young generation would lead to some benchmark applications having data in the old generation, which would increase the size of the snapshots (as this generation is not collected by ALMA-PS).

Note that in ALMA we do not impose any configuration parameter on G1, letting the GC automatically adapt to the memory usage. This obviously leads to some data being promoted into the old generation. However, for ALMA this is not a problem since any region can be collected before creating a snapshot (see Section 3.2 for details).

All these three solutions were executed on a local OpenStack[10] installation, where we spawn system-VMs and perform the JVM migration between them (note that we could not conduct these experiments using JAVMM since we do not have access to it). The physical machines that host the system-VMs are Intel Xeon @ 2.13GHz with 40GB of RAM. Each of these physical machines (and thus the system-VMs) are connected using a 1Gbps network. We always spawn system-VMs in different physical nodes and we make sure that these physical nodes are being used only for our experiments. Each system-VM has 4 virtual CPUs and 2GBs of RAM except when we run experiments with ALMA-PS, which needs 4GBs of RAM to run (more details in Section 5.2).

With this environment setup, we approximate as much as possible the environment used for evaluating JAVMM (for which we present the results available in the paper [21]) and the environment used for evaluating ALMA. The amount of RAM and network bandwidth are the same for both JAVMM and ALMA; the virtual CPUs used in for evaluating JAVMM are slightly faster (AMD Opteron @ 2.2 GHz) than those used for evaluating ALMA (Intel Xeon @ 2.13 GHz). This gives a little advantage to JAVMM since the migration engine run faster when a migration needs to be performed.

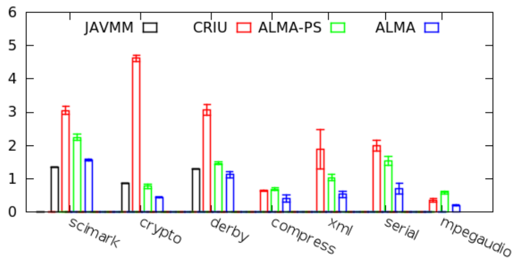
In this section we start by characterizing the applications included in both benchmarks w.r.t. memory utilization, and then present the evaluation results regarding: i) application downtime - amount of time that the application is stopped during migration; ii) network bandwidth usage - amount of data transferred through the network for migrating the application; iii) total migration time - time between the migration starts and the application resumes at the destination site; iv) application throughput - throughput difference between normal execution and execution including a migration; v) migration-aware GC performance overhead - the overhead imposed by our migration-aware GC versus the original G1; vi) ALMA performance with more resources - performance results (application downtime) when more cores and/or more network bandwidth are used.

5.1 Benchmark Description

Table 1 shows a summary of the memory characterization for the benchmarks used in our experiments. Other applications belonging to either SPEC or DaCapo benchmark are not presented because: i) some do not run in our JVM (as they fail to compile, for example the compiler benchmark application fro SPEC) or could not be migrated using CRIU

[9]The paper authors did not provide their solution for legal reasons. Having access to the source code would have enabled us to obtain more results.

[10]OpenStack is a cloud computing software platform. It is accessible at openstack.org.



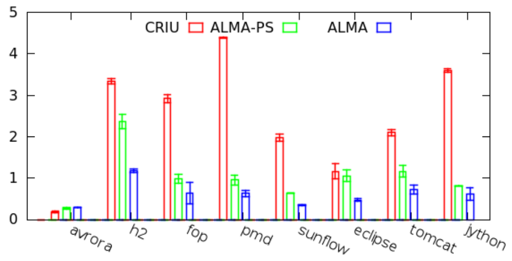


Figure 5: Application Downtime (seconds) for SPEC (left) and DaCapo (right) benchmarks

Benchmark	AR	GR	% Gbg	% Yng	HU
scimark	11.85	6.28	53.22	.21	481.40
derby	815.53	301.75	37.41	37.41	449.10
crypto	258.46	250.53	96.93	.57	349
compress	31.25	0.94	2.88	3.60	55.60
xml	740.94	614.98	82.72	82.38	149.30
serial	585.86	181.62	30.92	30.87	187.90
mpegaudio	86.07	66.27	76.75	86.42	24.30
avrora	1.94	1.44	74.38	75.22	22.60
h2	405.14	121.54	29.62	34.75	423.00
fop	223.66	140.20	60.23	63.33	176.00
pmd	130.63	84.91	64.57	68.45	232.30
snuflow	457.03	389.33	82.63	82.18	142.20
eclipse	9.05	4.80	53.40	57.67	107.50
tomcat	61.05	53.11	87.49	88.35	127.90
jython	794.59	659.51	82.54	82.54	178.10

Table 1: Benchmark Analysis for SPEC and DaCapo

(for example the tradebeans and tradesoap benchmark applications from DaCapo); ii) the others provide similar performance results and this leads to no new conclusions (in other words, the applications that we present are representative regarding the concerns of this evaluation).

The top rows (Table 1) refer to SPEC benchmark applications while the bottom rows refer to DaCapo benchmark applications. For each application we present: i) allocation rate (AR), the amount of data allocated by the application per unit of time (MB/s); ii) garbage creation rate (GR), i.e. the amount of dead data allocated per unit of time (MB/s); iii) percentage of allocated heap space which is unreachable (% Gbg) upon a minor collection; iv) percentage of used heap space which belongs to the young generation [11] (% Yng) upon a minor collection; v) heap usage (HU), i.e. amount of application data in the heap (this includes both live and dead objects) upon a minor collection.

Each one of these metrics is obtained by looking into G1 GC logs (we did not modify the logging infrastructure for the JVM) produced by running each benchmark application. We analyze the last GCs runs before migration starts. This ensures that these metrics represent the state of the JVM when the migration is performed. For example, to obtain the allocation rate (AR), we consider the last two consecutive allocation failure triggered [12] GCs before the migration starts. We take the heap usage after one GC and the heap usage right before the next one and divide it by the time elapsed between the two GCs. All values are averages of at least 5 runs (we enforce additional runs when outliers are detected).[13] This is also true for all values presented during the evaluation section. We found that these metrics are stable at least during the migration process. This means that: i) the size of the application working set (i.e., the amount of live data left after the GC runs) is stable, and ii) GCs run periodically when the percentage of free space approaches zero, setting the heap usage back to the working set size.

Note that all metrics in Table 1 are obtained using the G1 GC. Other GCs might produce slightly different results because of the different heap partitioning and different collection techniques. Nevertheless, for all generational collectors (i.e., collectors belonging to the same family of G1), the conclusions taken from Table 1 also apply.

5.2 Application Downtime

In this section, we present the results obtained when measuring the application downtime: time span between the moment the JVM is stopped at the source site and the JVM starts at the destination site. In other words, the time interval during which the application does not run (neither on the source site nor on the destination site).

These results were obtained for all systems using a total of 15 applications (presented in Table 1). For each experiment we start the application at the source site, let it run for 1 minute and then migrate the enclosing process (JVM included) to the destination site. We found that 1 minute is enough for all applications to warm-up and to reach their maximal resource consumption (mainly CPU and memory). All applications run at least five times; we show both the average and the standard deviation for these runs. More runs are performed when outliers are detected. This procedure is also used in the next sections.

Figure 5 shows the results for the application downtime. For each benchmark application, results are grouped, having one bar for system (from left to right): JAVMM, CRIU, ALMA-PS, and ALMA. This organization of columns is also used in subsequent figures. We note that CRIU is the migration solution with worse downtime. This is because it snapshots all the process memory, not taking into account unused or unreachable memory, and also because it uses NFS to transfer all snapshots. Regarding ALMA-PS, the measured downtime is much better than CRIU's (except

[11]The young generation comprehends all heap regions which contain recently allocated objects. Objects that survive at least two garbage collections are promoted (to the old generation) and no longer belong to the young generation.

[12]An allocation failure happens when no more free memory exists to satisfy an allocation request. This event triggers a garbage collection.

[13]We execute each experiment 5 times given that the results obtained remain stable even with more runs.

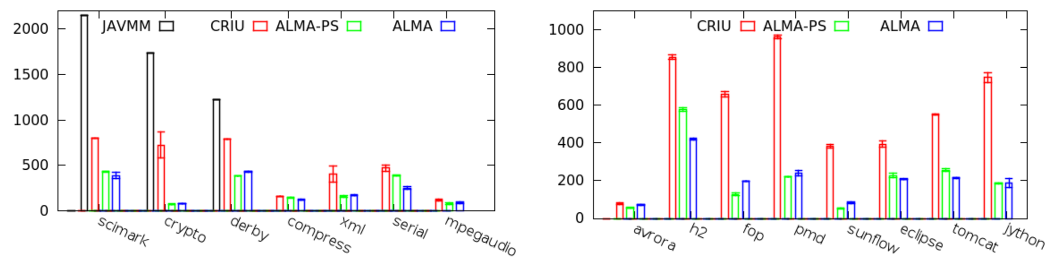


Figure 6: Network Bandwidth Usage (MBs) for SPEC (left) and DaCapo (right) benchmarks

mpegaudio) but still worse than ALMA. The reasons are the following. First, ALMA-PS initializes the young generation with the size of 1GB. This forces the process of taking a snapshot and restoring it to handle 1GB of memory. If this young generation size pre-condition was not imposed, taking and restoring a snapshot would handle potentially much less memory (the actual amount of memory used by the application). The mpegaudio application is a clear example: it uses around 24MB of memory (Table 1). Therefore, the overhead of handling 1GB instead of 24MB (approximately) makes ALMA-PS perform worse than ALMA and CRIU. In ALMA, we do not impose such young generation size precondition and therefore, this overhead does not exist, i.e, we only process the amount of memory that the application actually uses. Second, old generation garbage is not collected by ALMA-PS, which only forces a minor (young) collection. This way, all garbage that resides in the old generation will be transferred to the destination site. This is specially noticeable in h2 (DaCapo) and scimark (SPEC), for example.

Regarding ALMA and JAVMM (still see SPEC results from Figure 5), ALMA achieves better results in 2 out of 3 applications. Considering that the environment in which both systems are evaluated (JAVMM uses slightly faster CPUs), we expect ALMA to perform even better versus JAVMM if running in the same exact environment.

Another important difference between ALMA and JAVMM is that ALMA runs in the same system-VM as the application while JAVMM runs directly on the physical machine (that hosts the system-VM containing the application). This means that ALMA might take a little longer to take a snapshot compared to JAVMM if the system-VM CPU is exhausted by the application. Therefore, our environment represents a worst case scenario for ALMA since the CPU is exhausted by the applications.

Taking into account these application downtime results and the benchmarks applications characterization shown in Table 1, it is possible to draw some general conclusions. High allocation ratio does not imply a high application downtime. For example, the jython application has one of the highest allocation ratio but the corresponding application downtime is not among the highest ones. The highest downtime (and therefore the most costly applications to migrate) are the ones with high allocation ratio and low garbage creation ratio; in other words, applications with higher long-lived objects creation ratio lead to higher application downtime. Examples of such applications are h2, scimark, and derby. Even for these worse cases, the downtime with ALMA is less than CRIU, ALMA-PS. ALMA and JAVMM achieve similar downtime results for scimark and derby.

5.3 Network Bandwidth Usage

In this section we present the evaluation results of ALMA regarding the network bandwidth usage, i.e., the amount of data transferred through the network to migrate an application (see Figure 6). JAVMM clearly yields the worse results, even worse than CRIU. This is due to the fact that JAVMM migrates a whole system-VM. Note that, since the goal is to migrate an application from one machine to another, ALMA only migrates the application process (including the JVM) while JAVMM migrates the whole system-VM.

CRIU follows JAVMM as it does not remove unreachable data from the snapshots; thus, it transfers more data than ALMA and ALMA-PS. Comparing ALMA and ALMA-PS, ALMA is superior in 10 out of 15 applications. The only benchmark applications where PS achieves better results are the following: derby, avrora, fop, pmd, and sunflow. The common particular feature of these applications comes is that most garbage (collected before taking the snapshot) originates from the young generation. The better results of ALMA-PS are due to the fact that its Parallel Scavenge collector is more efficient collecting the young generation than G1 (which is used by ALMA). This comes from the fact that in G1, although objects are all in the young generation, they occupy several regions which imply handling inter-region references (stored in card tables) and, consequently, need more GC effort to collect all young garbage; such inter-region references do not exist in the GC of ALMA-PS as all young objects are in the same region (young generation).

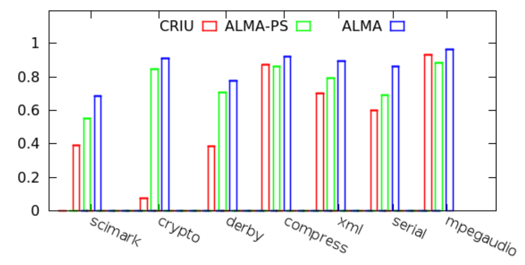
In general, applications that use more memory tend to consume more memory bandwidth during migration. From Table 1 and Figure 6, we may conclude that applications with more heap usage and less garbage percentage (e.g. scimark, derby, h2) result in increased network bandwidth usage.

5.4 Application Throughput

Figure 7 shows the normalized results for the throughput of CRIU, ALMA-PS, and ALMA. These results are obtained by sampling the benchmark throughput (number of operations) each five seconds. The measured number of operations is specific to each benchmark, i.e., one cannot compare the number of operations of two different benchmarks. The only possible comparison (which we do) is the number of operations between multiple runs of the same benchmark.

The average throughput in normal execution of the benchmarks represents the value one in Figure 7. The normalized throughput for each system, represents the throughput achieved when the migration occurred.

The throughput results have a strong correlation with the



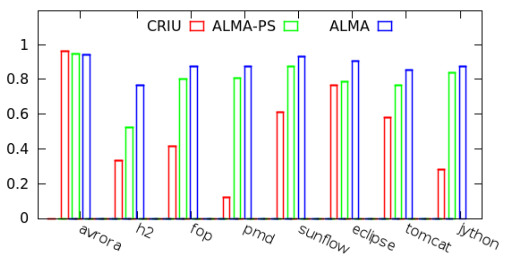


Figure 7: Application Throughput (normalized) for SPEC (left) and DaCapo (right) benchmarks

application downtime (see Section 5.2). In other words, there is no relevant slowdown in the application throughput after starting at the destination site (i.e., the application is already running at a normal throughput and does not need to warm-up). Therefore, most conclusions derived from analyzing application downtime are still applicable to application throughput.

In short, CRIU is clearly the solution with lower throughput for almost all benchmarks, followed by ALMA-PS. ALMA is the solution with highest throughput, which is above 80 % of the normal throughput for almost all benchmarks. The benchmark with lower throughput is scimark, which is a CPU and memory bound benchmark, reason why the migration of this specific benchmark produces a severe throughput slowdown for all systems. Compared to ALMA-PS, ALMA achieves higher throughput in 14 out of 15 bechmark applications.

5.5 Total Migration Time

This section presents the results for total migration time, i.e., the time between a process migration is requested and the process resuming at the destination site.

Once again (see Figure 8), JAVMM performs worse than all others. This results from the fact that JAVMM migrates a whole system-VM while the other solutions migrate only a process (the JVM). Regarding CRIU, the results are proportional to those presented in Figure 6 (bandwidth usage) since the total migration time mostly comes from transferring snapshot data.

ALMA performs better than any other solution. ALMA-PS, which shows better results for network bandwidth usage (Figure 6) in some cases, has the drawback of forcing a 1GB young generation space; this increases the cost of each snapshot and restoration of the process. Reducing the size of the young generation wouldn't help either because it would lead to more young collections and push more objects into the old generation, which is not collected by ALMA-PS before a migration. As with application downtime, the mpegaudio application provides a clear example of this overhead: ALMA-PS achieves the worst performance because the process heap size is very small but the young generation is still set to 1GB.

Table 2 shows the average of each one of the previously presented evaluation results (application downtime, network usage, total migration time, and throughput) of each solution normalized to ALMA. We could not measure the throughput for JAVMM since we could not reproduce our experiments with JAVMM.

ALMA clearly achieves the best performance in all three metrics. Compared to JAVMM, ALMA: i) improves the downtime by 13%, ii) network usage is 4.42 times lower, and iii) total migration time is 5.69 times faster.

Metric	CRIU	JAVMM	ALMA-PS
Downtime	3.58	1.13	1.68
Net. Usage	2.86	4.42	1.41
Total Migr. Time	2.57	5.69	1.06
Throughput	0.61	NA	0.89

Table 2: Performance Results Normalized to ALMA

Benchmark	G1 GC	Migr. GC	Migr. GC (Norm.)
scimark	19 ms	18 ms	0.94
derby	7 ms	7 ms	1.00
crypto	12 ms	8 ms	0.67
compress	2 ms	3 ms	1.50
xml	6 ms	11 ms	1.83
serial	2 ms	4 ms	2.00
mpegaudio	3 ms	7 ms	2.33
avrora	5 ms	12 ms	2.40
h2	102 ms	36 ms	0.35
fop	19 ms	26 ms	1.36
pmd	17 ms	22 ms	1.29
sunflow	5 ms	6 ms	1.20
eclipse	14 ms	38 ms	2.71
tomcat	14 ms	17 ms	1.21
jython	5 ms	13 ms	2.60

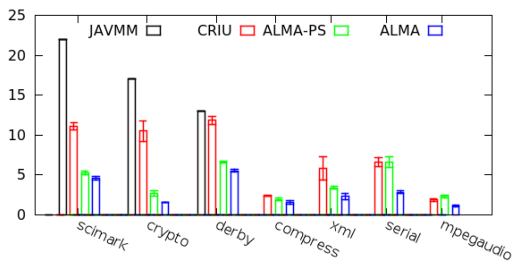
Table 3: ALMA Migration Aware GC Overhead Compared to G1 GC for SPEC and DaCapo

ALMA-PS presents the closest performance results when compared to ALMA. Table 2 shows that ALMA achieves 41% better performance compared to ALMA-PS regarding network usage, 68% regarding downtime, 6% regarding total migration time, and 11% for the application throughput (including migration).

5.6 Migration Aware GC Overhead

As already said, ALMA's migration-aware GC collects all heap regions whose `GCRate` is greater than the network bandwidth (see Section 3). This section shows the performance penalty of running such a migration-aware GC. Table 3 presents the average duration of: i) column G1 GC - each collection done with the default G1 GC (as if there was no ALMA), ii) column Migr. GC - using the migration aware policy described in Section 3), and iii) column Migr. GC (Norm.) - the normalized values for the migration aware GC w.r.t. the G1 GC.

As expected, a migration-aware GC (as it happens in



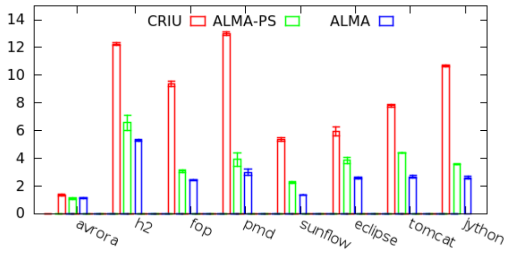


Figure 8: Total Migration Time (seconds) for SPEC (left) and DaCapo (right) benchmarks

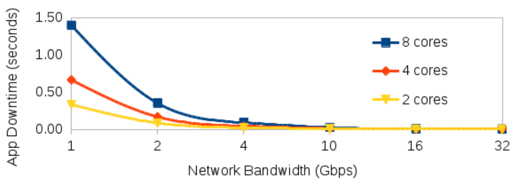


Figure 9: ALMA Application Downtime With More Cores Versus More Network Bandwidth

ALMA) takes longer than a G1 GC in 12 out of 15 applications. This is due to the fact that ALMA migration aware policy selects more regions to collect than the default G1 policy (which tries to minimize the application pauses), and thus, takes more time to finish.

In 3 applications (h2, scimark, and crypto) this is not true, i.e. the migration aware GC is faster than G1 GC. In these particular cases, this is due to the fact that G1 performs several full GCs[14] because these applications allocate large blocks of memory which occupy most of the heap leading to allocation failures. For this reason, ALMA migration aware GC takes less time than the average default G1 GC.

Nevertheless, even the cases where the migration-aware GC is slower than G1 GC, the difference in time is very small compared to the application downtime during a migration. In other words, an increase of a few dozens of milliseconds in GC duration will have a negligible impact on the migration downtime.

5.7 ALMA with More Resources

In this last experiment, we study the performance impact of ALMA on the application downtime, when increasing the number of cores used by the application and the network bandwidth available (used to transfer the application snapshots). For this experiment only, we used 3 different system-VMs (also hosted in our local OpenStack installation) with 2, 4, and 8 cores. We performed this experiment with only one application, crypto. We chose this particular application because it dirties memory at a constant rate, and does not concentrate memory operations in a specific heap area (which is frequent in derby, for example). This behavior evidenced by crpyto represents the worst case for ALMA; other applications of the benchmarks are much less demanding memory wise.

$$Downtime = \frac{SizeIncSnapshot}{NetBandwidth} \tag{6}$$

$$SizeIncSnapshot = \frac{SizeInitSnapshot}{NetBandwidth} * DirtyRate \tag{7}$$

$$Donwtime = \frac{SizeInitSnapshot}{NetBandwidth^2} * DirtyRate \tag{8}$$

The results are shown in Figure 9. Since our installation only has 1Gbps, we estimate the remaining values with more network bandwidth. The estimated values are obtained as follows (see Eqs. 6,7, and 8): i) we start by measuring the application dirty rate (by measuring the size of the incremental snapshot) using 2, 4, and 8 cores. Then, with the size of the initial snapshot divided by the network bandwidth we get the time needed to transmit the initial snapshot. Multiplying it with the dirty rate, we get the size of the incremental snapshot (Eq. 7). This enables us to estimate the application downtime (by replacing Eq. 7 in Eq. 6 to obtain Eq. 8). We also consider that part of both the initial and the incremental snapshots are filtered as garbage (this percentage is taken from Table 1).

Figure 9 clearly shows that increasing the number of cores, results in increasing the application downtime. This comes from the fact that the application will dirty memory faster. On the other hand, when we increase the amount of network bandwidth used for migration, the application downtime drops because the application has less time to dirty memory (since the snapshot gets transferred faster).[15] One important conclusion to take from our experiment is that the application downtime (see Eq. 8) is: i) proportional to the number of cores (as it is multiplied by the application dirty rate), and ii) inversely proportional to the square of the network bandwidth (as it is divided by the square of the network bandwidth). In other words, doubling the number of cores will double the application downtime but doubling the network bandwidth will reduce the downtime to one quarter of its initial value. Note that these conclusions only hold for ALMA which has one initial snapshot and one incremental snapshot (taken right after transmitting the initial snapshot).

[14] A full GC happens when the heap has no more free memory to satisfy an allocation request, and the G1 collection of the young generation fails. In a full GC, the entire heap is collected and compacted. This is a particularly costly operation.

[15] Note that memory gets dirty by an application running on the source site while the first snapshot is transferred to the destination site.

6. RELATED WORK

Solutions to migrate applications can be characterized along two aspects: i) when the execution control is transferred to the destination site: before (pre-copy) or after (post-copy) the memory is migrated, and ii) take advantage (or not) of the state of the application (e.g. avoid transferring unused memory, or look for memory similarities to use compression).

Pre-copy [44, 7], is the most common technique to transfer a system-VM. The bulk of the system-VM's memory pages is transferred to the destination site, while the application keeps executing. If a transmitted page is dirtied in the mean time, it gets re-transmitted in the next round. This produces an iterative process in which dirtied pages are transferred until: i) a small writable working set (WWS) [7] is identified, or ii) a preset number of iterations is reached. When one of these conditions is met, the system-VM is suspended, and the last modified pages and the processor state are transferred. After this final step, the system-VM is resumed on the destination site.

In Hines [20], the opposite approach (post-copy [48]) was proposed. Using a post-copy strategy, the control is transferred to the destination site and all the remaining pages are lazily transferred when needed (the system-VM running on the destination site will page fault, triggering the page transfer). To avoid the transmission of free pages, the authors employ a dynamic self ballooning mechanism to force the guest OS to free pages before the migration starts. Despite the obvious overhead of waiting for the transmission of memory pages each time a page fault happens, the authors advocate that their approach results in less network overhead since each page is transferred once.

In Vogt [45], authors use a mix of both pre-copy and post-copy for checkpointing applications. While the hot WWS is copied when the checkpoint is triggered, pages not included in the WWS are lazily copied upon modification (this process is very similar to Linux's `fork`).

Regardless of when memory is transmitted, there are several mechanisms proposed to reduce the amount of memory to transmit. Compression [23] is used to reduce the size of the memory snapshots to transmit in a pre-copy approach; the compression cost can be dynamically tuned according to the application dirty rate. Memory deduplication [10, 11] has also been proposed to reduce the number of memory page duplicates when migrating several system-VMs at the same time. A similar technique was proposed by Knauth [27], where the authors propose reusing data from previous checkpoints to when a system-VM returns to a site where it was in the past. Skipping file caches and free pages [28] (soft pages) was also suggested; the system scans memory pages when migration is triggered and marks soft pages to avoid transferring them.

Others have taken a different approach to reduce the size of the memory to transmit by analyzing and taking advantage of the state of the application. In Giuffrida [19], the authors trace application resources (allocated memory, for example) and transmit only the usable data instead of transmitting all the memory that the application can use. In Kawachiya [26] and Hou [21] the authors propose looking into the JVM heap and forcing a Garbage Collection (GC) to reduce the used memory. In Garrochinho [41], the authors look into the internals of the JikesRVM and extract only the necessary information to restore the application at the destination site.

Our approach for process live migration (which is targeted to migrate JVMs) inherits some aspects of the previous solutions. We also use the GC to minimize the amount of process state to transmit. However, as opposed to i) Kawachiya [26] which forces a full mark-and-sweep collection (thus decreasing the responsiveness and increasing the down time of the application), and ii) Hou [21] which only performs a minor collection to clean the young generation (and leaves the rest of the heap with potentially lots of garbage), in ALMA we add a new migration aware policy to G1 GC to: i) provide useful information about the liveliness of the heap and the cost of collecting it, and ii) enforce a migration aware GC operation in which only the heap regions whose *GCRate* is greater than the network bandwidth are are effectively considered for collection.

Regarding process migration (not system-VM), most systems either require re-linking with some process migration tool library ([5, 47, 43]), modify the application code ([32, 36, 37]), or require modification to existing Kernels ([4, 15, 18]).

Other systems such as [30, 25] use CRIU to provide fast OS updates (via checkpoint and restoring processes) [25] and to migrate preempted jobs in Hadoop clusters [30]. In Silva [40], authors propose replaying of JVMs (coupled with several optimization techniques such as dynamic pruning of unreachable data, and a lightweight checkpoint mechanism) as a way to migrate JVMs. This approach has two problems: i) serialization of memory accesses has a severe impact on application's performance, and ii) the logs used for replaying the server state can easily achieve significant size, thus increasing the amount of data that needs to be transmitted to the destination site. We could not find any other published work on process migration taking advantage of application state (JVM state, in our scenario) to reduce the costs of migration. We also could not find any recent and relevant live process remote migration system to compare with ALMA.

In short, ALMA's distinctive feature is the possibility to dynamically calculate the optimal set of memory pages to transfer by comparing the cost of collecting each heap region with the potential benefits versus the network bandwidth. By harnessing reachability information provided by the GC, ALMA is able to identify all dead or unused data and thus, reduce the number paces included in the snapshot.

7. CONCLUSIONS

This paper presents a novel approach to migrate JVM applications by analyzing the G1 heap to identify heap fragments which should be collected before migration. ALMA separates the migration of the application from the hosting system (system-VM) by migrating only the application process instead of migrating the whole hosting system-VM. ALMA is evaluated using two well-known benchmark suites (SPECjvm2008 and DaCapo 9.12) and shows very good performance results. The code is publicly available at https://github.com/rodrigo-bruno/ALMA-JMigration.

8. 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 scholarship SFRH/BD/103745/2014.

9. REFERENCES

[1] A. W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989.

[2] J. Armstrong and R. Virding. One pass real-time generational mark-sweep garbage collection. In *Memory Management*, pages 313–322. Springer, 1995.

[3] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis.

[4] M. Bozyigit, K. Al-Tawil, and S. Naseer. A kernel integrated task migration infrastructure for clusters of workstations. *Computers & Electrical Engineering*, 26(3):279–295, 2000.

[5] M. Bozyigit and M. Wasiq. User-level process checkpoint and restore for migration. *ACM SIGOPS Operating Systems Review*, 35(2):86–96, 2001.

[6] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, Nov. 1970.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[8] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[9] T. Das, P. Padala, V. N. Padmanabhan, R. Ramjee, and K. G. Shin. Litegreen: Saving energy in networked desktops using virtualization. In *USENIX annual technical conference*, 2010.

[10] U. Deshpande, B. Schlinker, E. Adler, and K. Gopalan. Gang migration of virtual machines using cluster-wide deduplication. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 394–401. IEEE, 2013.

[11] U. Deshpande, X. Wang, and K. Gopalan. Live gang migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 135–146. ACM, 2011.

[12] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM, 2004.

[13] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123. ACM, 1993.

[14] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 274–284, New York, NY, USA, 2000. ACM.

[15] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Softw., Pract. Exper.*, 21(8):757–785, 1991.

[16] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614, March 2014.

[17] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, page 7. ACM, 2011.

[18] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9. IEEE Computer Society, 2005.

[19] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum. Mutable checkpoint-restart: automating live update for generic server programs. In *Proceedings of the 15th International Middleware Conference*, pages 133–144. ACM, 2014.

[20] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60. ACM, 2009.

[21] K.-Y. Hou, K. G. Shin, and J.-L. Sung. Application-assisted live migration of virtual machines with java applications. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 15:1–15:15, New York, NY, USA, 2015. ACM.

[22] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High performance virtual machine migration with rdma over modern interconnects. In *Cluster Computing, 2007 IEEE International Conference on*, pages 11–20. IEEE, 2007.

[23] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. Live virtual machine migration with adaptive, memory compression. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[24] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[25] S. Kashyap, C. Min, B. Lee, T. Kim, and P. Emelyanov. Instant os updates via userspace checkpoint-and-restart. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[26] K. Kawachiya, K. Ogata, D. Silva, T. Onodera, H. Komatsu, and T. Nakatani. Cloneable jvm: a new approach to start isolated java applications faster. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 1–11. ACM, 2007.

[27] T. Knauth and C. Fetzer. Vecycle: Recycling vm checkpoints for faster migrations. In *Proceedings of the 16th Annual Middleware Conference*, pages 210–221. ACM, 2015.

[28] A. Koto, H. Yamada, K. Ohmura, and K. Kono. Towards unobtrusive vm live migration for cloud computing platforms. In *Proceedings of the*

Asia-Pacific Workshop on Systems, page 7. ACM, 2012.

[29] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[30] J. Li, C. Pu, Y. Chen, V. Talwar, and D. Milojicic. Improving preemptive scheduling with application-transparent checkpointing in shared clusters. In *Proceedings of the 16th Annual Middleware Conference*, pages 222–234. ACM, 2015.

[31] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[32] M. Litzkow, T. Checkpointing, T. Process Migration for MPInbaum, J. Basney, and M. Livny. *Checkpoint and migration of UNIX processes in the Condor distributed processing system*. Computer Sciences Department, University of Wisconsin, 1997.

[33] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[34] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 23–32. ACM, 2007.

[35] R. Nathuji and K. Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 265–278. ACM, 2007.

[36] T. Osman and A. Bargiela. Process checkpointing in an open distributed environment. In *Proceedings of European Simulation Multiconference, ESM*, volume 97, 1997.

[37] J. S. Plank, M. Beck, G. Kingsley, and K. Li. *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

[38] T. Printezis and D. Detlefs. *A generational mostly-concurrent garbage collector*, volume 36. ACM, 2000.

[39] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. Specjvm2008 performance characterization. In *Computer Performance Evaluation and Benchmarking*, pages 17–35. Springer, 2009.

[40] J. M. Silva, J. Simão, and L. Veiga. Ditto–deterministic execution replayability-as-a-service for java vm on multiprocessors. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 405–424. Springer, 2013.

[41] J. Simão, T. Garrochinho, and L. Veiga. A checkpointing-enabled and resource-aware java virtual machine for efficient and robust e-science applications in grid environments. *Concurrency and Computation: Practice and Experience*, 24(13):1421–1442, 2012.

[42] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 53. IEEE Press, 2008.

[43] G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 526–531. IEEE, 1996.

[44] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. *Preemptable remote execution facilities for the V-system*, volume 19. ACM, 1985.

[45] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. Tanenbaum, and C. Giuffrida. Speculative memory checkpointing. In *Proceedings of the 16th Annual Middleware Conference*, pages 197–209. ACM, 2015.

[46] P. R. Wilson and T. G. Moher. Design of the opportunistic garbage collector. In *ACM SIGPLAN Notices*, volume 24, pages 23–35. ACM, 1989.

[47] V. C. Zandy, B. P. Miller, and M. Livny. Process hijacking. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 177–184. IEEE, 1999.

[48] E. Zayas. Attacking the process migration bottleneck. In *ACM SIGOPS Operating Systems Review*, volume 21, pages 13–24. ACM, 1987.